NTNU

Norwegian University of
Science and Technology

Low Energy AES Hardware for Microcontroller

Øivind Ekelund

Master of Science in Electronics
Submission date: July 2009
Supervisor: Per Gunnar Kjeldsberg, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

Cryptographic algorithms are commonly used with microcontrollers today. As performance demands increase, so does the need for dedicated cryptographic hardware inside the microcontroller itself. While area and speed are two essential parameters, minimizing the energy per encryption/decryption has become increasingly important. The Advanced Encryption Standard (AES) is one of the most used symmetric cryptographic ciphers. This thesis will focus on hardware implementation of AES, tailored for low energy microcontrollers.

Main objects:

- Evaluate existing AES software solutions to serve as a performance benchmark. The ARM Cortex M3 processor should be used.

- Evaluate existing AES hardware implementations with regards to energy, area and speed.

- Implement, in HDL code, a low energy AES hardware implementation suited for microcontrollers, based on the initial evaluations and a cost/performance analysis.

Assignment given: 15. January 2009
Supervisor: Per Gunnar Kjeldsberg, IET

Preface

This Master thesis is a continuation of an earlier project [38] which gave an introduction to the Advanced Encryption Standard and underlying theory. The submodules MixColumns and SubBytes were given a close look and several implementations of these were explored in [38]. Using this as a basis, a complete AES core has been developed in this thesis. As this thesis is based on a previous project, parts of Chapters 2 and 3 are similar to corresponding Chapters in [38], but some adjustments and extensions have been made.

To fully understand the contents of this thesis, the reader should have basic knowledge about electronics and digital design as well as binary arithmetics.

Parts of this work has been done at Energy Micro's premises in Oslo and I would like to thank the employees at Energy Micro, especially my supervisor, Rasmus Larsen, for guidance and helpful input during this work. I would also like to thank my supervisor at NTNU, Per Gunnar Kjeldsberg, for support throughout the process of writing this thesis.

Øivind Ekelund, July 2009, Moss

Abstract

Cryptographic algorithms, like the Advanced Encryption Standard, are frequently used in todays electronic appliances. Battery operated devices are increasingly popular, creating a demand for low energy solutions. As a microcontroller is incorporated in virtually all electronic appliances, the main objective in this thesis is to evaluate possible hardware implementations of AES and implement a solution optimized for low energy consumption, suited for incorporation in a microcontroller. A good cost/performance balance is also a design goal.

An existing solution based on a 32 bit architecture with support for 128 bit keys was chosen as a basis and altered in order to lower area and energy consumption. The alterations yielded a 13.6% area reduction as well as 14.2% and 3.9% reduction in energy consumption in encryption and decryption mode, respectively. In addition to alterations in the datapath, low energy techniques like clock gating and numerical strength reduction has been applied in order to further lower the energy consumption.

The proposed architecture was also extended in order to accommodate 256 bit keys. Although this increased the area by 9.2%, the power consumption was still reduced by 7.6% and 1.3% in en- and decryption, compared to the architecture chosen as basis.

As AES is an algorithm which easily can be parallelized, a high throughput solution utilizing a 128 bit datapath was implemented. This AES module is able to process 372.4 Mbps at an operating frequency of 32 Mhz and is based on the same architecture as the 32 bit datapath solution. In addition, this implementation yielded excellent energy per encryption figures, 24.5% lower than the 32 bit solution.

The alternative to performing AES in a dedicated hardware module is to perform it using software. In order to have a basis for comparison, a software solution optimized for 32 bit architectures was implemented. Simulations show that the energy consumption attained when performing AES in the proposed hardware module is approximately 2.3% of what a software solution would use. In addition, the throughput is increased by a factor of 25.

The architecture proposed in this thesis combines relatively high throughput with modest demands to area and low energy per encryption.

Contents

List of Figures

List of Tables

Abbreviations

AES	-	Advanced Encryption Standard
AES128	-	AES using 128 bit key
AES192	-	AES using 192 bit key
AES256	-	AES using 256 bit key
ALU	-	Arithmetic Logic Unit
CBC	-	Cipher Block Chaining
CFB	-	Cipher Feedback
CISC	-	Complex Instruction Set Computer
CMOS	-	Complementary metal–oxide–semiconductor
CPU	-	Central Processing Unit
CTR	-	Counter
DMA	-	Direct Memory Access
DMAC	-	Direct Memory Access Controller
ECB	-	Electronic Code Book
FSM	-	Finite State Machine
GPIO	-	General Purpose I/O
HW	-	Hardware
I/O	-	Input/Output
LSB	-	Least Significant Bit
LUT	-	Look Up Table
MCU	-	Microcontroller Unit
MOSFET	-	Metal–Oxide–Semiconductor Field-Effect Transistor
MSB	-	Most Significant Bit
MUX	-	Multiplexer
NIST	-	National Institute for Standards and Technology
NMOS	-	n-channel MOSFET
NSR	-	Numerical Strength Reduction
OFB	-	Output Feedback
RAM	-	Random Access Memory
RISC	-	Reduced Instruction Set Computer
RO	-	Read Only
ROM	-	Read Only Memory
RTL	-	Register Transfer Level
SPI	-	Serial Peripheral Interface
SW	-	Software
UART	-	Universal Asynchronous Receiver/Transmitter
XOR	-	Exclusive or

Glossary

Affine transformation	-	Transformation consisting of multiplication by a matrix followed by addition of a vector
Data column	-	A column (32 bits) in the state
GF(2^n)	-	Galois Field (i.e. finite field) with 2^n elements
Key column	-	A column (32 bits) in the cipher- or roundkey
Nonce	-	A data vector not expected to recur
N_b	-	Number of columns in a state, in AES N_b=4
N_k	-	Number of 32-bit words in the Cipher Key, in AES N_k=4,6 or 8
N_r	-	Number of rounds, in AES N_r=10,12 or 14
State	-	A 4x4 block of bytes, contains the data in AES
Sbox	-	Substitution box. Used in SubBytes

Chapter 1

Introduction

Electronic appliances surround us in our everyday lives. Everything from our mobile phone to our car keys are electronic devices. In virtually all these electronic devices, a microcontroller is incorporated in order to implement some sort of functionality. A microcontroller could perform any task, for instance turning on the device when a button is pressed, or more complex tasks like data processing. The microcontrollers are often a significant contributer to the overall energy consumption, and as the number of battery operated appliances increase, so does the need for low energy solutions as prolonged battery life is highly desired.

Energy Micro is a Norwegian semiconductor company founded in 2007. Their goal is to develop the worlds most energy friendly microcontrollers based on the 32 bit ARM Cortex M3 processor. They plan to reach this goal by means of numerous low power modes and possibility for autonomous operation without CPU intervention through a wide range of peripherals.

Communication between nodes is frequently used in electronic systems (e.g., mobile phones and smart cards) and in many applications it is crucial that this communication is carried out in a secure manner, in other words: The data transmitted should not be readable to anyone else than the data is intended for. Numerous cryptographic algorithms have been developed for this purpose, and in the later years, The Advanced Encryption Standard, AES, has become one of the most widely used algorithms. AES is a symmetric block cipher processing 128 bits at a time using 128-, 192-, or 256 bit keys. It is considered a very secure algorithm and it is predicted to be widely used in many years to come.

AES can easily be realized in software, but this could lead to unnecessary use of energy and time due to the fact that execution on a CPU is accompanied by energy- and time consuming memory accesses and overhead instructions, like updating loop indices and calculating memory addresses. If AES is performed in a dedicated hardware module, most of the energy will be consumed performing computations defined by the algorithm. In addition to reduction in energy consumption, the throughput achievable in a dedicated hardware solution is superior to what can be achieved using software.

Numerous hardware solutions exists today, some optimized for low area, some optimized for high throughput and some optimized for low energy consumption. The aim of this thesis is to evaluate possible solutions and implement a hardware solution based on the evaluation. The implementation should be tailored for microcontrollers and have low energy consumption as well as a good cost/performance balance.

Thesis outline

Chapter 2 in this thesis will give an introduction to the Advanced Encryption Standard algorithm and underlying theory. After this, theory concerning power consumption and approaches for limitation of this is presented in Chapter 3. A brief introduction to methods for power estimation, both in software and hardware, will also be given. The theory part of the thesis will then be concluded with Chapter 4 giving a short introduction to microcontrollers.

Presentation and evaluation of a software implementation of AES optimized for 32 bit architectures will be given in Chapter 5 before two existing hardware implementations are presented in Chapter 6.

In this thesis, an AES implementation has been developed with emphasis on minimizing energy per encryption while maintaining a good cost/performance balance. The 32 bit architecture is based on a previous solution, but three major changes were made resulting in improvements both in area and energy/power consumption. In addition, energy saving approaches like clock gating and numerical strength reduction has been utilized to further reduce area and energy/power consumption. This implementation will be presented in Chapter 7 along with a parallelized 128 bit solution yielding high throughput and excellent energy per encryption.

Description of the synthesis and verification process will then be presented in Chapter 8 before evaluation of the different implementations is done in Chapter 9. The thesis ends with conclusions and suggestions for future work in Chapter 10.

Main contributions

The main contributions in this thesis are:

- An AES core supporting 128- and 256 bit keys, optimized of low energy/encryption while maintaining a good cost/performance balance. The AES core consumes 8.03 nJ/encryption and an area equivalent to 7536 NAND2 gates.
- A high throughput AES core supporting 128 bit keys, optimized low energy/encryption. This implementation consumes 5.63 nJ/encryption and is able to process 372.4 Mbps @ 32 Mhz.
- Application of numerical strength reduction in AES resulting in area and energy savings.
- A software implementation of AES optimized for 32 bit architectures.

Chapter 2

Advanced Encryption Standard

2.1 Rijndael

In January 1997 the National Institute for Standards and Technology (NIST) organized a contest for the new Advanced Encryption Standard (AES). Three years later, in October 2000, the algorithm Rijndael was announced the winner. Rijndael was designed by the Belgians Joan Daemen and Vincent Rijmen and was a "surprise winner" because many observers did not believe that the US would adopt an encryption standard developed by non-US citizens. Rijndael won the contest due to its elegance, efficiency, security, and principled design.

Rijndael is a symmetric block cipher which means that it maps plaintext blocks to ciphertext blocks and that the same key is used in both en- and decryption. The size of both the plaintext- and ciphertext blocks is 128 bits in AES. During encryption and decryption, the algorithm scrambles the data during several rounds of different basic operations. Refer to Section 2.3 for detailed description of the algorithm.

AES is not exactly the Rijndael-algorithm, it comes with a few extra restrictions: While Rijndael allows any key- and and blocksizes that are a multiple of 32 bits and between 128 and 256, the AES has a fixed, 128-bit blocksize and key sizes of 128, 192 and 256 bits [8]. In this report, AES with 128-, 192-, and 256 bit keys will be referred to as AES128, AES192, and AES256, respectively.

2.2 Finite Fields

Operations in AES are performed on basic units of 8 bits, one byte. All bytes are interpreted as elements of the finite field GF(2^8). This ensures that the results of all multiplications and additions also are elements of the same finite field. Hence, a constant word length can be used without overflow problems. All basic operations in AES can be described as operations over the finite field GF(2^8).

To represent a byte, mainly hexadecimal notation will be used in this report. For example, the byte $\{01001010\}$ will be written as $\{4A\}$. Another representation used by the literature is polynomial representation. The byte $b_7b_6b_5b_4b_3b_2b_1b_0$ could be represented as $\sum_{i=0}^{7} b_i x^i$. $\{01001010\}$ would then be written as $x^6 + x^3 + x$.

2.2.1 Addition in GF(2^8)

Addition of two bytes in the finite field is done by performing a bitwise addition modulo 2 on each bit pair in the two bytes that are to be added. This translates to a simple bitwise XOR operation. If $c_7c_6c_5c_4c_3c_2c_1c_0$ is the sum of $a_7a_6a_5a_4a_3a_2a_1a_0$ and $b_7b_6b_5b_4b_3b_2b_1b_0$, then $c_i = a_i \oplus b_i$.[22]

2.2.2 Multiplication in GF(2^8)

Multiplication of two elements in GF(2^8), denoted by $\bullet$, is done by performing a multiplication of the two elements modulo an irreducible polynomial. For AES this irreducible polynomial is defined as $m(x) = x^8 + x^4 + x^3 + x + 1$, or $\{01\}\{1b\}$ in hexadecimal notation.

Multiplication by the polynomial x, or $\{02\}$ in hexadecimal notation, can be done by doing a left shift followed by a conditional subtraction of the irreducible polynomial $m(x)$ [22]. If the most significant bit, MSB, of the element that is to be multiplied with $\{02\}$ is zero, then no subtraction is needed. If the MSB is one, then the subtraction of $m(x)$ should be performed. The subtraction is carried out in the same manner as an addition, a bitwise XOR. This operation is referred to as $xtime()$. Examples:

$$\{7F\} \bullet \{02\} = \{7F\} << 1 = \{FE\}$$
$$\{8F\} \bullet \{02\} = (\{8F\} << 1) \oplus (\{01\}\{1B\}) = \{05\}$$

$\{XX\} << i$ represents $\{XX\}$ shifted i places to the left. The $xtime()$ operation can be used in order to multiply two arbitrary polynomials. In regular binary multiplication, the product can be computed using a sequence of shift and add operations [12]:

						1	0	1	1	1	0	multiplicand ($\{2E\}$)
$\times$						1	0	1	1	0	1	multiplier ($\{2D\}$)
						1	0	1	1	1	0	
					0	0	0	0	0	0		
				1	0	1	1	1	0			
			1	0	1	1	1	0				
		0	0	0	0	0	0					
	1	0	1	1	1	0						
1	0	0	0,	0	0	0	1,	0	1	1	0	product ($\{816\}$)

Multiplications in the finite field can be performed in a similar manner, substituting left shift with $xtime()$ and addition with bitwise XOR.

	$\{2E\}$	multiplicand
$\bullet$	$\{2D\}$	multiplier
	$\{2E\}$	
	$\{00\}$	
	$\{B8\}$	$xtime^2(\{2E\})$
	$\{6B\}$	$xtime^3(\{2E\})$
	$\{00\}$	
	$\{B7\}$	$xtime^5(\{2E\})$
	$\{4A\}$	product

Note that the product is represented by 8 bits in the finite field multiplication. Regular binary multiplication produces a 12-bit product.

2.3 AES algorithm

There are four basic operations used in AES: AddRoundKey , ShiftRows, MixColumns, and SubBytes. The three latter operations also have inverses, called InvShiftRows, InvMixColumns and InvSubBytes. They are repeatedly applied to the block which is to be en- or decrypted. A datablock of 16 bytes is in AES referred to as a *state*. Figure 2.1 gives an overview of how en- and decryption is performed.

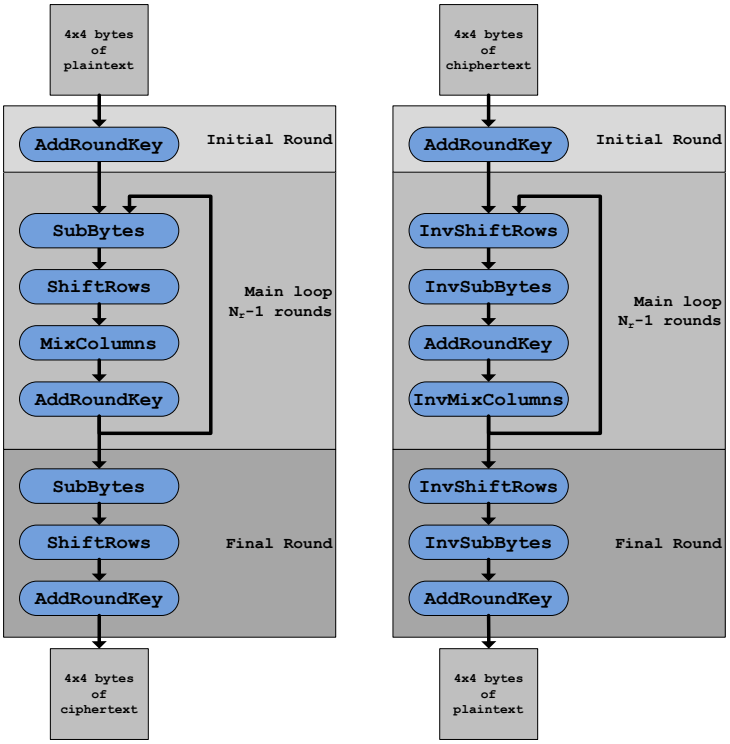

Figure 2.1: AES en- and decryption

2.3.1 AddRoundKey

AddRoundKey is the part of the algorithm which makes the output subject to the cipherkey. Each byte in the state is XOR'ed with a corresponding byte in the expanded key, as illustrated in Figure 2.2. The expanded key is derived from the cipherkey according to the key schedule described in Chapter 2.3.5. AddRoundKey is its own inverse.

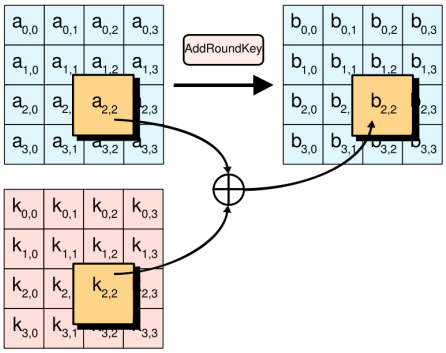

Figure 2.2: AddRoundKey operation, [36]

2.3.2 SubBytes

The SubBytes operation provides the non-linear property of the cipher which is crucial for protection against differential and linear cryptanalysis [9]. It substitutes the state, one byte at a time, using a substitution box known as the Rijndael S-box. Figure 2.3 illustrates how SubBytes is performed.

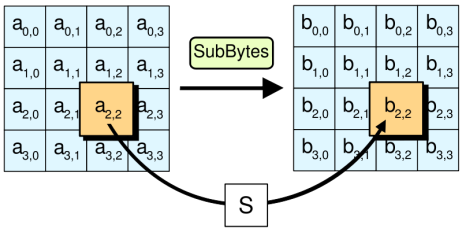

Figure 2.3: SubBytes operation, [36]

The S-box consists of a multiplicative inversion in $GF(2^8)$ in sequence with an invertible affine transformation. The inverse a to an element b is defined such that $a \bullet b = \{01\}$. The element $\{00\}$ is its own inverse. Affine transformation involves the multiplication of a matrix followed by the addition of a vector, as shown in equation 2.1.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{2.1}$$

InvSubBytes is performed like SubBytes substituting the affine transformation with its inverse, given in equation 2.2. The inverse affine transformation has to be performed prior

to the multiplicative inversion.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.2}$$

2.3.3 ShiftRows

ShiftRows is an operation which shifts each row in the state in a cyclical manner. A criteria for this step is that each row should be shifted with different offsets [8]. As the block which AES operates on consists of four rows, the offsets has to be 0, 1, 2 and 3. See figure 2.4 for illustration.

This step introduces inter column diffusion to the algorithm which provides resistance against differential and linear cryptanalysis [8].

InvShiftRows is done by shifting the rows in the opposite direction with the same offset as in ShiftRows.

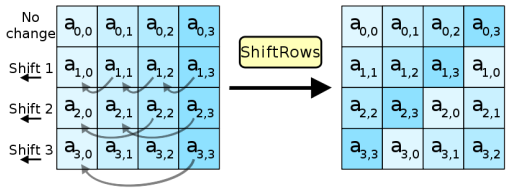

Figure 2.4: ShiftRows operation, [36]

2.3.4 MixColumns

MixColumns performs a transformation of the state, column by column. Each column is interpreted as a polynomial with coefficients in $GF(2^8)$ and multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\}$. As shown in [8], this operation can be written as the matrix multiplication given in Equation 2.3.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} \{02\} & \{03\} & \{01\} & \{01\} \\ \{01\} & \{02\} & \{03\} & \{01\} \\ \{01\} & \{01\} & \{02\} & \{03\} \\ \{03\} & \{01\} & \{01\} & \{02\} \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \tag{2.3}$$

S and S' are the columns before and after transformation, respectively.

InvMixColumns is performed similarly to MixColumns, the only difference being that the inverse of the polynomial in MixColumns is used. The inverse polynomial is $c^{-1}(x) =$

$\{0B\}x^3 + \{0D\}x^2 + \{09\}x + \{0E\}$, resulting in the matrix multiplication given in Equation 2.4.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} \{0E\} & \{0B\} & \{0D\} & \{09\} \\ \{09\} & \{0E\} & \{0B\} & \{0D\} \\ \{0D\} & \{09\} & \{0E\} & \{0B\} \\ \{0B\} & \{0D\} & \{09\} & \{0E\} \end{bmatrix} \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \tag{2.4}$$

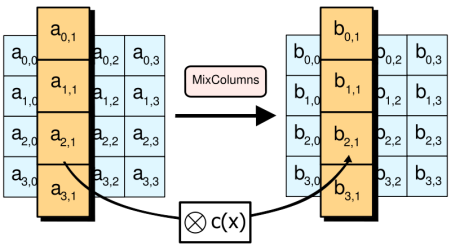

Figure 2.5: MixColumns operation, [36]

MixColumns introduces inter row diffusion to the cipher.

2.3.5 Key expansion

The AddRoundKey operation in AES adds a roundkey in each round of en- or decryption. This roundkey is derived from the cipherkey according to the Rijndael key schedule. The key schedule produces $N_r + 1$ roundkeys (11 for 128 bit key, 15 for 256 bit key), each consisting of 16 bytes. Algorithm 1 describes the key expansion.

Algorithm 1 *KeyExp*(word *CipherKey*[N_k], word *RoundKey*[$N_b * (N_r + 1)$])

```
1: word temp
2: i = 0
3: while i < N_k do
4:    RoundKey[i]=ChipherKey[i]
5:    i++
6: end while
7: i = N_k
8: while i < N_b * (N_r + 1) do
9:    temp = RoundKey[i − 1]
10:   if i mod N_k=0 then
11:      temp = SubWord(RotWord(temp)) ⊕ Rcon[i/N_k]
12:   else if N_k > 6 and i mod N_k=4 then
13:      temp = SubWord(temp)
14:   end if
15:   RoundKey[i]=RoundKey[i − N_k] ⊕ temp
16:   i++
17: end while
```

The keys used in the different rounds of AES are stored in the word-array RoundKey. The function SubWord() applies the SubBytes routine to each byte in the word, wordsize is

4 bytes. RotWord() rotates the word in a cyclical manner, RotWord($[a_0, a_1, a_2, a_3]$) would return $[a_1, a_2, a_3, a_0]$. Rcon[] contains the round constant words, given by $[\{02\}^{i-1}, \{00\}, \{00\}, \{00\}]$, where i is the round number, starting at 1.

2.3.6 Different modes of AES

In block ciphers, two equal plaintext blocks produce the same ciphertext blocks. This is a possible weakness, because the ciphertext might reveal patterns in the plaintext. To counter this effect, different modes of operation can be used. The five operation modes presented in the following sections are recommended by NIST [10].

Electronic Code Book, ECB

ECB is the simplest mode of AES, it simply encrypts the data 128 bits at a time. The advantage with this mode is that it is easy to implement and requires a few less operations than the other modes. The disadvantage is that it reveals patterns in the plaintext, as illustrated in Figure 2.11. Figure 2.6 illustrates how AES in ECB mode is performed.

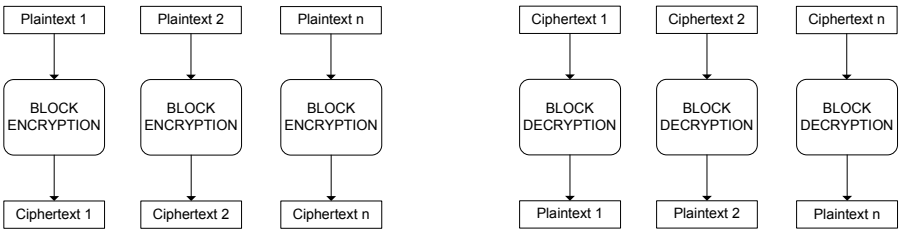

Figure 2.6: En- and decryption in ECB mode

Cipher Block Chaining, CBC

In CBC encryption mode, each plaintext block is XOR'ed with the previous ciphertext block as illustrated in Figure 2.7. In the first round, an initialization vector is used for XOR'ing with the plaintext. In CBC decryption mode, the output from the Block Cipher Decryption needs to be XOR'ed with the previous ciphertext in order to attain the plaintext.

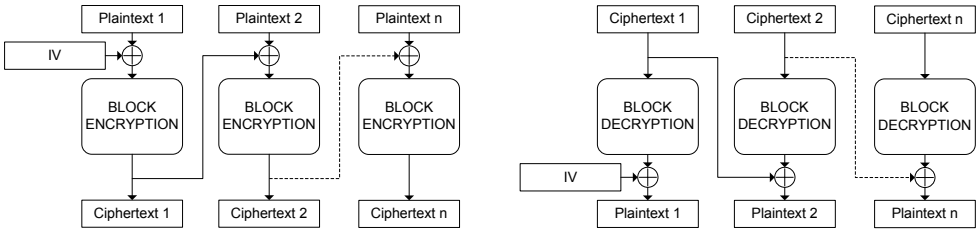

Figure 2.7: En- and decryption in CBC mode

Cipher Feedback, CFB

CFB transforms the AES block cipher into a stream cipher, meaning that the ciphertext is attained by combining the plaintext with a pseudorandom string for example using an XOR operation, which is the case in CFB. XOR'ing the ciphertext with the same string produces the plaintext. Figure 2.8 illustrates how the AES block cipher is used in CFB mode. Notice that block cipher *encryption* is used both in en- and decryption.

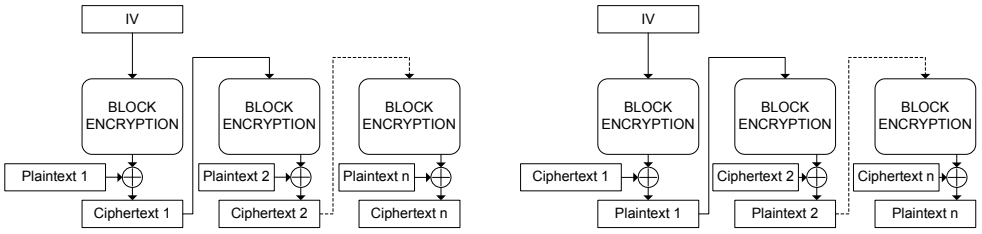

Figure 2.8: En- and decryption in CFB mode

Output Feedback, OFB

Like CFB, OFB transforms AES into stream cipher. The difference compared to CFB is that input to an encryption is the output from the previous block encryption, not the previous ciphertext. Figure 2.9 illustrates how OFB is performed. Also in this mode, only block cipher *encryption* is used.

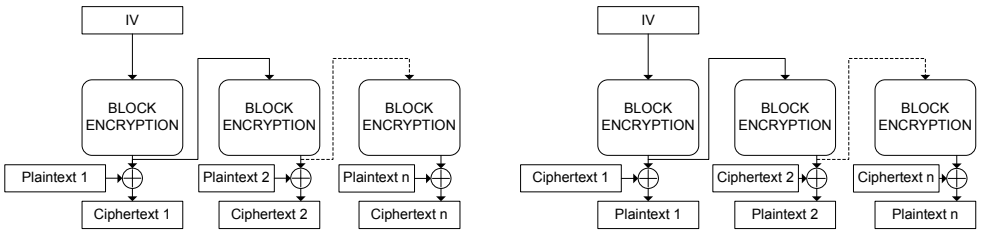

Figure 2.9: En- and decryption in OFB mode

Counter, CTR

Counter mode is another stream cipher using a counter to produce the cipher stream. En- or decryption is done by XOR'ing the plaintext/ciphertext with the cipherstream. Figure 2.10 illustrates AES in CTR mode. As in CFB- and OFB mode, CTR mode only use block cipher *encryption*.

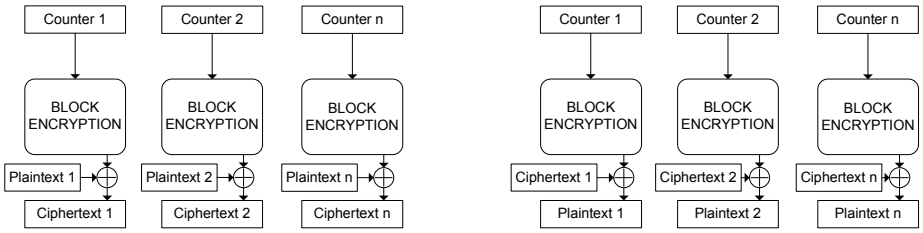

Figure 2.10: En- and decryption in CTR mode

In the modes CBC, CFB and OFB, initialization vectors, IV, are used as input. The initialization vectors are 128 bit vectors which can be computed using different strategies. One strategy recommended by [10] is to apply the forward block cipher to a nonce (a data vector not expected to recur) using the same key as used in encryption. The nonce should be a unique data block for each execution of the encryption process. For more details on generation of IVs and counters in CTR mode, refer to [10].

Figure 2.11 illustrates how the ciphertext can reveal patterns in the plaintext in ECB mode. The other four modes described in the preceding sections ensures that two equal plaintext blocks does not produce the same ciphertext blocks. This prohibits the ciphertext from revealing patterns in the plaintext.

Figure 2.11: Encryption using ECB and other modes, respectively, [37]

2.4 Multiplicative inversion through isomorphic mapping

The SubBytes routine involves multiplicative inversion in $GF(2^8)$. As there are 256 elements in $GF(2^8)$, this can be done using a look up table containing 256 bytes. An alternative is to compute the inverse directly. In order to reduce the complexity of the inversion, the element could be mapped to a finite field of lower order, for example $GF(2^4)$. This would enable the inversion to be done using a 16 byte look up table or a relatively small combinational circuit. [28] describes an approach which uses this strategy. The mapping and its inverse corresponds to matrix multiplications as shown in [28]. The matrices are given in Appendix A. The mapping produces a polynomial of $GF(2^4)^2$ on the form $P_H x + P_L$, where P_H and P_L are elements of $GF(2^4)$. The inverse polynomial modulo $I(x)$, $P_H^{-1}x + P_L^{-1}$ is then given by equation 2.5. The irreducible polynomial $I(x)$ is on the form $I(x) = x^2 + x + \lambda$, where λ is an element in $GF(2^4)$ which can be freely chosen as long as $I(x)$ remains irreducible.

$$1 = (P_H x + P_L)(P_H^{-1}x + P_L^{-1}) \, mod \, I(x) \tag{2.5}$$

As shown in [8], P_H^{-1} and P_L^{-1} can then be computed using Equation 2.6 and 2.7, respectively.

$$P_H^{-1} = P_H(\lambda P_H^2 \oplus P_H P_L \oplus P_L^2)^{-1} \tag{2.6}$$

$$P_L^{-1} = (P_H \oplus P_L)(\lambda P_H^2 \oplus P_H P_L \oplus P_L^2)^{-1} \tag{2.7}$$

Figure 2.12 depicts an architecture performing Equations 2.6 and 2.7.

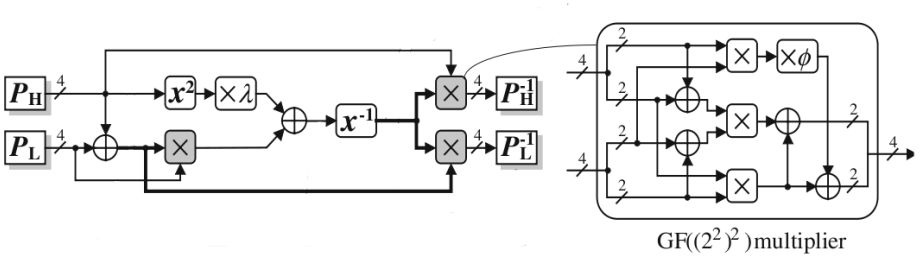

Figure 2.12: Inversion in $GF(2^4)$, [28]

Multiplication in $GF(2^4)$ could be performed as depicted to the right in Figure 2.12. As shown in [19], the $GF(2^2)$ multiplications performed inside the $GF(2^4)$ multiplier can be performed using a small number of AND and XOR gates.

2.5 Cryptanalysis

Cryptanalysis is the study of deriving information from an encrypted message without knowing the key. Different approaches can be made in order to break a cipher, and they can be divided into two main groups: algorithm based attacks, and implementation attacks which is also called side channel attacks. Cryptanalysis is a very wide field, and only a few examples will be presented in this section.

Side channel attacks

Side channel attacks is a collective term for all types attacks where information is gained from the physical implementation of a cryptographic system.

Differential power analysis is a side channel attack which exploits the fact that power consumption might vary with the data being processed. By measuring the power consumption during encryption, information about the data or key can be collected, and used in order to break the cipher. Especially, if there exist a relationship between the key and power consumption, this type of attack could be efficient. The key expansion in AES ensures that even if the cipherkey is all ones or zeros, the roundkeys will be of such a nature that the power consumption would not reveal any information about the key. This makes it hard to break the AES cipher using power analysis. The physical implementation of an AES module also has impact on how susceptible it is to this kind of attacks. [13] suggest to use masking in order to prevent direct operations between the key and data. This however would add complexity to the hardware which in turn leads to increased energy consumption. As only a small part of the circuitry is targeted in a power analysis attack, all power consumption not correlated to the targeted part appears as noise to the attacker. Based on this, one can

conclude that implementations utilizing large datapaths would be better protected from this kind of attack. [13]

Timing attack is another side channel attack which can be used if processing time during encryption depends on the data or key. In AES, the processing time of all operations are inherently independent of both key and data, making AES well protected against timing attacks.

Algorithm based attacks

Linear cryptanalysis is a general form of cryptanalysis based on finding affine relationships between ciphertext and plaintext. If the cipher is not properly constructed to withstand this kind of attack, these relations can lead to information regarding the key. Another general form of cryptanalysis is differential cryptanalysis. The basic idea in differential cryptanalysis is to study differences in the output based on differences in the input of a cipher. The non linear properties of the Sbox is the main contributer to resistance against linear and differential cryptanalysis. The Sbox used in AES have extremely low correlation between in- and output. In addition, when applying an input difference to the Sbox one can derive little or no information about the output difference. These properties ensures that attacks based on linear or differential cryptanalysis will not succeed against AES [9]. For more details on algorithm based attacks, refer to [9].

AES has become a world standard and is expected to remain a standard for 30 years. Although numerous attempts for breaking the cipher has been made, no results implies that the security of AES should be questioned [9].

Chapter 3

Power Consumption

3.1 Sources of power consumption

The power consumption in CMOS technology can be split into three main components; dynamic, leakage and short circuit. [6]

$$P_{total} = P_{dynamic} + P_{leakage} + P_{short-circuit} \tag{3.1}$$

3.1.1 Dynamic power

In a CMOS circuit, each node is associated with a capacitance, C. During operation, the nodes switch values from logical zeros to logical ones a number of times. Each time, an amount of power is dissipated in the process of charging the capacitance at the given node. The dynamic power component is given in Equation 3.2, where V_{dd} is the supply voltage, f is the clock frequency and α is the number of transitions per clock cycle.

$$P_{dynamic} \propto CV_{dd}^2 f\alpha \tag{3.2}$$

3.1.2 Short circuit power

In CMOS, when the output of a logic gate changes value, there will be a short period of time when both the N- and P-network are partially conducting. This results in short circuit power dissipation due to the current flowing from V_{dd} directly to ground. Both N- and P-networks are (partially) on when $V_{tn} < V_{in} < (V_{dd} - |V_{tp}|)$ [6]. [24] states that with careful design, this power consumption source can be kept to be less than 15 % of the dynamic power.

3.1.3 Leakage

Leakage power is power consumption due to the leakage currents flowing though transistors which are not supposed to conduct. The leakage current can be split into three main components [24]:

- Source/drain junction leakage current
- Gate direct tunneling leakage

- Sub-threshold leakage through channel

The source/drain leakage current flows from the source or drain to the substrate. Gate direct tunneling leakage is the current flowing through the gate of the transistor to the substrate. Sub-threshold leakage current flows through the channel of a transistor that is not supposed to conduct, this current is given by Equation 3.3, where K and n are functions of the technology used and η is the drain-induced barrier lowering coefficient [24].

$$I_{DS} = K(1 - e^{-\frac{V_{DS}}{V_T}})e^{\frac{(V_{GS}-V_T+\eta V_{DS})}{nV_T}} \tag{3.3}$$

Figure 3.1 depicts the contribution of leakage power compared to dynamic power. As the transistor sizes decrease, the power dissipation due to leakage increase exponentially. Figure 3.2 illustrates how the leakage currents vary with temperature. The leakage currents grow exponentially with the temperature. It should also be noted that leakage power is given as W/cm^2, hence it is proportional to the area.

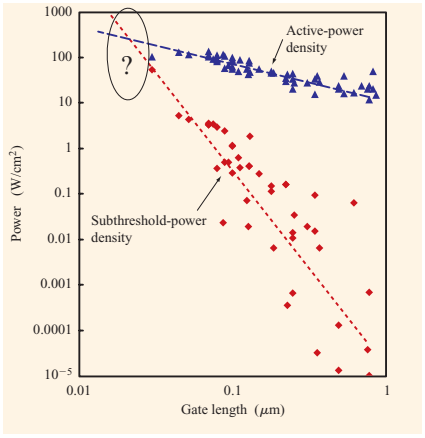

Figure 3.1: Leakage vs Dynamic power, [21]

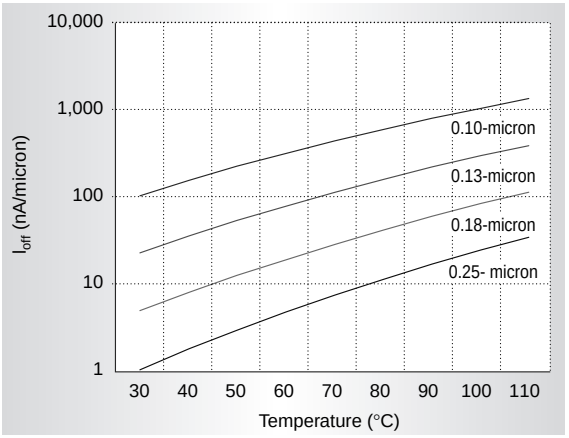

Figure 3.2: Leakage vs Temperature, [3]

3.2 Software power consumption

As software executes on hardware, the basic mechanisms for power consumption mentioned in Section 3.1 also apply here. The drawback, in terms of power consumption, associated with software is that the microprocessor executing the code often has to use several instructions in order implement functionality that could be done easily using dedicated hardware. Each instruction consume power and in addition, several power consuming memory accesses has to be performed in order to implement the desired functionality. As execution of software involves more switching of nodes than the same functionality implemented in hardware, a hardware solution would generally result in lower energy consumption.

During software execution, memory accesses represents a significant part of the energy consumption. [7] estimates that data- and instructions supply consumes 70% of the total energy.

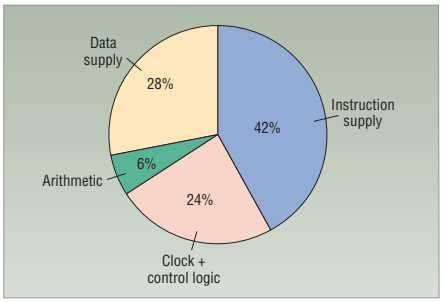

Figure 3.3: Energy consumption in embedded processors, [7]

As seen in Figure 3.3, only 6% of the total energy is dissipated performing arithmetics. Only 59% of these 6% (3.54%) are spent on *useful* arithmetics. The remainder is spent on overhead, like updating loop indices and calculation of memory addresses. The amount of energy consumed in useful arithmetics is comparable to what a hardwired module would consume, as similar arithmetic units are used [7]. Energy consumption due to memory accesses and overhead generally makes software implementations significantly less energy friendly than a hardware implementation.

3.3 Power reduction techniques

3.3.1 Voltage scaling

Equation 3.2 states that the dynamic power consumption is proportional to V_{dd}^2. Thus, lowering the supply voltage could potentially lead to a significant reduction of power consumption. The drawback with lowering the supply voltage is that the logic gates become slower, i.e., the delays through them increase. If the increased delay represents an unacceptable degradation of the total design, different countermeasures could be used.

One approach is to lower the threshold voltage of the transistors [6]. Unfortunately, lowering the threshold voltage increases the leakage current. As seen in Equation 3.3, there is an exponential relationship between the channel leakage current and the threshold voltage.

Another possible countermeasure to compensate for the increased delay due to voltage scaling is pipelining. By inserting one or more pipeline stages, the critical path through the combinational logics can be significantly reduced. This allows the voltage to be reduced while the throughput is kept constant. The drawbacks associated with pipelining are increased area and additional power consumption in the pipeline registers. In addition, a latency of N cycles is introduced when a pipeline consisting of N stages is used. If the reduction in power consumption due to voltage scaling outweighs the drawbacks of pipelining, this method can be used to make a module more energy friendly [6].

3.3.2 Clock and data gating

Clock distribution represents a major part of the power consumption in a chip. As much as 50% or even more of the dynamic power can be spent on supplying the sequential parts of the design with clock signals [15]. The clock distribution circuitry represents such a large part of the power consumption due to high switching activity and the high drive strength

of the clock buffers, which is necessary in order to minimize clock delay. A widely used approach to reduce the power dissipated in clock distribution is clock gating. By using clock gates, the clock signals to parts of the design can be shut down resulting in reduced power consumption in clock distribution. There are two types of clock gating, combinational and sequential.

Combinational clock gating involves disabling the clock for registers that do not change state. This could also be implemented using a feedback mux, but using clock gating is favorable as it saves both area (no feedback mux is needed) and power in the clock tree. The logic functionality is exactly the same, making is easy to verify equivalence. This type of clock gating reduces activity in the clock tree, but not in the fan out of the registers. A combinational clock gate is depicted in Figure 3.4.

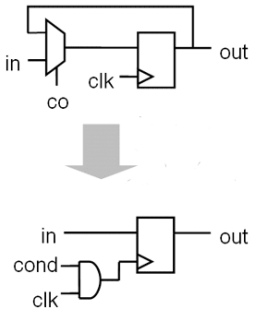

Figure 3.4: Combinational clock gate

Sequential clock gating involves locating states when the output of registers change even though they do not need to. Gating the clock to these registers would eliminate unnecessary switching in the fan out of the register and thus saving energy. This type of clock gating is more efficient than combinational clock gating as it limits switching in the circuitry following the registers in addition to saving energy in the clock tree. Since the logic functionality is changed, verifying equivalence using this type of clock gating is harder than what is the case in combinational clock gating. Figure 3.5 depicts a sequential clock gate.

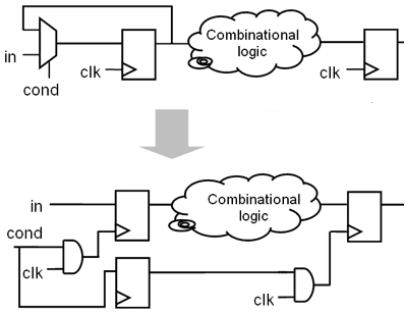

Figure 3.5: Sequential clock gate

Data-gating is a technique based on the same principle as sequential clock gating; keeping the inputs to a logic block constant will prevent the gates in the block from consuming

dynamic power. A simple *and* operation between the input(s) of a logic block and an enable signal would prevent undesired switching in the circuitry.

As both clock- and input-gates consume area and power, an evaluation has to be made whether usage of these would lead to a better solution. If a gate is open most of the time, it would probably lead to a higher average power consumption in addition to the increased area. On the other hand, if the gate is closed most of the time, the reduction in power consumption can be significant. When using clock gating, the number of flip flops gated need to be large enough to outweigh the power consumed in the gates.

3.3.3 Power gating

An approach to reduce energy consumption due to leakage is to turn off the power supply for modules which are not in use. This can be done by using an NMOS in series with the logic gates, as depicted in Figure 3.6.

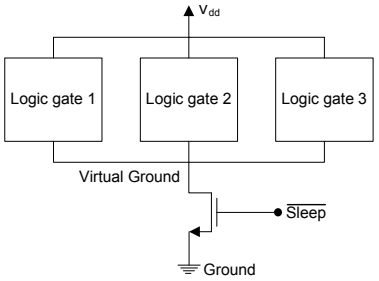

Figure 3.6: Power gate

Asserting the *Sleep* signal disconnects the logic gates from the ground, minimizing the leakage current. Power gate transistors should incorporate high threshold voltages, V_t, to keep the leakage current through the power gate itself at a minimum. As seen in Equation 3.3, increasing the threshold voltage greatly reduces leakage currents. In order for the logic gates connected to the power gate to function properly, the sleep transistor has to be carefully sized. If the voltage drop over the sleep transistor is too large, the delays through the logic gates will increase. Using a large sleep transistor solves this, but increases area overhead and the dynamic power consumed for turning the transistor on and off [24]. For more details on sizing of power gate transistors, refer to [24]. Due to the fact that turning the power gate on and off consumes energy, there exists a lower time limit indicating how long the *Sleep* signal must be asserted in order to save energy. Using the power gate for shorter periods than this time limit only results in increased energy consumption. It should be noted that using power gates on sequential circuitry, like registers, results in loss of the data stored.

3.3.4 Numerical strength reduction

Constant matrix multiplication is involved in quite a lot of algorithms. [25] presents a numerical transformation technique which reduces the strength of these matrix multiplications. This technique is based on subexpression elimination. The idea in subexpression elimination is to analyze the computation that is to be done, extract subexpressions involved in multiple computations, and share these.

Example:

Equation 3.4 shows an example of a constant matrix multiplication.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \{05\} & \{00\} & \{04\} & \{00\} \\ \{00\} & \{05\} & \{00\} & \{04\} \\ \{04\} & \{00\} & \{05\} & \{00\} \\ \{00\} & \{04\} & \{00\} & \{05\} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{3.4}$$

Straightforward calculation could be performed as in Equations 3.5 through 3.8.

$$y_0 = \{05\} \times x_0 + \{04\} \times x_2 \tag{3.5}$$
$$y_1 = \{05\} \times x_1 + \{04\} \times x_3 \tag{3.6}$$
$$y_2 = \{04\} \times x_0 + \{05\} \times x_2 \tag{3.7}$$
$$y_3 = \{04\} \times x_1 + \{05\} \times x_3 \tag{3.8}$$

These calculations would require eight multiplications and four additions. Applying numerical strength reduction reduces the number of multiplications needed. The procedure can be divided into four steps.

1. Represent the coefficients in each column in binary form
2. Perform iterative matching on the coefficients to derive common subexpressions
3. Write each y_i as a sum of subexpressions
4. Perform iterative matching on the expressions for y_i to find common subexpressions

Step 1 and 2:

The coefficients in each column can be represented with a set of subexpressions. In this case, all columns consist of the coefficients $\{04\}$ and $\{05\}$. These coefficients can be represented by the subexpressions $\{04\}$ and $\{01\}$. $\{05\}$ would then be written as $\{04\} + \{01\}$. In the case of more complex coefficients, the subexpressions can be found using iterative matching, described in step 4.

Column 0	Column 1	Column 2	Column 3
$\{04\}$	$\{04\}$	$\{04\}$	$\{04\}$
$\{01\}$	$\{01\}$	$\{01\}$	$\{01\}$

Step 3:

The subexpressions represents unique products which can be used to calculate each y_i.

$$p_1 = \{04\} \times x_0, \quad p_2 = \{01\} \times x_0, \quad p_3 = \{04\} \times x_1, \quad p_4 = \{01\} \times x_1,$$
$$p_5 = \{04\} \times x_2, \quad p_6 = \{01\} \times x_2, \quad p_7 = \{04\} \times x_3, \quad p_8 = \{01\} \times x_3$$

$$y_0 = p_1 + p_2 + p_5$$
$$y_1 = p_3 + p_4 + p_7$$
$$y_2 = p_1 + p_5 + p_6$$
$$y_3 = p_3 + p_7 + p_8$$

Step 4:

Iterative matching can now be performed in order locate common subexpressions. Iterative matching can be divided into four steps:

1. Represent the expressions for each y_i in a binary format.
2. Determine the number of bitwise matches between the expressions, choose the best match.
3. Create a new expression consisting of the shared subexpressions found in step 2. Return remainders of the y_i's and the new expression to the expression set.
4. Repeat steps 2 and 3 until no further improvements are made.

p_i	1	2	3	4	5	6	7	8
y_0	1	1	0	0	1	0	0	0
y_1	0	0	1	1	0	0	1	0
y_2	1	0	0	0	1	1	0	0
y_3	0	0	1	0	0	0	1	1

Examining the table above reveals that y_0 and y_2 share the subexpressions p_1 and p_5. y_1 and y_3 share the subexpressions p_3 and p_7. A new table is made with two new expressions, C_{02} and C_{13}. These represent the common expressions for y_0, y_2 and y_1, y_3, respectively. ry_i represents the remainder of y_i, that is what needs to be added to the common expressions in order to form y_i.

p_i	1	2	3	4	5	6	7	8
C_{02}	1	0	0	0	1	0	0	0
C_{13}	0	0	1	0	0	0	1	0
ry_0	0	1	0	0	0	0	0	0
ry_1	0	0	0	1	0	0	0	0
ry_2	0	0	0	0	0	1	0	0
ry_3	0	0	0	0	0	0	0	1

No further improvements can be made and the new expressions for y_i can be written as seen in Equations 3.9 through 3.14

$$C_{02} = \{04\} \times x_0 + \{04\} \times x_2 = \{04\} \times (x_0 + x_2) \tag{3.9}$$
$$C_{13} = \{04\} \times x_1 + \{04\} \times x_3 = \{04\} \times (x_1 + x_3) \tag{3.10}$$
$$y_0 = C_{02} + x_0 \tag{3.11}$$
$$y_1 = C_{13} + x_1 \tag{3.12}$$
$$y_2 = C_{02} + x_2 \tag{3.13}$$
$$y_3 = C_{13} + x_3 \tag{3.14}$$

The common subexpressions, C_{02} and C_{13}, are shared and only need to be computed once. The new equations require two multiplications and six additions. Also, only multiplications

by {04} needs to be performed which is less complex than multiplication by {05} needed in the original computation of y. Numerical strength reduction results in less complex hardware able to do the computations using less energy.

3.3.5 Energy versus power

Energy- and power consumption are obviously closely related, the relationship being that power consumption over time results in energy consumption, as shown in equation 3.15. As a battery is able to store a certain amount of energy, the life of the battery might be prolonged if a module is designed to consume a large amount of power for a short period of time instead of a small amount of power for a long period of time.

$$\text{Energy} = \int_T \text{Power(t)} \ \mathrm{d}t \tag{3.15}$$

Datapath width is a design parameter which has great impact on both power consumption and execution time. Increasing the datapath width generally increases the power consumption, but could also lead to a reduction in execution time.
"In general, if a datapath is too narrow, energy is increased because of the increased instruction cycles." [24]

Expanding the datapath of a hardware module could lead to lower energy consumption if the computations that is to be performed are inherently associated with a bitwidth. For instance, addition of two 32 bit integers would be far more energy efficient using a 32-bit datapath opposed to a 8-bit datapath. Using a datapath narrower than this bitwidth would lead to additional execution cycles and control circuitry. Using an even broader datapath enables parallelization of computations. If the calculations to be performed are possible to execute in parallel, a significant speedup and possible energy reduction is achievable.

[4] explores how the width of the datapath affects the energy consumption. A program performing MPEG-2 decoding was evaluated on a soft core processor using various datapath widths. The energy consumption in the CPU grew in a non-monotonic manner as the width of the datapath was increased. Depending on the task that is to be performed, the datapath width of a computational unit should be optimized in order to minimize energy consumption.

3.4 Power estimation

The ability to estimate the power consumption of a module is essential for a designer in order to evaluate the quality of a design. Power estimation is done by combining parameters like supply voltage and operating frequency with a description of the design and data regarding activity in the circuitry. The different approaches differ in the models being used and how the activity data is collected. In general, the approaches which produces the most accurate estimations require significantly more time and effort than the less accurate ones [24].

3.4.1 Design models

When estimating power consumption in a design, a model of the design has to be provided to the simulation and estimation tool. The detail level of this model influences the accuracy

of the estimation and the execution time of the simulations. More detailed models generally produce more accurate results, but increases the simulation time.

RTL level models describes the design as a collection of memory elements and combinational black boxes [20]. At this abstraction level the composition of logic gates are not known to the simulator making it impossible to calculate exact activity in the combinational parts of the design. As a result of this, accuracy is degraded. For instance, glitches due to delays in the combinational logics are not accounted for.

Gate level models provide information about all logical gates in the design, usually by means of a netlist. This makes it possible to attain more precise information about the activity in the circuitry. If information about delays through logical gates is provided, the energy dissipation due to glitches can be estimated, which can represent a significant part of the energy consumption, typically about 20% [29].

For even more accurate power estimation, transistor level or post layout models can be used leading to increased accuracy and simulation time.

3.4.2 Estimating switching activity

Simulation based estimation is a straightforward way to estimate power consumption. During simulation, switching activity in different parts of the design is logged. Combining this with information like power supply and capacitance at different nodes in the design, allows average power to be computed. The circuit on which simulations are performed can be represented using models of different detail levels, for instance register transfer level, RTL, or gate level. Simulation on RTL models are quite fast, at the expense of accuracy. In order to include energy consumption due to glitches, gate level simulation has to be performed.

Probability based estimation is an alternative to simulation. This approach uses probabilities to describe switching in the circuitry. When performing power estimation using Synopsys' Power Compiler, the designer has the possibility to define switching activity in different parts of the design using probabilities and toggle rates. The designer can, for instance, specify that an input to a design is a logical 1 50% of the time and that it toggles 10 times in 1000 time units. Power compiler can then use this data to estimate the power consumption of the design. This is done by propagating the switching activity defined by the designer using a zero delay simulator [34]. In addition to short execution time, this approach has the advantage that the exact stimuli does not have to be known, which is often the case when power estimation is to be performed [20]. Other approaches for computing switching activity without simulation are presented in [20].

3.4.3 Software power estimation

Estimation of power consumption during software execution could be performed in the same manner as hardware. However, this requires a detailed model of the processor and simulations would be relatively time consuming making it impractical or impossible [35].

[35] proposes an estimation method for software power consumption based on an instruction level power model. Each instruction is analyzed with regards to power consumption and cycle execution time. By investigation of the assembly code, an estimate of the power

consumption can be calculated. [26] simplifies this approach by using the average power for all instructions. Physical measurements shows that this method is sufficient and accurate within 8% with 99% confidence [26].

As mentioned in Section 3.2, memory accesses represent a major part of the total energy consumption when software is executed. Estimation of this contribution to the total energy can be performed by simply counting the number of memory accesses made. Combining this with a constant representing energy dissipated per memory access provides an estimate of energy consumption due to memory accesses. Energy dissipated in a memory access is determined by memory size, presence of cache, and what kind of memory that is being used, among others. For details on how energy consumption per memory access can be calculated, refer to [17].

Chapter 4

Microcontrollers

Microcontroller theory is a wide field and numerous aspects of microcontroller design could be presented. To limit the size of this chapter, only topics which are relevant for discussion in this thesis will be presented.

4.1 Architecture

Modern microcontrollers are miniaturized, single chip computers incorporating many of the same basic blocks as a regular computer, for instance memory (both volatile and non-volatile), Central Processing Unit (CPU), and bus system. In addition, microcontrollers are equipped with a set of peripherals aiding the CPU and enabling communication with the outside world. Microcontrollers can be divided into two architectural classes, Von Neumann and Harvard [33].

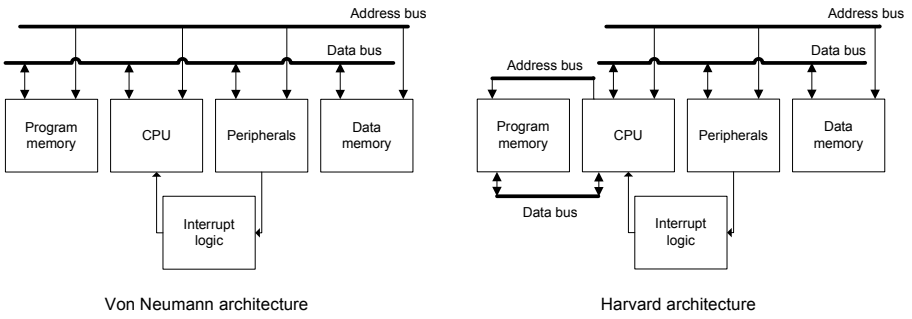

Figure 4.1: Von Neumann vs Harvard architecture, [33]

Figure 4.1 shows the two fundamental architectures. The difference between the two is that Harvard architectures have separate buses for program- and data memory. The separate buses enables the next instruction to be fetched while the previous instruction is being executed, resulting in significantly increased computer speed. Computers based on Harvard architectures often have reduced instruction sets, leading to a less complex and faster CPUs. Because of this, Harvard architectures are often referred to as Reduced Instruction Set Computers (RISC). Von Neumann architectures often come with rather complex instruction sets and are therefore often called Complex Instruction Set Computers

(CISC). In order to execute these complex instructions, CISC CPUs are relatively complex and slow, compared to RISC computers [33].

4.2 Peripherals

A microcontroller is equipped with a set of resources, called peripherals. These resources are hardware modules specialized for some specific task. Almost all microcontrollers incorporate the following peripherals [33]:

- General Purpose I/O ports (GPIO)
- Asynchronous serial interface (UART)
- Synchronous serial interface (SPI)
- Several types of timers
- Analog to digital converters (ADC)

For details regarding the peripherals mentioned above, refer to [33].

In addition to the mentioned peripherals, many microcontrollers include peripherals specialized for some sort of computations. These peripherals can relieve the CPU from certain types of computations. This enables the CPU to enter a low power mode or perform other tasks in parallel. Cryptographic algorithms are examples of computation intensive tasks which could be implemented in a specialized peripheral module.

4.3 Memory map

A CPU is able to address a certain amount of memory locations ($2^{32} = 4G$ for 32-bit architectures). Most CPUs utilizes memory mapped I/O, meaning that the CPU makes no distinction between memory devices and peripherals like a UART or an AES module [5]. All resources (like peripherals and memory devices) available to the CPU are represented by addresses within this address space. In order to access a peripheral, the CPU simply reads or writes to an address assigned to the specific peripheral. Figure 4.2 shows an example of a memory map, taken from from the ARM Cortex M3 processor.

4.4 Direct Memory Access

Some microcontrollers include a Direct Memory Access Controller, DMAC. A DMAC is a unit which can be used for data transfers without invoking the CPU. It can be programmed by the CPU to transfer an amount of data from one memory location to another upon some sort of request, either from the CPU itself or from a peripheral module. When the data transfer is completed, the DMAC issues an interrupt signaling the CPU that the task is performed. When DMA is used, the CPU is only involved at the beginning and end of a transfer [30]. As use of DMA relieves the CPU, the CPU can reside in a low-power mode or alternatively perform other tasks while the transfer is being performed. A DMAC can also enhance the throughput of a peripheral module as a data transfer can be done without involving the CPU [23]. A DMAC is typically equipped with a number of channels which can be configured independently to perform memory transfers.

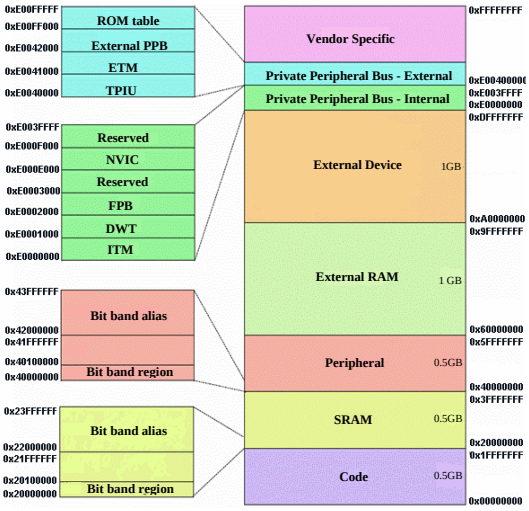

Figure 4.2: ARM Cortex M3 memory map, [27]

4.5 Microcontrollers and power

Microcontrollers are incorporated in most modern electronic products [23], and as prolonged operation time in battery operated devices is highly desired, low-power design has become increasingly important in microcontrollers. Another aspect making low-power design increasingly important is the increasing transistor counts in todays microcontrollers. High power consumption in a small chip, like a microcontroller, would require a heat sink. Designs utilizing microcontrollers often have strict requirements regarding mechanical devices, such as heat sinks, and it is therefore desirable for a microcontroller to be able to function without such a device. As seen in Equation 3.2, the power consumption in CMOS devices is proportional to the frequency, hence the power consumption can be a limiting factor to performance [23]. As a result of this, low-power design is a necessity in microcontrollers.

Low-power modes

One of the most common measures taken in order to limit power consumption in microcontrollers is low-power modes. In many applications, the demands for performance varies over time. Low-power modes enables the device to adapt its power consumption according to the performance demands. Actions taken to lower power consumption could for instance be disabling clock signals or power supply (by means of power gating) to parts of the design. For instance, the ARM Cortex M3 based MCU STM32F10106 from STMicroelectronics incorporates three low power modes [31]:

1. *Sleep mode* stops the CPU. All peripherals are running an can wake up the CPU by issuing an interrupt.

2. *Stop mode* stops the CPU and all peripherals while retaining the contents of the RAM and registers.

3. *Standby mode* switches off the voltage regulator, stopping the CPU and peripherals. RAM and register contents are lost in this mode.

The general trend associated with low power modes is that lower power consumption is achieved at the cost of reduced functionality. This is also the case for the above mentioned example. In *Sleep mode*, all peripherals are active enabling the MCU to perform tasks which does not involve the CPU. In the two other low-power modes, no functionality is available.

4.6 Introduction to ARM Cortex M3

The Cortex M3 processor from ARM has been specified for comparison with the hardware solution in this thesis. This section will give a brief introduction to the ARM Cortex M3.

The Cortex M3 is 32-bit processor based on a Harvard architecture. It is designed to deliver high performance while maintaining low cost and power consumption. The core, which occupies an area of approximately 33000 gates, incorporates a 3-stage pipeline, consisting of Instruction fetch, instruction decode, and instruction execute. Hardware support for division and single cycle multiplication is included in the Arithmetic Logic Unit, ALU. These features, among others, results in a performance of 1.25 $\frac{DMIPS}{MHz}$ when evaluated using the Dhrystone benchmark. The Thumb-2 instruction set architecture, which is a blend of 16- and 32-bit instructions, delivers the performance of 32-bit ARM instructions and matches the code density of the 16-bit Thumb instruction set. For more details regarding the ARM Cortex M3, refer to [27]. Figure 4.3 shows an overview of the ARM Cortex M3.

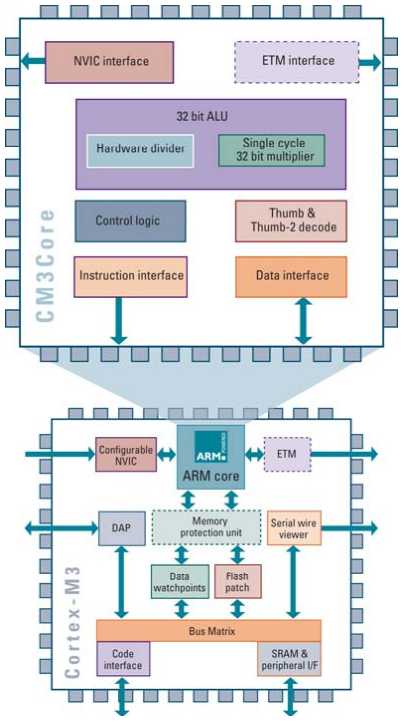

Figure 4.3: ARM Cortex M3, [27]

Chapter 5

Software Solution

An alternative to implementing AES in a dedicated hardware module is to implement it in software. Advantages with this approach are reduced design time and saved area as no additional hardware is required. Although no extra hardware is required, the code itself requires memory space which might be considered an additional cost. Another advantage with implementing the algorithm in software is flexibility. Software can be changed after production and corrected if any bugs should occur, this is not possible in hardware implementations. The drawback with implementing AES in software is that microprocessors are not particularly suited to perform the operations needed in the algorithm and therefore leading to decreased speed and increased energy dissipation.

5.1 Software implementation

As part of this thesis, a software version of AES was implemented based on the technique described in [2]. The main idea is to transpose the state matrix allowing the MixColumns operation to be parallelized. In order to produce the correct result, the roundkeys also needs to be transposed. [2] describes how the transposed keys can be computed directly. Alternatively, the roundkeys can be calculated as described in Section 2.3.5 and then transposed. The key expansion implemented in this thesis computes the transposed roundkeys directly and takes 939 cycles in AES128 and 1204 cycles in AES256. Transposing the state- and key matrix allows the AES processing to be executed fast and with relatively modest demands to code size as only two 256 byte look up tables are needed.

[1] presents another implementation utilizing ten 256 byte look up tables avoiding hardware computation of MixColumns and its inverse. This decreases execution time, but increases the code size dramatically. The version based on transposing the state was chosen for implementation as it combines fast execution and relatively low code size.

SubBytes and ShiftRows

The SubBytes operation substitutes each byte in the state, as described in Section 2.3.2. ShiftRows is implemented by proper selection of the bytes chosen for substitution. The substitution is implemented using two 256 byte look up tables, storing the values for SubBytes and its inverse. Since this implementation operates on a transposed version of the state, ShiftRows has to be performed on the columns, not the rows. SubBytes is unchanged as it

is a bytewise operation. Simulations show that substituting and shifting a whole state takes 68 cycles which averages to 4.25 cycles per byte.

MixColumns

Transposing the matrix enables the multiplications in MixColumns to be executed in parallel, dramatically reducing execution time. Equations 5.1 through 5.4 shows how MixColumns is performed on the transposed state. x_i denotes column i in the transposed state before MixColumns is applied, y_i denotes column i after transformation.

$$y_0 = \{02\} * x_0 \oplus \{03\} * x_1 \oplus x_2 \oplus x_3 \tag{5.1}$$
$$y_1 = x_0 \oplus \{02\} * x_1 \oplus \{03\} * x_2 \oplus x_3 \tag{5.2}$$
$$y_2 = x_0 \oplus x_1 \oplus \{02\} * x_2 \oplus \{03\} * x_3 \tag{5.3}$$
$$y_3 = \{03\} * x_0 \oplus x_1 \oplus x_2 \oplus \{02\} * x_3 \tag{5.4}$$

The difference compared to the original MixColumns is that the multiplications are performed on entire words, not bytes. The symbol $*$ denotes a set of four ordinary multiplications over the field $GF(2^8)$ performed on each byte of the word in parallel. Inverse MixColumns is performed in the same manner, substituting the multiplication coefficients to form the inverse of MixColumns.

	Straightforward	NSR
MixColumns	334	52
MixColumns^{-1}	667	86

Table 5.1: MixColumns and NSR

Numerical strength reduction, described in Section 3.3.4, was applied in order to further reduce execution time. The simplified equations shown in Appendix C were used to reduce the complexity of the computations needed. As seen in Table 5.1, applying numerical strength reduction reduces the cycle count significantly opposed to a straightforward implementation. The code for the different implementations can be viewed in Appendix D. Simulations show that MixColumns transformation on the entire state takes 52 cycles, resulting in 13 cycles per column. Inverse MixColumns uses higher order multiplications and therefore consumes more time: 86 cycles per state, averaging to 21.5 cycles per column.

5.2 Evaluation on ARM Cortex M3

As discussed in Section 3.4.3, there exists different approaches for estimation for energy consumption during software execution. As the available tools only could provide cycle counts, the energy estimation was performed using this information in combination with the average power consumption of the ARM Cortex M3, provided by [27]. This approach should be sufficiently accurate as [26] concludes that such an approach provides an accuracy within 8% with 99% confidence.

The software implementation was evaluated using IAR Embedded Workbench 5.4, which provided cycle counts and code size. STMicroelectronics' MCU STM32F101C6 was used during the simulations. STM32F101C6 was chosen for simulation as it is an ARM Cortex

M3 based MCU [31]. Two 256 and one 10 byte (containing the roundconstants) look up tables were used, resulting in 524 bytes of read only data (RO data). RAM usage varies with the key size, 192 bytes for AES128 and 256 bytes for AES256. The state needs 16 bytes and the roundkeys need 176 and 240 bytes in AES128 and AES256, respectively. The evaluation results are summarized in Table 5.2. The cycle count and energy figures are based on en- or decryption of a 128 bit data block.

	Cycles	Code size	RO data	RAM footprint	nJ/datablock
AES128 encryption	1388	1374	524	192	333
AES128 decryption	1697	1374	524	192	407
AES256 encryption	1956	1374	524	256	469
AES256 decryption	2401	1374	524	256	576
Straightforward implementation, from [38]					
AES128 encryption	3509	972	524	192	842
AES128 decryption	5014	972	524	192	1203

Table 5.2: Performance and cost for AES software

The lower part of Table 5.2 summarizes key figures for the straightforward software implementation of AES used in [38]. As can be observed, the cycle counts are significantly larger. This is due to the fact that the implementation used in [38] was not optimized for 32 bit architectures, performing MixColumns bytewise instead of on whole words simultaneously.

The ARM Cortex M3 uses $0.24 \times 10^{-6} \frac{mW}{Hz}$ when synthesized using the 180nm ARM SAGE-X standard cell library [27]. Combining this with cycle counts for the different modes provides the amount of energy needed for en- or decryption. Equation 5.5 was used to calculate the energy figures. E is the energy consumed, P is power and f is the frequency.

$$E = \#cycles \times \frac{1}{f} \times P = \#cycles \times \frac{1}{f} \times 0.24 \times 10^{-6} \times f = \#cycles \times 0.24 \times 10^{-6} \quad (5.5)$$

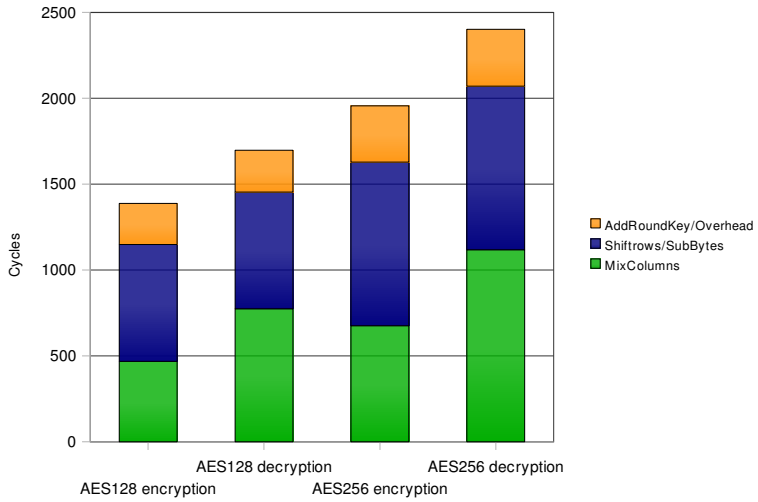

Figure 5.1: Cycle counts for software AES on ARM Cortex M3

Figure 5.1 shows the contribution of the different parts of the algorithm to the cycle count. AddRoundKey and SubBytes/Shiftrows use the same amount of cycles both in en- and decryption mode. InvMixColumns needs more cycles than MixColumns as higher order multiplications are needed. Cycle counts scale linearly with increasing key size as the only difference is the number of rounds applied.

It should be noted that the energy figures in Table 5.2 does not include memory accesses. The contribution of memory accesses to total energy consumption is discussed in Section 3.2. As no information regarding the memory system is available, energy consumption due to memory accesses is hard to predict. Because of this, the figures in Table 5.2 will be used for comparison in this thesis. These figures represent a lower bound for the actual energy consumption as they only include energy dissipated in the core and not energy consumed in memory transfers.

This evaluation was done to serve as a performance benchmark in order to be able to compare hardware- versus software solutions.

Chapter 6

Existing hardware solutions

Important aspects to be considered when implementing an AES module is width of the datapath, implementation of MixColumns and SubBytes and how key expansion is to be performed. Numerous AES hardware modules have been implemented based on different design goals, for instance low area, low power or high throughput. [32], [16], [11], and [28] present different architectures for different optimization goals. In order to give the reader insight in how AES modules can be optimized for different design goals, [11] and [28] are given a brief presentation in Sections 6.1 and 6.2.

6.1 8 bit datapath example

[11] presents an architecture optimized for low area and low power consumption. It is based on an 8 bit datapath and supports en- and decryption with 128 bit keys. The AES module consists of a datapath, a 32 x 8 bit RAM array, control circuitry and a I/O module. Support for 192- and 256 bits keys could be implemented using the same datapath, but would require additional storage and more complex control circuitry. Figure 6.1 shows an overview of the architecture.

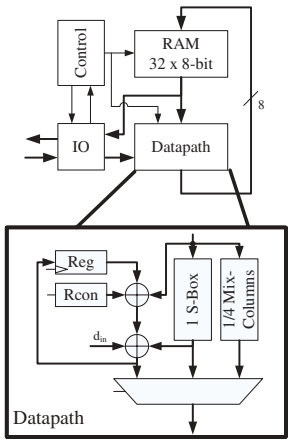

Figure 6.1: AES architecture with 8 bit datapath, [18]

The datapath includes an Sbox, a MixColumns module, 2 x 8 bit XOR arrays, an Rcon

module and an 8 bit register used to store intermediate values during key expansion and AddRoundKey. This AES module uses on-the-fly key expansion, meaning that the roundkeys are computed during en- or decryption. On-the-fly key expansion saves a considerable amount of area as only one roundkey (16 bytes) needs to be stored at a time, opposed to storing all the roundkeys which would require 176 bytes in AES128. The Sbox is able to substitute one byte per cycle and is based on direct computation using combinational logics. A pipeline stage is also incorporated in the Sbox to shorten the critical path and reduce glitching. MixColumns is computed one byte at a time. As the MixColumns operation takes four bytes as input, a pre-loading phase of three cycles is needed. Performing MixColumns on an entire state takes 28 cycles. The 8 bit architecture enables Shiftrows to be performed using appropriate addressing to the RAM. The datapath is controlled using a finite state machine which also controls which part of the RAM that is to be written to at a given time. This implementation performs encryption and decryption in 1032 and 1165 cycles, respectively. These cycle counts include I/O operations. The total area of the implementation is 3400 gates. For further details, refer to [11].

6.2 32 bit datapath example

[28] proposes an AES module utilizing a 32 bit datapath. A 32 bit datapath enables a whole column to be processed simultaneously. The datapath consists of modules performing three of the four basic steps of AES in one cycle, namely MixColumns, SubBytes and AddRoundKey. The fourth step, Shiftrows, is performed in the dataregister. As the AES state consists of four columns, one round takes $4+1 = 5$ cycles. SubBytes, MixColumns, and AddRoundKey are performed columnwise in the first four cycles while Shiftrows is performed on the entire state in the fifth cycle. Figure 6.2 depicts the architecture presented in [28].

Like the architecture presented in Section 6.1, this implementation utilizes on-the-fly key expansion. The circuitry used in key expansion can be seen on the top right hand side of Figure 6.2. In consists of four 32 bit arrays of XOR'gates and four 32 bit 2:1 muxes. This enables the next roundkey to be computed in one cycle, both in en- and decryption mode. The fixed datapath in this architecture performs SubBytes, MixColumns and AddroundKey, in that order. In decryption mode, the order should be SubBytes followed by AddroundKey and MixColumns. To compensate for this switch in decryption mode, [28] proposes to use an extra MixColumns module performing inverse MixColumns on the roundkey. This produces correct result because MixColumns and InvMixColumns are linear operations [22]. Equation 6.1 shows equality due to the linear property of MixColumns.

$$MixColumns(state \oplus roundkey) = MixColumns(state) \oplus MixColumns(roundkey) \quad (6.1)$$

This implementation supports AES128, needs 54 cycles for both en- and decryption, excluding I/O operations. The area consumption is approximately 5400 gates [28]. For further details, refer to [28].

An implementation of this architecture was made for comparison with the proposed architecture presented in Chapter 7. The proposed architecture was developed using this architecture as a basis.

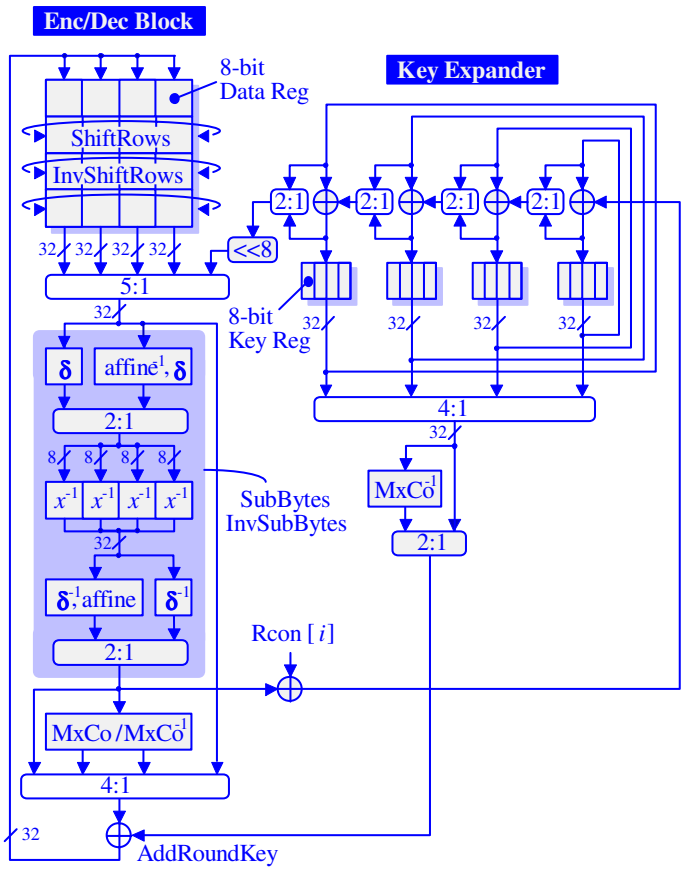

Figure 6.2: AES architecture with 32 bit datapath, [28]

Chapter 7

Hardware implementation

An important design choice to be made when implementing AES in hardware is the width of the datapath. The width of the datapath has great influence on execution time, power consumption, area, and throughput. Due to the organization of data in the AES algorithm, natural choices for datapath widths are 8, 32, 64, or 128 bits, capable of processing one byte, one column, two columns, or the entire state simultaneously. In Section 3.3.5, optimization of the datapath based on the computation to be done was discussed. In AES, most operations are byte oriented with exception of the MixColumns operation. This operation takes 32 bits as input favoring a 32 bit (or larger) datapath, in terms of energy consumption.

In Chapter 6, two architectures were presented utilizing 8- and 32 bit datapaths. It is obvious that the 8 bit datapath consumes less power, at the expense of increased execution time. As execution time of the 8 bit solution is approximately twenty times as large as in the 32 bit architecture, the 8 bit architecture would have to use 20 times less power than the 32 bit solution in order for the two solutions to be equivalent in terms of energy consumption. As the same computations are made in both architectures, the 32 bit architecture would probably consume approximately four times as much power as the 8 bit solution due to the fact that the datapath is four times as wide. These assumptions indicate that the 32 bit solution would consume roughly five times less energy than the 8 bit solution. Also, the 8 bit architecture would require a more complex control module in addition to having significantly lower throughput.

As the main design goal in this thesis is low energy consumption, a 32 bit architecture was chosen for implementation. The architecture presented in Section 6.2, was chosen as a basis because it combines short execution time with relatively modest demands to the control circuitry. As discussed in Section 2.5, wider datapaths comes with inherently improved protection against power analysis attacks giving yet another argument for using a 32 bit datapath. An even wider datapath could be used, but this would result in increased area and power consumption. In addition, a wider datapath would require key expansion to be performed faster, leading to more complex circuitry for key expansion. This will be discussed in Section 7.1.4.

While the architecture presented in Section 6.2 only supports 128 bit keys, the proposed architecture was extended to support 256 bit keys. This allows the AES module to be used in applications where AES128 does not provide sufficient security. AES256 can be implemented with small alterations in the data- and keypath, as described in Section 7.1.1 and 7.1.4. AES192 could also be implemented using mainly the same datapath, but the

key expansion- and control circuitry would need significant alterations resulting in increased area and energy consumption.

In addition to the extended functionality, alterations in the datapath were made resulting in smaller area as well as lower power consumption. These alterations are described in Section 7.1.1. This architecture allows en- and decryption to be performed in 55 and 75 cycles in AES128 and AES256 mode, respectively.

7.1 AES core

The AES module is intended to be incorporated in a microcontroller as a peripheral. A peripheral would need a bus interface in order to communicate with the CPU in the microcontroller. Due to lack of time, this interface has not been implemented, but Figure 7.10 shows an overview of how the AES peripheral could be structured. This thesis concentrates on the contents of the AES core.

7.1.1 Datapath

The proposed architecture utilizes a 32 bit datapath, able to perform MixColumns, AddRoundKey and SubBytes in one cycle. These operations are performed on one column (32 bits) at a time, enabling a whole state to be processed in 4 cycles. A fifth cycle is used to perform (Inv)ShiftRows. During this fifth cycle, the Sboxes in the datapath are used to compute a part of the next roundkey. More details on computation of roundkeys are presented in Section 7.1.4. The datapath is mainly the same as the one presented in Section 6.2, but some alterations were made:

- MixColumns and the Sbox have switched place
- The InvMixColumns module used on roundkeys has been removed
- The circuitry for key expansion has been simplified

Figure 7.1 depicts the proposed data- and keypath.

As the Sbox produces a lot of glitches [38], it was moved to the end of the datapath to prevent these glitches from propagating through the rest of the datapath consuming energy along the way. To compensate for this switch, SubBytes is performed in the initial round and not in the final round.

In decryption mode, the AddRoundKey operation is supposed to be performed prior to InvMixColumns. In the fixed datapath depicted in Figure 7.1, this is not the case. [28] solves this by performing a InvMixColumns operation on the roundkey. The proposed architecture solves the problem by adding the roundkey in the MixColumns module, presented in section 7.1.2. The XOR-gates performing AddRoundKey after MixColumns is omitted by means of multiplexing in decryption mode. This solution allows the InvMixColumns and the following mux seen on the right hand side in Figure 6.2 to be removed, saving both area and energy.

The simplifications in the key expansion circuitry will be presented in Section 7.1.4.

A few changes had to be done in the datapath to accommodate 256 bit keys. In AES256, the Sbox is applied to two different parts of the key during key expansion. To allow this, the mux prior to the Sbox was expanded to a 4:1 mux (3:1 is sufficient when only AES128

is implemented). The mux after the Sbox also has to be expanded to a 3:1 mux in order to be able to omit the addition of roundconstant during key expansion.

Like in [28], the Sboxes are used in both the key- and datapath. This sharing of the Sboxes lowers the requirement from eight to four Sboxes. The drawback is that the cycle count for each round increases from four to five. As low area is one of the design goals and the number of Sboxes have great impact on area, sharing was implemented.

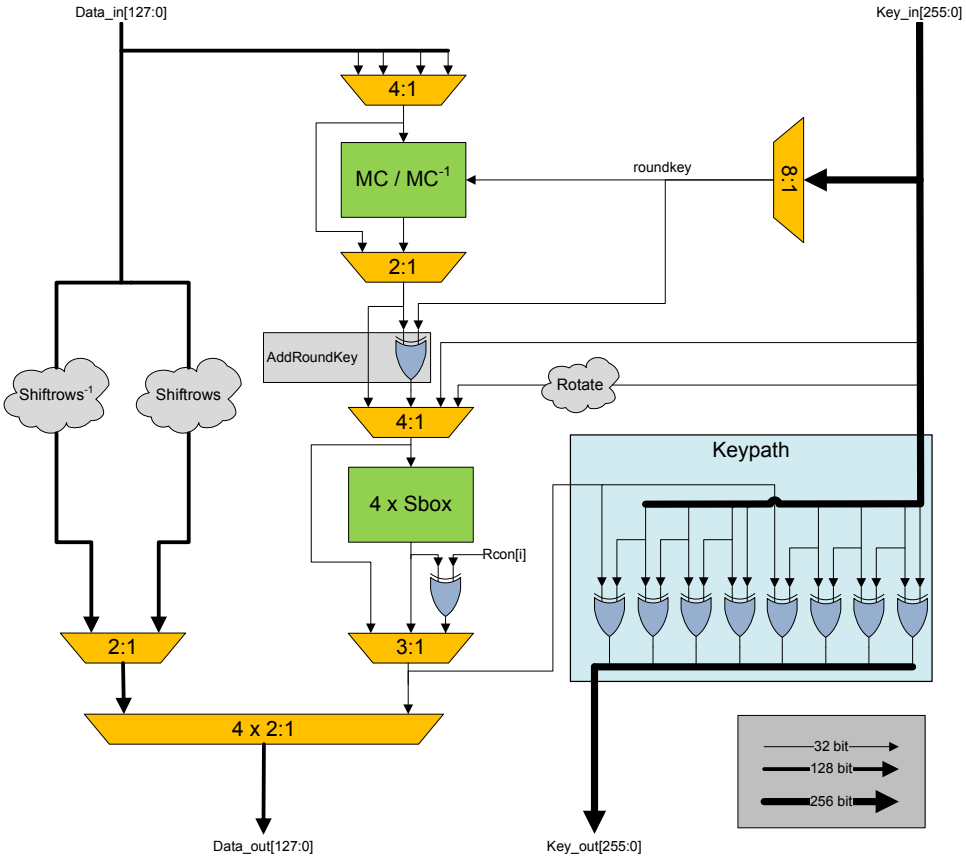

Figure 7.1: Data- and keypath

ShiftRows, InvShiftRows, and Rotate, seen in Figure 7.1 are implemented by means of wiring.

7.1.2 MixColumns

In [38], different implementations of MixColumns were explored. As the datapath in this AES module is 32 bits wide, the 32-bit versions from [38] were evaluated for use. The difference between the three MixColumns implementations evaluated in [38] is the way InvMixColumns is performed. The straightforward way to calculate InvMixColumns is to calculate M^{-1} directly as is the case in the implementation seen in Figure 7.2. M and M^{-1} represent multiplication with the matrices seen in Equations 2.3 and 2.4, respectively. The drawback with the straightforward implementation is that multiplication with M^{-1} is relatively complex, compared to the multiplications needed in the two other implementations.

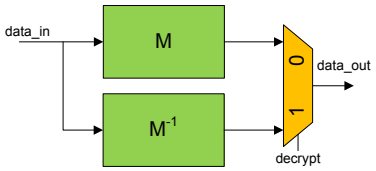

Figure 7.2: MixColumns, straightforward

In Figure 7.3, an implementation which calculates M^{-1} by adding $M^{-1} - M$ and M is shown. As $M^{-1} - M$ requires less complex computations than M^{-1}, this implementation is more energy friendly and requires less area than the straightforward implementation. The matrix $M^{-1} - M$ can be viewed in Appendix A.

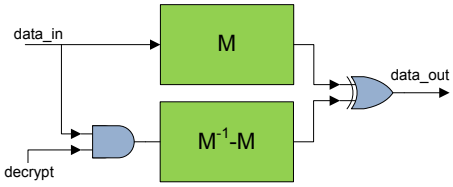

Figure 7.3: MixColumns, parallell

Figure 7.4 shows a third approach for calculating M^{-1}. Multiplication with $(M^{-1})^2$ prior to multiplication with M results in multiplication with M^{-1}. This implementation requires less area than the other implementations as the matrix $(M^{-1})^2$, which can be viewed in Appendix A, is a matrix containing multiple zeros and no high order coefficients. The XOR-gates seen in Figure 7.4 were added in order to perform AddRoundKey prior to InvMixColumns in decryption mode.

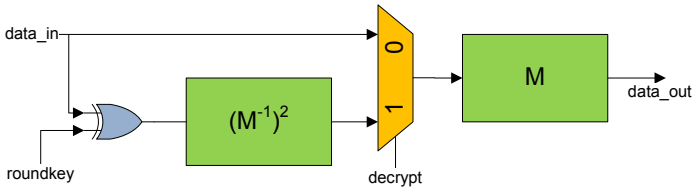

Figure 7.4: MixColumns, serial with AddRoundKey

In [38], the three implementations were synthesized using a 90 nm library and evaluated with regards to area, delay and power consumption. Table 7.1 summarizes the results. Note that the XOR-gates in Figure 7.4 were not included during these evaluations.

	Area (NAND2 eq.)	Power (Enc/dec)	Delay
Straightforward	630	62.68/62.57 μW	0.96 ns
Parallel	422	17.45/42.10 μW	0.95 ns
Serial	354	26.96/37.75 μW	1.07 ns
Serial, without NSR	393	30.15/44.94 μW	-

Table 7.1: Comparison of MixColumns implementations

The evaluation indicated that the two implementations not performing straightforward computation of M^{-1} are preferable both in terms of area and energy consumption. As mentioned in section 7.1.1, it is desirable to do AddRoundKey prior to InvMixColumns in decryption mode. This can easily be implemented in the MixColumns implementation utilizing the serial architecture. The only alterations needed are the added XOR-gates prior to the block performing multiplication with $(M^{-1})^2$, as seen in Figure 7.4. This will only have effect in decryption mode, as desired. In order to include the same functionality in the parallel architecture, an extra mux would have to be added in addition to the XOR-gates. Based on this, and the fact that this implementation has low area and power consumption, the MixColumns module utilizing the serial architecture was used in the AES module.

MixColumns and numerical strength reduction

When a 32 bit datapath is utilized, the MixColumns operation can be performed on a whole column in a single cycle. This can be taken advantage of in order to simplify the hardware needed. Numerical strength reduction (described in Section 3.3.4) is proposed applied to the matrices used in MixColumns.

The example presented in Section 3.3.4 shows how the block performing multiplication with $(M^{-1})^2$ can be simplified, leading to reduction both in area and energy consumption. In order to adapt the equations for the matrix multiplication to finite field arithmetics, addition has to be substituted with XOR and multiplication has to be substituted with finite field multiplication. The equations used to perform multiplication with M were derived using the same procedure and can be seen in Appendix C. As can be seen in Table 7.1, applying numerical strength reduction reduces area with 10% and power consumption with 11% and 16% in en- and decryption mode, respectively.

7.1.3 Sbox

Designing a compact and energy efficient Sbox is one of most important tasks when implementing AES in hardware. Especially, when a 32 bit datapath is used, the Sbox has to be well designed as four of them are needed. In [38], three different strategies for performing SubBytes were explored. One of the implementations was an architecture based on two 256 byte look up tables. One look up table contained pre computed values for the multiplicative inversion in sequence with the affine transformation, S_{rd}. The other look up table contained the values for inverse affine transformation in sequence with multiplicative inversion, S_{rd}^{-1}. The outputs from these look up tables were muxed in order to choose between SubBytes and InvSubBytes. Figure 7.5 depicts the architecture.

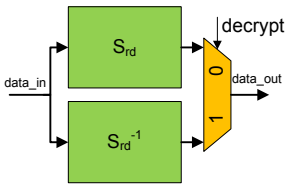

Figure 7.5: Sbox with two look up tables

The main disadvantage with the implementation in Figure 7.5 is that the two look up tables requires a considerable amount of area. Combinational logics was used to implement the look up tables. Using ROM instead might lead to a better result, but ROM generation has to be done using special tools which were not available.

The second architecture evaluated in [38] utilized one 256 byte look up table. This table contained the multiplicative inverses for each element in $GF(2^8)$. The affine transformation and its inverse were computed using combinational logics after and before the multiplicative inversion. The architecture can be viewed in Figure 7.6.

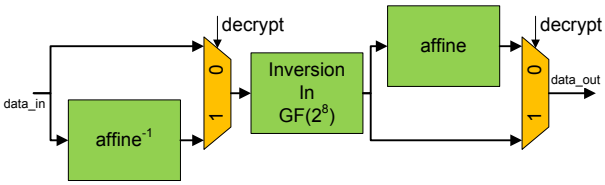

Figure 7.6: Sbox with one look up table

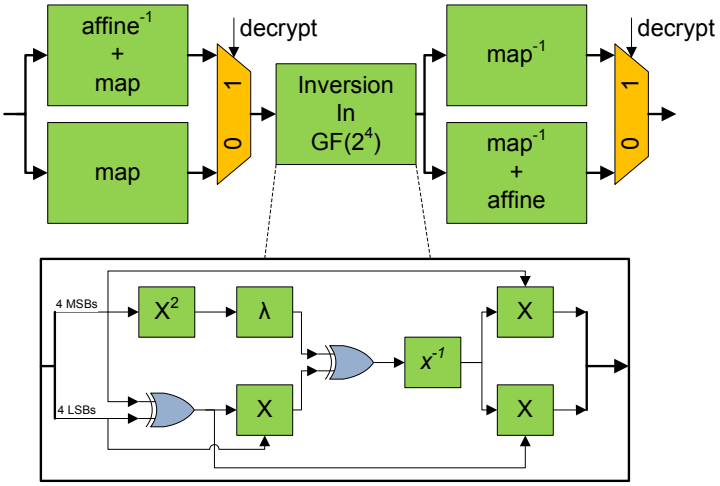

Figure 7.7: Sbox with inversion in $GF(2^4)$

A third architecture was explored utilizing isomorphic mapping to compute the multiplicative inverses without using large look up tables. Theory concerning multiplicative inversion using isomorphic mapping is presented in Section 2.4. Mapping to the lower order field can be interpreted as a matrix multiplication, similar to the affine transforms. This

enables these to be combined in order to simplify the hardware. Figure 7.7 depicts the architecture. The matrices used during mapping and affine transforms can be viewed in Appendix A.

The three architectures were synthesized using 90 nm technology and evaluated with regards to area, delay and power consumption in [38]. Table 7.2 summarizes the results.

	Area (NAND2 eq.)	Power	Delay
2 LUTs	1408	95.49 μW	1.15 ns
1 LUT	821	88.43 μW	2.30 ns
Mapped to $GF(2^4)$	300	64.19 μW	2.93 ns

Table 7.2: Comparison of SubBytes implementations

The evaluation in [38] clearly indicates that the architecture utilizing isomorphic mapping is preferable both in terms of area and power consumption. The delay in this architecture is somewhat larger than the delay in the other implementations. Lowering the voltage supply on the implementation using two look up tables such that its delay matches the delay of the implementation using isomorphic mapping might lead to a different conclusion. This would require multiple voltage domains which is not always available in microcontrollers. As long as the target frequency is reached with the version using isomorphic mapping, this architecture is preferred.

Another important aspect to be considered when choosing Sbox architecture is the area. In an AES module with 32 bit wide datapath, four Sboxes are needed to fully utilize the architecture. As the implementation using isomorphic mapping is considerably smaller than the others, this will lead to a relatively large area reduction. Based on this, the Sbox utilizing isomorphic mapping was chosen for the AES module.

7.1.4 Key expansion

On-the-fly key expansion was implemented as this dramatically reduces the amount of storage needed. Calculating the roundkeys prior to en- or decryption would require 240 bytes to be stored in AES256 mode. Using on-the-fly key expansion lowers the storage requirement to 32 bytes. A consequence of on-the-fly key expansion is that the key provided to the AES module in decryption mode needs to be last roundkey used in encryption mode, not the original cipherkey. This small disadvantage is acceptable as the reduction in hardware cost is relatively large.

[28] presents an on-the-fly key expander which computes the entire 128 bit roundkey in one cycle both in en- and decryption mode. In encryption mode, one column in the roundkey is constructed using the previous key column as an input, as seen in Equations 7.1 through 7.4. In order to compute the whole roundkey in one cycle, each roundkey column needs to propagate to the next, creating a relatively long combinational path. As the output from the Sbox (containing a lot of glitches, [38]) is used as an input to these calculations, these glitches will propagate through the combinational logics used in key expansion consuming energy along the way.

In this architecture, the order in which the data columns are processed has no impact on the result, which also enables the columns in the roundkeys to be computed in an arbitrary order and not necessarily simultaneously. This fact can be taken advantage of, simplifing

the combinational logics needed for key expansion.

An alternative architecture for the key expander is proposed, computing the roundkey over four cycles, removing the need for muxes in key expansion circuitry in addition to shortening the combinational path to one XOR-gate opposed to four XOR-gates and three 2:1 muxes, as used in [28]. This alteration does not reduce the overall speed of the AES module as the key expansion is performed in parallel with the data processing.

In Equations 7.1 through 7.8, K^i_{cj} represents column j in roundkey i while K^{i+1}_{cj} represents column j in the next roundkey $(i+1)$. The function $rsc()$ rotates the input column one byte before using the Sbox to substitute the column byte by byte and finally adding the roundconstant. Equations 7.1 through 7.4 describes the relationship between K^i_{cj} and K^{i+1}_{cj} in encryption mode while Equations 7.5 through 7.8 describes the relationship between K^{i+1}_{cj} and K^i_{cj} in decryption mode.

$$K^{i+1}_{c3} = K^i_{c3} \oplus rsc(K^i_{c0}) \tag{7.1}$$
$$K^{i+1}_{c2} = K^i_{c2} \oplus K^{i+1}_{c3} \tag{7.2}$$
$$K^{i+1}_{c1} = K^i_{c1} \oplus K^{i+1}_{c2} \tag{7.3}$$
$$K^{i+1}_{c0} = K^i_{c0} \oplus K^{i+1}_{c1} \tag{7.4}$$

$$K^i_{c3} = K^{i+1}_{c3} \oplus rsc(K^i_{c0}) \tag{7.5}$$
$$K^i_{c2} = K^{i+1}_{c2} \oplus K^{i+1}_{c3} \tag{7.6}$$
$$K^i_{c1} = K^{i+1}_{c1} \oplus K^{i+1}_{c2} \tag{7.7}$$
$$K^i_{c0} = K^{i+1}_{c0} \oplus K^{i+1}_{c1} \tag{7.8}$$

Examining these equations, keeping in mind that the columns can be calculated in an arbitrary order, reveals that the key expansion circuitry seen in Figure 6.2 can be simplified. Figure 7.8 shows an illustration on how the roundkeys can be computed using only XOR-gates. The columns in each step shows which roundkey words are present in the keyregister at a given time. During encryption, the initial contents of the keyregister is K^i and the next roundkey K^{i+1} is to be calculated. In decryption mode, the roundkeys are calculated in reverse order, starting with K^{i+1}.

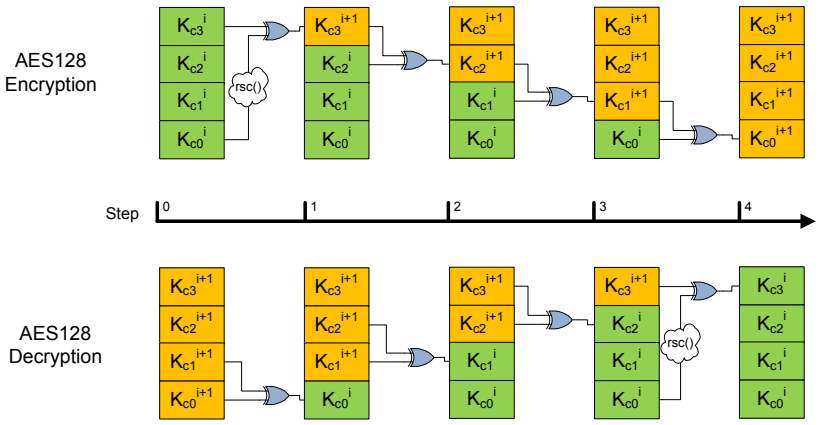

Figure 7.8: Key expansion in AES128

Altering the order in which each column in the next roundkey is computed, the muxes [28] uses in key expansion can be omitted. This does not only save area, it also eliminates the relatively long path through the muxes and XOR-gates seen on the top right hand side of Figure 6.2. Shortening the combinational path in key expansion circuitry prevents propagation of glitches from the Sbox resulting in lower energy consumption.

A similar approach is used in AES256 mode, keeping the combinational logics needed to a minimum. An illustration of key expansion with 256 bit keys can be viewed in Appendix B. The circuitry for key expansion can be viewed on the lower right hand side of Figure 7.1.

7.1.5 Sequencer

The sequencer is the module controlling the datapath and which parts of the key- and dataregisters that are to be written to at a given time. It consists of a finite state machine, FSM, and a counter keeping track of the en- or decryption progress. The FSM controls when the counter is to be incremented as well as the control signals to the datapath, and write strobes, *data_we* and *key_we*, to the registers. The roundconstants used in key expansion are also computed in this module. Using the three most significant bits of the counter as a selectsignal, the roundconstants are calculated using combinational logics. Figure 7.9 shows the states in the FSM.

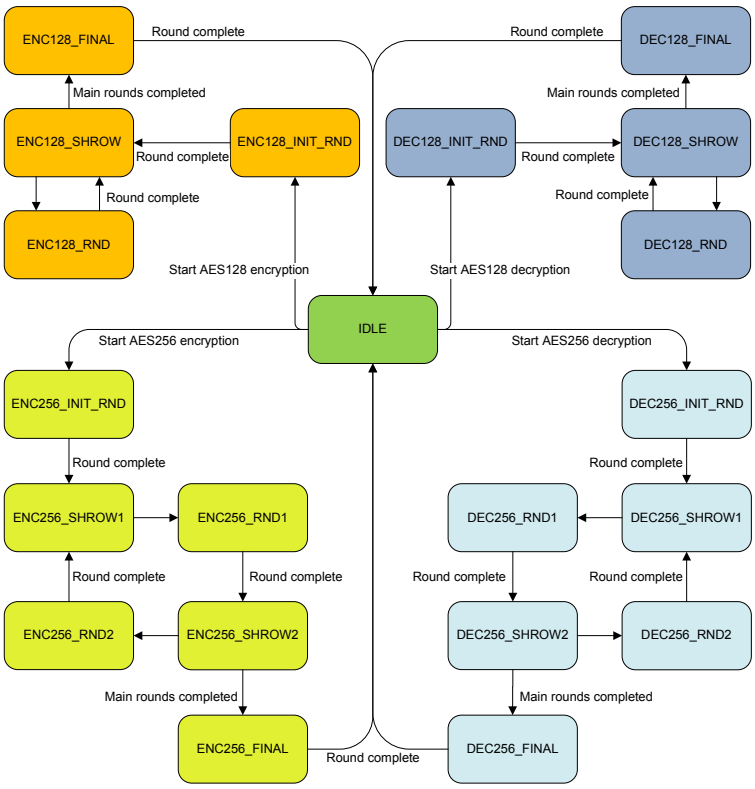

Figure 7.9: Sequencer, finite state machine

In all states named XXX_RNDX and XXX_FINAL, the counter is incremented, a round is complete when the two least significant bits in the counter is {11}. In states named

XXX_SHROWX, the Shiftrows operation is performed and the state machine will only reside in these states for one cycle at a time. Table B.1 in Appendix B shows an overview of the states and brief descriptions of the actions taken in each state.

In order to reduce power consumption, clock gating is used on all registers in the design. Both the state- and counter register are equipped with clock gates. In addition, the signals *key_we* and *data_we* are used as enable signals for clock gates in the data- and key registers. The sequencer is shown on the lower right hand side in Figure 7.10.

7.2 AES peripheral module

The AES module is intended to be incorporated in a microcontroller as a peripheral module. In order to access the data- and keyregister through the system bus, an interface module has to be made. In this thesis, only the AES core has been implemented, but an example of an interface module and its features will be presented to give the reader an image of how the complete AES peripheral module might be organized. Figure 7.10 depicts an example peripheral, the interface module consists of everything but the AES core and bus.

The control-, data-, and keyregisters would be memory mapped for easy access through the bus interface. A simple control module handles interaction between the control module in the AES core (the sequencer) and the control register. The control module would also handle DMA- and interrupt requests to the system.

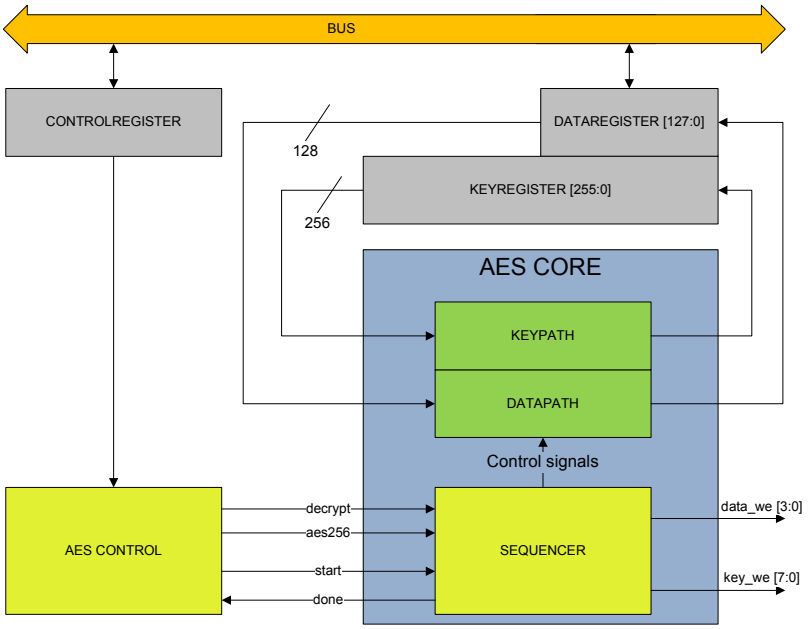

Figure 7.10: AES module

DMA support

In a microcontroller with low power mode functionality, autonomous AES processing could lead to a potentially large reduction in energy consumption. If the DMAC could be pro-

grammed to applying AES on large datablocks without inferring the CPU, the CPU could reside in a low power mode, saving energy.

Features in the AES module which could simplify DMA operation need to be specialized for the DMAC which is to be used, but the following examples would most likely ease DMA operation.

- Key buffering. In AES128, half the key register is used as a buffer.
- Datastart. Writing 128 bits to the data register starts AES processing.
- Xorwrite. When writing to the data register, the new content is XOR'ed with the old.

When on-the-fly key expansion is being used, the contents of the key registers are altered during AES processing. In order to use the same key multiple times without writing it through the bus interface each time, one part of the 256 bit key register could be used as a buffer in AES128. This limits the need for energy consuming data transfers between each en- or decryption in addition to simplifying DMA operation.

Ability to start AES processing automatically when new data is written to the data registers is another feature which would ease DMA operation. This eliminates the need for writing an additional start command to the control register for each data block.

Four of the five modes of AES presented in Section 2.3.6 requires XOR'ing of the new and old data in the data registers. This could be performed by the CPU, but adding this functionality in the AES peripheral requires few modifications (32 XOR-gates and a mux in a 32 bit system) and enables other modes than ECB to be performed autonomously using DMA.

Figure 7.11 illustrates how AES128 in CBC mode can be performed using DMA. Enabling datastart, xorwrite, and keybuffering allows encryption in CBC mode to be performed using only two DMA channels. The AES peripheral would issue a DMA request upon completion of a datablock and DMA channel two places the ciphertext in a specified memory location. DMA channel one will then write the new plaintext block to the peripheral where it is XOR'ed with the ciphertext. When 128 bits of plaintext is written to the data registers, AES processing automatically begins as datastart is enabled. This is repeated until all data is encrypted and the DMAC issues a request to the CPU. The other modes can be

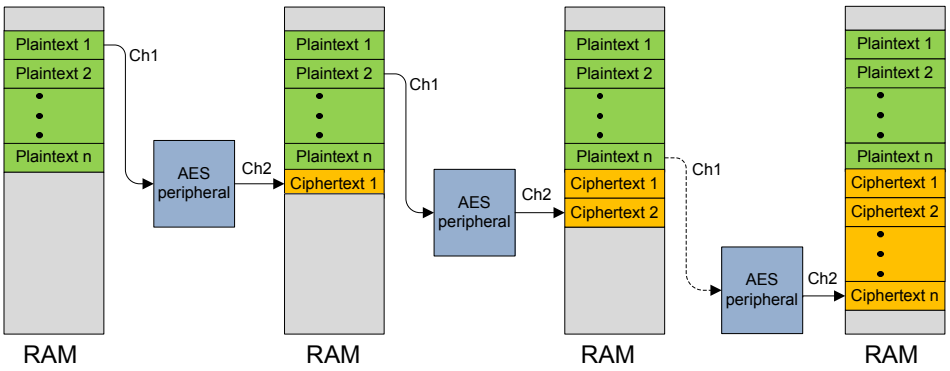

Figure 7.11: AES using DMA in CBC mode

implemented in a similar manner. When AES256 is to be performed, keybuffering can not

be used and a third DMA channel will need to be configured to updating the keyregister between each block en- or decryption.

7.3 AES core with 128 bit datapath

AES is an algorithm which easily can be parallelized and the architecture presented in this thesis needs very few modifications in order to expand the datapath resulting in a significant speedup. An AES module utilizing a 128-bit datapath was implemented using basically the same architecture as the one with 32 bit datapath, the difference being that the datapath is four times as wide. The 128 bit architecture uses 20 Sboxes and four MixColumns modules. 16 Sboxes are used in the main datapath while 4 Sboxes are used in key expansion. The delay through the datapath remains approximately the same as in the 32 bit version as the only change in the datapath is duplication of the processing elements. The architecture is depicted in Figure 7.12.

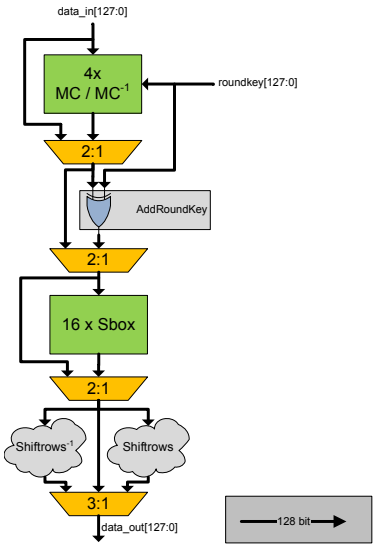

Figure 7.12: 128 bits datapath

Utilizing a 128 bit datapath enables en- and decryption to be performed in 11 cycles in AES128 mode. This results in a significant increase in throughput, at the cost of increased area and power consumption. As one round is completed in one cycle, computation of roundkeys also has to be performed in one cycle. Hence, the simplified key expansion circuitry presented in Section 7.1.4 can not be used. To enable single cycle key expansion, the key expansion circuitry presented in Section 6.2 was utilized.

Chapter 8

Verification and synthesis

8.1 Verification

In this work, initial verification of the AES core has been performed on different levels throughout the design process. The submodules MixColumns and Sbox were both verified using randomly generated stimuli in [38]. Golden devices were used to verify correctness. This verification was performed both on RTL code and netlist.

The different implementations of the AES core went through initial verification using a Verilog testbench. This testbench performed encryption 100 times, before 100 decryptions. Contents of the data- and keyregisters were compared to precomputed correct values between each en- and decryption. The precomputed correct values were derived using the software version of AES presented in Chapter 5.

As the large key and datablock sizes used in AES makes complete verification of the module practically impossible, known answer tests provided by [14] were used to verify correctness. For these tests, an interface module for the proposed AES core was provided by Energy Micro and it was simulated using a model of a complete system including the ARM Cortex M3 core, bus and AES peripheral module. The tests were written in C code, and performed the following verification steps:

- Varying key, constant plaintext, 128 bit key size
- Constant key, varying plaintext, 128 bit key size
- Varying key, constant plaintext, 256 bit key size
- Constant key, varying plaintext, 256 bit key size

No errors were found, indicating that the AES core works correctly.

8.2 Synthesis

The AES core, key- and dataregisters were synthesized with the ARM Sage-X 180 nm standard cell library using Synopsys' Design Vision. The synthesis tool was configured for

synthesis with low power consumption as main goal and a target frequency of 32 MHz. The synthesis scripts can be viewed in Appendix D.3.

As the cell library only was available in the process corners fast and slow, all versions of the AES core were evaluated using both. Key attributes for the process corners fast, typical, and slow are summarized in Table 8.1.

Parameter	Fast	Typical	Slow
Supply Voltage	1.98 V	1.8 V	1.62 V
Temperature	-40°C	25°C	125°C
Process derating factor	0.793	1	1.27

Table 8.1: Parameters for libraries

The energy figures for the ARM Cortex M3 (used for comparison in this thesis) are also based on the ARM Sage-X 180nm library, but most likely using the typical process. As an approximation to what the result would be using typical, Equation 8.1 was used to derive the energy figures presented in Chapter 9.

$$P_{typical} = \frac{1}{2} \times \left(\frac{P_{fast}}{V_{fast}^2} + \frac{P_{slow}}{V_{slow}^2} \right) \times V_{typical}^2 \tag{8.1}$$

Equation 8.1 cancels out the power supply factor from simulations done using slow and fast. After calculating the average between the two, multiplication with the power supply in the typical case is done resulting in an approximation of what the power figure would be using the typical library. An approximation of the power consumption in the typical case is done in order to give a more just comparison with the energy figures from software.

Maximum operating frequencies are based on delay through the critical path when the slow version of the library was used.

8.3 Power estimation

As the design in this thesis is relatively small, gate-level simulations on netlist was used in order to estimate power. This approach gives the most accurate result as every node in the design is included in the estimations. Delays through logic gates was also included making it possible for power consumed due to glitches to be included in the estimations. Other approaches mentioned in Section 3.4 could be used in order to speed up simulations, but as gate-level simulations on a design of this size are completed in a matter of minutes, the most accurate approach was chosen.

The netlists provided by the synthesis tool was evaluated using a testbench performing en- or decryption 100 times. The operating frequency was 32 MHz and Mentor Graphics' Modelsim. Toggle data in the different modes (AES128 and AES256 en- and decryption) was collected and evaluated using Synopsys' Power Compiler.

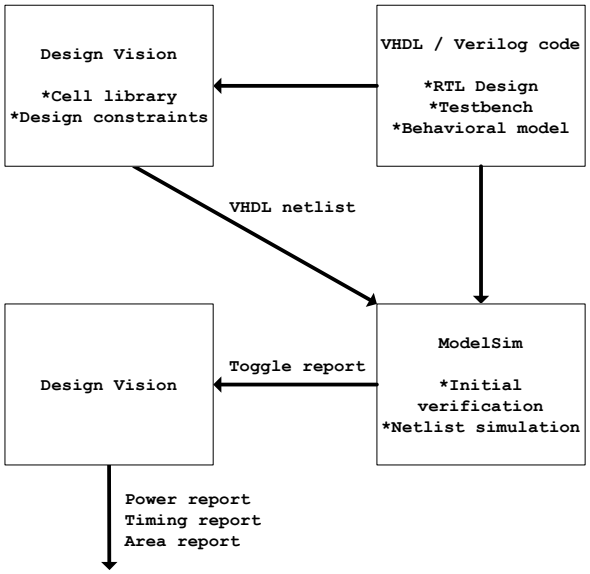

Figure 8.1: Design flow

Figure 8.1 depicts the flow from RTL code to generation of netlists and power estimates. The testbenches used in verification and power estimation can be viewed in Appendix D.

Chapter 9

Evaluation

In this thesis, multiple hardware implementations of AES has been developed in addition to a software version. Evaluation and comparison of the different implementations will be made in this chapter.

9.1 Impact of alterations in data- and keypath

In Chapter 7.1.1, three alterations in the data- and keypath were presented and this section will evaluate the impact of each of these alterations. Figures 9.1 and 9.2, shows power- and area figures for different AES implementations. The different versions are:

- V1: An implementation of the AES module described in Section 6.2. Support for AES128.
- V2: Similar to V1, except the InvMixColumns module used on the key has been removed as described in Section 7.1.1. Support for AES128.
- V3: Similar to V2, but the combinational logics for key expansion has been simplified as described in Section 7.1.4. Support for AES128.
- V4: The proposed architecture. Similar to V3, but the Sbox and MixColumns have switched place. Support for AES128.
- V5: The proposed architecture. Similar to V4. Support for AES128 and AES256.

The same versions of the Sbox and MixColumns has been used in all implementations in order to give a just comparison.

V1, architecture chosen as basis

V1 is an implementation of the architecture proposed in [28]. The data- and keypath were adapted to the interface of the AES core, presented in Section 7.2.

V2, Removal of InvMixColumns

In implementation V2, the InvMixColumns module applied to the roundkey in decryption mode and the subsequent mux has been removed. Consequently, the area was reduced by

approximately 7.5%. Power consumption is also reduced by 4.5% and 3.3% in en- and decryption mode, respectively. In encryption mode, InvMixColumns is not used on the roundkey, but the module would still consume power as its inputs switches. A data gate could prevent this at the cost of additional area and increased power consumption in decryption mode. The reduction in decryption mode is due to the fact that it is more efficient to perform InvMixColumns on the data and roundkey combined rather than performing InvMixColumns separately before combining the results.

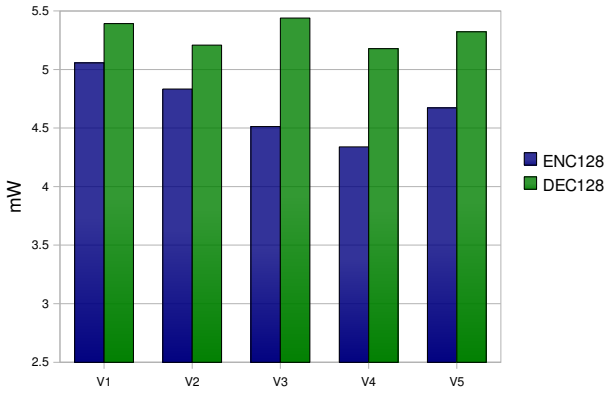

Figure 9.1: Comparison of power consumption

V3, Simplification of key expansion circuitry

As Figure 9.2 indicates, simplification of the key expansion circuitry reduces the area by 7.6% compared with implementation V2. Power consumption in encryption mode was also reduced by 6.6% compared to implementation V2. In implementation V2, key expansion in encryption mode is performed in one cycle requiring propagation of results through a relatively long combinational path. As input to this computation is output from the Sbox, the glitches produced in the Sbox will also be propagated through the key expansion circuitry consuming energy along the way.

In decryption mode however, the power consumption is slightly increased (4.4% compared to V2). This is due to the fact that key expansion in decryption mode does not require propagation of results in order to compute the next roundkey. Consequently, the glitches from the Sbox are not propagated through the key expansion circuitry. The increase of power consumption in decryption mode is due to the fact that the sequencer has to be slightly more complicated in order to compute the roundkeys correctly when using the simplified key expansion circuitry.

V4, Swapping MixColumns and Sbox

Implementation V4 is similar to V3, the difference being that MixColumns and the Sbox has switched place. In [38] it was shown that the Sbox used in this implementation produces quite a lot of glitches. As these glitches will propagate through the circuitry following the Sbox, a swap was made in order to minimize the circuitry which these glitches propagate through. The effect of this swap is power reduction of 3.8% and 4.8% in en- and decryption mode, compared to implementation V3. As the swap requires an extra mux, the area is

increased by 1.1% compared to implementation V3. It should be noted that a different implementation of the Sbox might yield different results due to this swap.

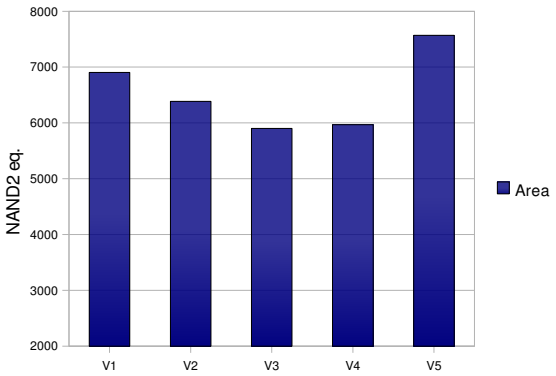

Figure 9.2: Comparison of area

V5, support for AES256

Figures 9.2 and 9.1 shows that V5 has both higher power- and area consumption compared to V4. Support for AES256 requires a slight change in the datapath, duplication of the key expansion circuitry and a more complex sequencer. In addition, extra registers are needed to store the key. The combination of these alterations leads to increased area, longer critical path and slightly increased power consumption.

9.2 Evaluation of architectures

In addition to the proposed 32 bit architecture, an architecture utilizing a 128 bit datapath was implemented. This implementation allows one round to be completed in one cycle resulting in a reduction in execution time by a factor of five compared to the proposed 32 bit architecture. Table 9.1 summarizes key figures for four different implementations. The power figures are based on an average between en- and decryption in AES128 mode.

AES w/128 bit datapath

As can be seen in Table 9.1, the implementation with 128 bit datapath is favorable in terms of energy. This is due to the simplified datapath and control circuitry. Although this implementation in the most energy friendly, the area and power consumption could make this architecture impractical for implementation in a microcontroller, depending on the budgets for area and power. In addition, if power gating is to be implemented, the size of the power gate would have to be relatively large in order to supply enough power. This would increase the overhead energy needed to turn on the gate. If power gating is not used, the increased area would lead to an increase approximately by a factor of two in leakage power, as leakage power is proportional to area.

Figure 9.3 shows how the different parts of the design contribute to the critical path in the proposed 32 bit architecture (V4). In the 128 bit architecture, the delay in the sequencer will be greatly reduced as no selection of data is needed (the whole state is processed each round). This leads to an increased maximum frequency.

The throughput of the 128 bit solution in superior compared with the other architectures. Lowering the throughput to match the 32 bit architectures, for instance by means of frequency- and/or voltage scaling could lead to an even more energy friendly solution. If this is possible in the microcontroller in which the AES module is to be incorporated, this solution should be considered if the area budget allows it.

	V1 [28]	V4	V5	AES w/128 bit datapath
Area (NAND2 eq.)	6904	5964	7536	16505
Max frequency	50.4 Mhz	53.4 MHz	43.5 MHz	73.4 MHz
Throughput @32MHz	74.5 Mbps	74.5 Mbps	74.5 Mbps	372.4 Mbps
nJ/datablock, AES128 encryption	8.69	7.46	8.03	5.63
nJ/datablock, AES128 decryption	9.26	8.90	9.14	5.97
nJ/datablock, AES256 encryption	-	-	11.02	-
nJ/datablock, AES256 decryption	-	-	12.80	-
Power @32MHz	5.23 mW	4.76 mW	5.00 mW	16.88 mW

Table 9.1: Key figures for AES implementations

Proposed architecture

Table 9.1 contains key figures for two implementations utilizing the proposed datapath (V4 and V5). The difference between the two being that V4 does not support 256 bit keys.

Comparison of the architecture chosen as a basis (V1) and the proposed architecture (V4) reveals that the alterations resulted in a significant improvement both in area and energy consumption. In decryption mode, a 3.9% energy reduction is achieved due to the data- and keypath alterations. In encryption mode however, the reduction is 14.2%. In three of the five modes of AES presented in Section 2.3.6, only encryption is performed making the energy savings in encryption mode the only one of relevance in many applications. The alterations in the data- and keypath yielded an area reduction of 13.6%, reducing production costs and leakage currents. As discussed in Section 3.1.3, leakage power does not have great impact on the overall power consumption in older technologies, but as the transistor sizes continue to decrease, the contribution of leakage currents should be taken seriously. In addition, leakage currents increase exponentially with the temperature, making its contribution to total power larger for applications operating at high temperatures.

When AES256 is to be supported (V5), the area obviously increases as the keyregister needs to be twice as large. Although V5 has a more complex sequencer than V1, the energy figures in AES128 are still reduced compared to V1. As seen in Table 9.1, V5 consumes 7.6% and 1.3% less energy in AES128 en- and decryption. This is due to improvements in the data- and keypath in the proposed architecture.

Figures 9.3 and 9.4 show the contribution of the different parts of the datapath to the delay in implementations V4 and V5, respectively.

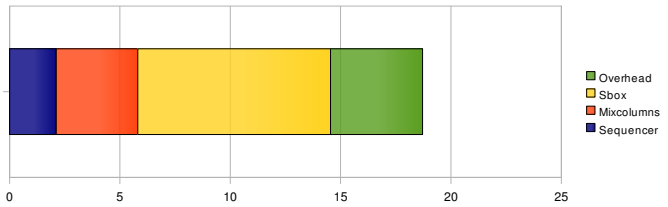

Figure 9.3: Delay through the datapath, V4

As can be observed, the main contributer to increased delay in V5 is the sequencer. The increased complexity needed to accommodate AES256 results in additional delay, area, and energy consumption.

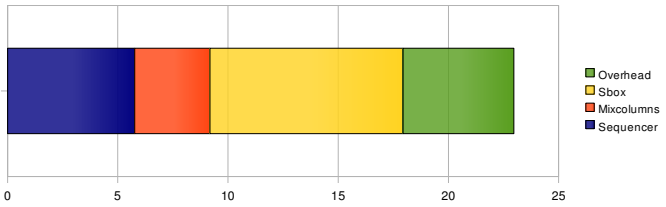

Figure 9.4: Delay through the datapath, V5

Voltage scaling could be used on the proposed architecture to lower energy consumption. If the scaling increases the delay in such a way that the target frequency is not reached, techniques like pipelining could be used to decrease the delay and maintain throughput. When synthesized with the slow version if the ARM Sage-X 180nm library, with 1.62V voltage supply, the target frequency is reached for both V4 and V5, and no pipelining is needed.

Cost/Performance balance

In this thesis, the implementations are to be evaluated in terms of energy, area and speed. Figure 9.5 shows energy per encryption and area for different implementations.

It is clear that the architecture with 128 bit datapath is favorable in terms of energy. And as can be seen in Table 9.1, it has superior throughput compared to the 32 bit implementations. However, when area consumption is taken into account, one of the 32 bit implementations may be seen as a better solution. The proposed architecture without AES256 functionality (V4) has the lowest energy per encryption and area among the 32 bit architectures, making this the most favorable solution when area is part of the cost function. If AES256 is needed, the proposed architecture (V5) still yields better energy per encryption than V1 in AES128 mode.

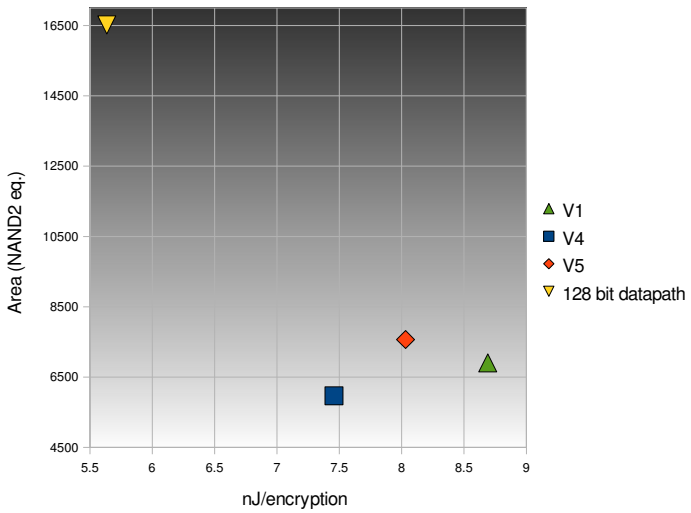

Figure 9.5: Area vs energy

9.3 Hardware versus software

A criterion for incorporating an AES hardware peripheral in a microcontroller is that some sort of performance gain is achieved, for instance increased throughput or decreased power/energy consumption. Table 9.2 summarizes energy consumption and throughput for the hardware and software solutions.

	Software	Hardware	Percentage
AES128 encryption, [nJ/128bits]	333	8.03	2.4%
AES128 decryption, [nJ/128bits]	407	9.14	2.2%
AES256 encryption, [nJ/128bits]	469	11.02	2.3%
AES256 decryption, [nJ/128bits]	576	12.80	2.2%
Throughput, [Mbps]	2.95	74.5	2525%

Table 9.2: Software versus hardware

It should be noted that the hardware figures in Table 9.2 does not account for I/O operations. Energy consumption due to I/O operations are hard to predict and were therefore not included in the calculations. As mentioned in Section 5.2, energy due to memory accesses is not included in the software figures either. Table 9.2 shows that the hardware implementation is superior to software both in terms of energy consumption and throughput. Energy consumption is reduced by over 97% while throughput is increased bu a factor of 25.

Furthermore, an AES peripheral greatly reduces the amount of memory accesses needed for AES processing, leading to even larger energy savings when performing AES in a dedicated hardware module. In addition, a microcontroller with DMA support would allow the CPU to enter a low power mode while the peripheral can process large amount of data, issuing a interrupt request upon completion, further increasing the possibilities for saving energy.

Applications involving AES processing would require significantly less energy resulting in prolonged battery life if an AES peripheral is included in the microcontroller.

In Section 3.2 it was said that only 3.54% of the energy consumed during software execution was spent on *useful* arithmetics and that this percentage is comparable to what a dedicated hardware module would use. This concurs with the percentages presented in Table 9.2.

Chapter 10

Conclusions

In this thesis, an AES core intended for incorporation in a microcontroller has been developed. The main design goal has been low energy consumption while maintaining a good cost/performance balance. An existing solution utilizing a 32 bit datapath was chosen as a basis. By simplifying and altering the datapath in the existing solution, area was reduced by 13.6% while energy consumption was lowered with 14.2% and 3.9% in AES128 en- and decryption, respectively.

The proposed architecture was also modified in order to accommodate 256 bit keys. This led to an increase in area by 9.2% compared with the existing solution chosen as basis. Although the area was increased, energy per encryption was still reduced by 7.6% and 1.3% in AES128 en- and decryption. The AES module with support for both AES128 and AES256 consumes an area equivalent to 7536 NAND2 gates, has a throughput of 74.5 Mbps @ 32 MHz and an average power consumption of 5 mW during operation.

Further parallelization was also explored by implementation of an AES module utilizing a 128 bit datapath. This solution yielded lowest energy per encryption and highest throughput, but the relatively large area led to a poor cost/performance balance.

A software solution optimized for 32 bit architectures has been implemented, evaluated on an ARM Cortex M3 MCU, and compared to the hardware solution. The results in this thesis indicate that performing AES in a dedicated hardware module leads to reduction in energy per encryption approximately by a factor of 40. In addition to the dramatic reduce in energy consumption, the throughput was increased by a factor of 25.

Numerical strength reduction was applied to MixColumns both in the software and hardware implementations allowing the MixColumns procedure to be performed using significantly less energy. In software, an 84% reduction in cycle count was attained leading to significantly reduced energy consumption. In hardware, energy consumption was reduced by 11% and the area was decreased by 10%. When performing the inverse, InvMixColumns, the energy savings were even larger with 87% and 16% in software and hardware, respectively.

Further work

An AES core has been implemented in this work. In order to incorporate this core in a microcontroller, an interface module has to be developed. This module would implement the bus interface in addition to managing the interrupt- and DMA requests. One of the main challenges when designing the interface module would be to make DMA operation

in the different AES modes as simple as possible. Initial verification of the core has been carried out, but additional verification should be performed on the AES peripheral when the interface module is included.

Bibliography

[1] Kubilay Atasu, Luca Breveglieri, and Marco Macchetti. *Efficient AES implementations for ARM based platforms.* Association for Computing Machinery, 2004.

[2] Guido Bertoni, Luca Bregvegliere, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. *Efficient Software Implementation of AES on 32-bit Platforms.* CHES, 2002.

[3] Shekhar Borkar. *Design challenges of technology scaling.* IEEE, Micro, 1999.

[4] Yun Cao and Hiroto Yasuura. *A system-level energy minimization approach using datapath width optimization.* International Symposium on Low Power Electronics and Design, 2001.

[5] John Catsoulis. *Designing Embedded Hardware, 2nd edition.* O'Reilly, 2005.

[6] Ananta P. Chandrakasan and Robert W. Brodersen. *Low Power Digital CMOS Design.* Kluwer Academic Publishers, 1995.

[7] William J. Dally, James Balfour, David Black-Shaffer, James Chen, R. Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. *Efficient Embedded Computing.* Computer, Vol 41, 2008.

[8] Joan Deamen and Vincent Rijmen. *The Design of Rijndael.* Springer, 2002.

[9] Hans Dobbertin, Lars Knudsen, and Matt Robshaw. *The Cryptanalysis of the AES - A Brief Survey.* Springer Berlin / Heidelberg, 2005.

[10] Morris Dworkin. *Recommendation for Block Cipher Modes of Operation.* National Institute of Standards and Technology, 2001.

[11] M. Feldhofer, J. Wolkerstorfer, and V. Rijmen. *AES implementation on a grain of sand.* Information Security, IEE Proceedings, 2005.

[12] Daniel D. Gajski. *Principles of Digital Design.* Prentice Hall, 1996.

[13] F.K. Gurkaynak, N. Felber, H. Kaeslin, and W. Fichtner. *Area, throughput and security considerations for AES crypto-ASICs.* Research in Microelectronics and Electronics, Volume 2, 25-28 July, 2005.

[14] Lawrence E. Bassham III. *The Advanced Encryption Standard Algorithm Validation Suite.* National Institute of Standards and Technology, 2002.

[15] Michael Keating, David Flynn, Robert Aitken, Alan Gibbons, and Kaijian Shi. *Low Power Methodology Manual for System-On-Chip Design.* Springer, 2007.

[16] MooSeop Kim, Juhan Kim, and Yongje Choi. *Low Power Circuit Architecture of AES Crypto Module for Wireless Sensor Network.* Proceedings of World Academy of Science, Engineering and Technology, Volume 8, 2005.

[17] Yanbing Li and Jörg Henkel. *A framework for estimation and minimizing energy dissipation of embedded HW/SW systems.* Proceedings of the 35th annual conference on Design automation, 1998.

[18] Sumio Morioka and Akashi Satoh. *An Optimized S-Box Circuit Architecture for Low Power AES Design.* CHES, 2002.

[19] Edwin NC Mui. *Practical Implementation of Rijndael S-Box Using Combinational Logic.*

[20] Farid N. Najm. *Power Estimation Techniques for Integrated Circuits.* IEEE/ACM International Conference on Computer Aided Design, 492-499, 1995.

[21] E. J. Nowak. *Maintaining the benefits of CMOS scaling when scaling bogs down.* International Business Machines Corporation, 2002.

[22] National Institute of Standards and Technology. *Fips-197: Advanced Encryption Standard*, 2001.

[23] Greg Osborn. *Embedded Microcontrollers and processor design.* Prentice Hall, 2009.

[24] Massoud Pedram and Jan Rabaey. *Power Aware Design Methodologies.* Kluwer Academic Publishers, 2002.

[25] Miodrag Potkonjak, Mani B. Srivastava, and Anantha Chandrakasan. *Efficient Substitution of Multiple Constant Multiplications by Shifts and Additions using Iterative Pairwise Matching.* Association for Computing Machinery, 1994.

[26] Jeffry T. Russell and Margarida F. Jacome. *Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors.* International Conference on Computer Design: VLSI in Computers and Processors, 1998.

[27] Shyam Sadasivan. *An Introduction to the ARM Cortex-M3 Processor.* ARM, 2006.

[28] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. *A Compact Rijndael Hardware Architecture with S-Box Optimization, in Advances in Cryptology — ASIACRYPT 2001.* Springer Berlin / Heidelberg, 2001.

[29] Amelia Shen, Abhijit Ghosh, Srinivas Devadas, and Kurt Keutzer. *On average power dissipation and random pattern testability of CMOS combinational logic networks.* IEEE Computer Society Press, 1992.

[30] William Stallings. *Operating systems Internals and design principles, 5th edition.* Prentice Hall, 2005.

[31] STMicroelectronics. *STM32F101x6 data sheet.* http://www.st.com/stonline/products/literature/ds/15058.pdf, accessed 15.05.09.

[32] Chih-Pin Su, Tsung-Fu Lin, Chih-Tsun Huang, and Cheng-Wen Wu. *A High-Throughput Low-Cost AES Processor.* IEEE Communications Magazine, 2003.

[33] Ioan Susnea and Marian Mitescu. *Microcontrollers in practice.* Springer, 2005.

[34] Synopsys. *Power Compiler User Guide, version Y-2006.06.* 2006.

[35] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. *Power Analysis of Embedded Software: A First Step Towards Software Power Minimization.* IEEE Transactions On Very Large Scale Integreation (VLSI) Sysytems, VOL. 2, NO. 4, 1994.

[36] Wikipedia. *Advanced Encryption Standard.* http://en.wikipedia.org/wiki/Advanced_Encryption_Standard, accessed 15.05.09.

[37] Wikipedia. *Block cipher modes of operation.* http://en.wikipedia.org/wiki/Block_cipher_modes_of_operation, accessed 15.05.09.

[38] Øivind Ekelund. *Low Energy Cryptographic Hardware - Advanced Encryption Standard.* Project report, NTNU, 2008.

Appendix A

Matrices

Matrices used in the Sbox

Inverse affine + mapping to $GF(2^4)$

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{A.1}$$

Mapping to $GF(2^8)$ + affine

$$\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{A.2}$$

Matrices used in MixColumns

$M^{-1} - M$

$$\begin{bmatrix} \{0C\} & \{08\} & \{0C\} & \{08\} \\ \{08\} & \{0C\} & \{08\} & \{0C\} \\ \{0C\} & \{08\} & \{0C\} & \{08\} \\ \{08\} & \{0C\} & \{08\} & \{0C\} \end{bmatrix} \tag{A.3}$$

$(M^{-1})^2$

$$\begin{bmatrix} \{05\} & \{00\} & \{04\} & \{00\} \\ \{00\} & \{05\} & \{00\} & \{04\} \\ \{04\} & \{00\} & \{05\} & \{00\} \\ \{00\} & \{04\} & \{00\} & \{05\} \end{bmatrix} \tag{A.4}$$

Appendix B

Tables and Figures

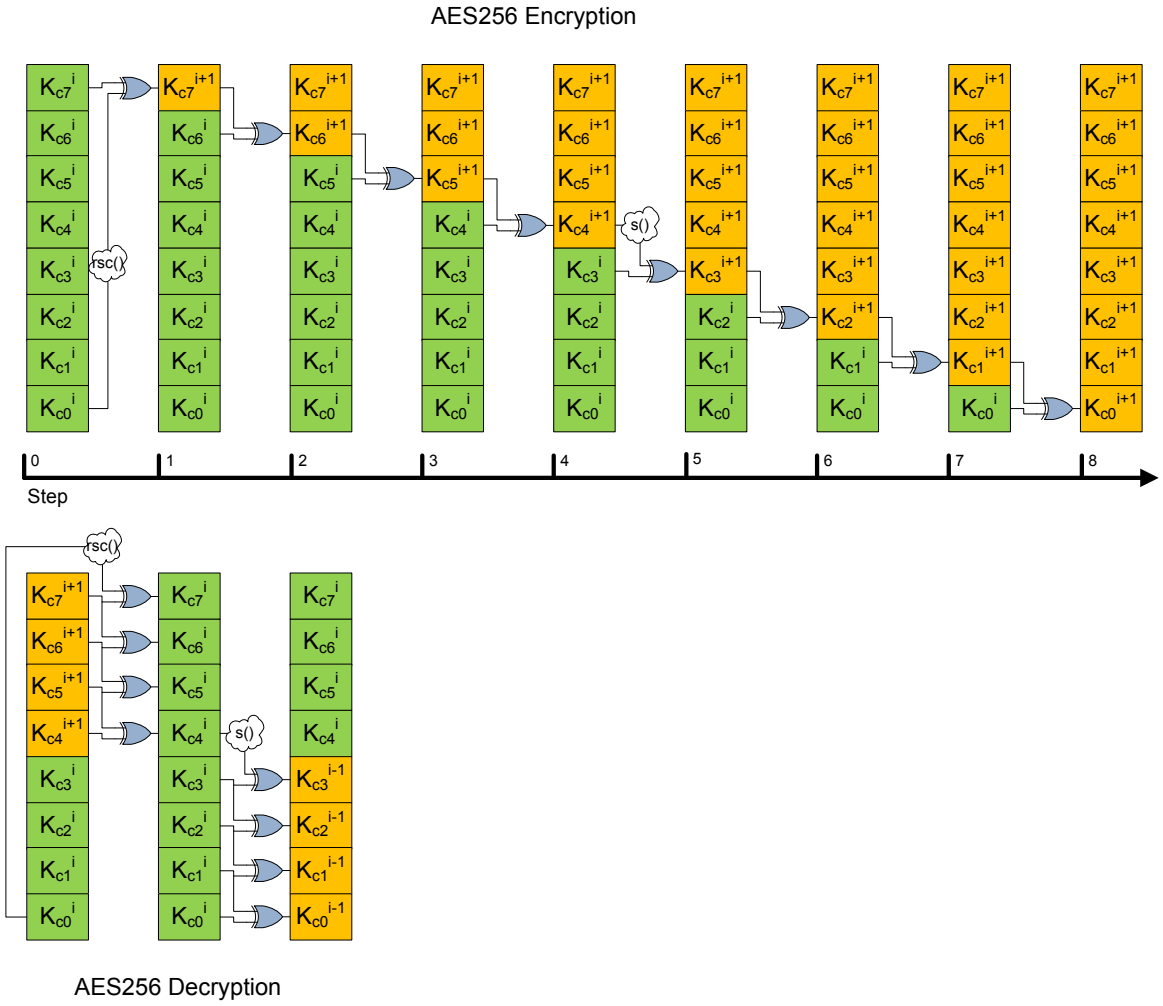

Figure B.1: Key expansion, AES256. $s()$ substitutes all bytes in a word using the Sboxes

State	Description
IDLE	Idle state, wait for start command
ENC128_INIT_RND	Initial round of AES128 encryption. Addroundkey and SubBytes are performed
ENC128_SHROW	Shiftrows is performed, Sbox is used in keyexpansion. KEY3 is expanded
ENC128_RND	Main rounds of AES128 encryption. MixColumns, Addroundkey and SubBytes are performed. KEY2 - KEY0 are expanded
ENC128_FINAL	Final round of AES128 encryption. Addroundkey is performed. KEY2 - KEY0 are expanded
DEC128_INIT_RND	Initial round of AES128 decryption, Addroundkey and inverse SubBytes are performed. KEY0 - KEY2 are expanded.
DEC128_SHROW	Inverse Shiftrows is performed, Sbox used in keyexpansion. KEY3 is expanded
DEC128_RND	Main rounds of AES128 decryption. Addroundkey, inverse MixColumns and inverse SubBytes are performed. KEY0 - KEY2 are expanded
DEC128_FINAL	Final round of AES128 decryption. Addroundkey is performed
ENC256_INIT_RND	Initial round of AES256 encryption. Addroundkey and SubBytes are performed
ENC256_SHROW1	Shiftrows is performed, Sbox used in keyexpansion. KEY7 is expanded
ENC256_RND1	Part of the main round. MixColumns, Addroundkey and SubBytes are performed. KEY6 - KEY4 are expanded
ENC256_SHROW2	Shiftrows is performed, Sbox used in keyexpansion. KEY3 is expanded
ENC256_RND2	Part of the main round. MixColumns, Addroundkey and SubBytes are performed. KEY2 - KEY0 are expanded
ENC256_FINAL	Final round of AES256 encryption. Addroundkey is performed
DEC256_INIT_RND	Initial round of AES256 decryption. Addroundkey and inverse SubBytes are performed
DEC256_SHROW1	Inverse Shiftrows is performed, Sbox used in keyexpansion. KEY4 - KEY7 are expanded
DEC256_RND1	Part of the main round. Addroundkey, inverse MixColumns and inverse SubBytes are performed
DEC256_SHROW2	Inverse Shiftrows is performed, Sbox used in keyexpansion. KEY0 - KEY3 are expanded
DEC256_RND2	Part of the main round. Addroundkey, inverse MixColumns and inverse SubBytes are performed
DEC256_FINAL	Final round of AES256 decryption. Addroundkey is performed

Table B.1: Description of states in FSM.
KEY7 - KEY0 represent the different words in the key register

Appendix C

Numerical Strength Reduction

Matrix multiplication used in MixColumns:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \{02\} & \{03\} & \{01\} & \{01\} \\ \{01\} & \{02\} & \{03\} & \{01\} \\ \{01\} & \{01\} & \{02\} & \{03\} \\ \{03\} & \{01\} & \{01\} & \{02\} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{C.1}$$

y can be computed using the following equations:

$$\begin{aligned} C_{01} &= (\{02\} \times x_1) + x_2 + x_3 \\ C_{23} &= x_0 + x_1 + (\{02\} \times x_3) \\ C_{03} &= \{02\} \times x_0 \\ C_{12} &= \{02\} \times x_2 \\ y_0 &= C_{01} + C_{03} + x_1 \\ y_1 &= C_{01} + C_{12} + x_0 \\ y_2 &= C_{23} + C_{12} + x_3 \\ y_3 &= C_{23} + C_{03} + x_2 \end{aligned} \tag{C.2}$$

Straightforward computation requires eight multiplications and sixteen additions, numerical strength reduction reduces this to four multiplications and twelve additions.

Matrix multiplication used in InvMixColumns:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} \{05\} & \{00\} & \{04\} & \{00\} \\ \{00\} & \{05\} & \{00\} & \{04\} \\ \{04\} & \{00\} & \{05\} & \{00\} \\ \{00\} & \{04\} & \{00\} & \{05\} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{C.3}$$

y can be computed using the following equations:

$$\begin{aligned} C_{02} &= \{04\} \times x_0 + \{04\} \times x_2 = \{04\} \times (x_0 + x_2) \\ C_{13} &= \{04\} \times x_1 + \{04\} \times x_3 = \{04\} \times (x_1 + x_3) \\ y_0 &= C_{02} + x_0 \\ y_1 &= C_{13} + x_1 \\ y_2 &= C_{02} + x_2 \\ y_3 &= C_{13} + x_3 \end{aligned} \tag{C.4}$$

Straightforward computation requires eight multiplications and four additions, numerical strength reduction reduces this to two multiplications and six additions.

Appendix D

Code

D.1 C code

```
1  ////////////////////////////////////////////////////////////
2  // Filename:     aes09_2.h
3  // Date:         22/05/09
4  // Author:       Oivind Ekelund
5  // Description: Header file for aes09_2.c
6  ////////////////////////////////////////////////////////////
7
8  #ifndef AES09
9  #define AES09
10
11 #define UINT32 unsigned int
12 #define BYTE unsigned char
13
14 //Macros for isolating bytes in a word
15 #define b3(x) ((x & 0xff000000)>>24)
16 #define b2(x) ((x & 0x00ff0000)>>16)
17 #define b1(x) ((x & 0x0000ff00)>>8)
18 #define b0(x) (x  & 0x000000ff)
19
20 void expand_key_trans(UINT32* rkc, UINT32* key, int aes256);
21 void encrypt(UINT32* rkc, UINT32* data, int aes256);
22 void decrypt(UINT32* rkc, UINT32* data, int aes256);
23
24 void shrow_subbytes(UINT32* data);
25 void invshrow_invsubbytes(UINT32* data);
26 void InvMixColumns(UINT32* data);
27 void MixColumns(UINT32* data);
28
29 #endif
```

```
#include <stdlib.h>
#include <stdio.h>
#include "aes09_2.h"

//Sbox, forward
const BYTE sbox[256] = {
  //0     1     2     3     4     5     6     7     8     9     A     B     C     D     E     F
  0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76, //0
  0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0, //1
  0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15, //2
  0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75, //3
  0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84, //4
  0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf, //5
  0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8, //6
  0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2, //7
  0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73, //8
  0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb, //9
  0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79, //A
  0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08, //B
  0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a, //C
  0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e, //D
  0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf, //E
  0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16  //F
};

//Sbox, inverse
const BYTE rsbox[256] = {
  0x52, 0x09, 0x6a, 0xd5, 0x30, 0x36, 0xa5, 0x38, 0xbf, 0x40, 0xa3, 0x9e, 0x81, 0xf3, 0xd7, 0xfb,
  0x7c, 0xe3, 0x39, 0x82, 0x9b, 0x2f, 0xff, 0x87, 0x34, 0x8e, 0x43, 0x44, 0xc4, 0xde, 0xe9, 0xcb,
  0x54, 0x7b, 0x94, 0x32, 0xa6, 0xc2, 0x23, 0x3d, 0xee, 0x4c, 0x95, 0x0b, 0x42, 0xfa, 0xc3, 0x4e,
  0x08, 0x2e, 0xa1, 0x66, 0x28, 0xd9, 0x24, 0xb2, 0x76, 0x5b, 0xa2, 0x49, 0x6d, 0x8b, 0xd1, 0x25,
  0x72, 0xf8, 0xf6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xd4, 0xa4, 0x5c, 0xcc, 0x5d, 0x65, 0xb6, 0x92,
  0x6c, 0x70, 0x48, 0x50, 0xfd, 0xed, 0xb9, 0xda, 0x5e, 0x15, 0x46, 0x57, 0xa7, 0x8d, 0x9d, 0x84,
  0x90, 0xd8, 0xab, 0x00, 0x8c, 0xbc, 0xd3, 0x0a, 0xf7, 0xe4, 0x58, 0x05, 0xb8, 0xb3, 0x45, 0x06,
  0xd0, 0x2c, 0x1e, 0x8f, 0xca, 0x3f, 0x0f, 0x02, 0xc1, 0xaf, 0xbd, 0x03, 0x01, 0x13, 0x8a, 0x6b,
  0x3a, 0x91, 0x11, 0x41, 0x4f, 0x67, 0xdc, 0xea, 0x97, 0xf2, 0xcf, 0xce, 0xf0, 0xb4, 0xe6, 0x73,
  0x96, 0xac, 0x74, 0x22, 0xe7, 0xad, 0x35, 0x85, 0xe2, 0xf9, 0x37, 0xe8, 0x1c, 0x75, 0xdf, 0x6e,
  0x47, 0xf1, 0x1a, 0x71, 0x1d, 0x29, 0xc5, 0x89, 0x6f, 0xb7, 0x62, 0x0e, 0xaa, 0x18, 0xbe, 0x1b,
  0xfc, 0x56, 0x3e, 0x4b, 0xc6, 0xd2, 0x79, 0x20, 0x9a, 0xdb, 0xc0, 0xfe, 0x78, 0xcd, 0x5a, 0xf4,
  0x1f, 0xdd, 0xa8, 0x33, 0x88, 0x07, 0xc7, 0x31, 0xb1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xec, 0x5f,
  0x60, 0x51, 0x7f, 0xa9, 0x19, 0xb5, 0x4a, 0x0d, 0x2d, 0xe5, 0x7a, 0x9f, 0x93, 0xc9, 0x9c, 0xef,
  0xa0, 0xe0, 0x3b, 0x4d, 0xae, 0x2a, 0xf5, 0xb0, 0xc8, 0xeb, 0xbb, 0x3c, 0x83, 0x53, 0x99, 0x61,
  0x17, 0x2b, 0x04, 0x7e, 0xba, 0x77, 0xd6, 0x26, 0xe1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0c, 0x7d
};

//Roundconstants
const BYTE rcon[10] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36};

//Key expansion for transposed keys
void expand_key_trans(UINT32* rkc, UINT32* key, int aes256){
  int i, Nb_Nr1, Nk;
  int rconst=0;

  if (aes256) {Nb_Nr1 = 60; Nk = 8;}
  else        {Nb_Nr1 = 44; Nk = 4;}

  //First key is the key itself
  for (i=0; i < Nk; i++){
    rkc[i] = key[i];
  }

  //Remaining roundkeys
  for (i=Nk; i < Nb_Nr1; i++){
    if(i%Nk==0){
      rkc[i]  = rkc[i - Nk] ^ (sbox[b0(rkc[i - Nk + 1])]  ^ rcon[rconst++])<<24;
    }
    else if (i%Nk==3){
      rkc[i]  = rkc[i - Nk] ^ (sbox[b0(rkc[i - Nk - 3])])<<24;
    }
    else{
      rkc[i]  = rkc[i - Nk] ^ (sbox[b0(rkc[i - Nk + 1])])<<24;
    }
    rkc[i] ^= (rkc[i] & 0xff000000)>>8;
    rkc[i] ^= (rkc[i] & 0x00ff0000)>>8;
    rkc[i] ^= (rkc[i] & 0x0000ff00)>>8;
  }
}


//Combined shiftrows and subbytes
void shrow_subbytes(UINT32* data){
  UINT32 d0t, d1t, d2t, d3t;
  d0t = data[0];
  d1t = data[1];
  d2t = data[2];
  d3t = data[3];

  data[0] = (sbox[b3(d0t)]<<24) | (sbox[b2(d0t)]<<16) | (sbox[b1(d0t)]<<8) | sbox[b0(d0t)];
  data[1] = (sbox[b2(d1t)]<<24) | (sbox[b1(d1t)]<<16) | (sbox[b0(d1t)]<<8) | sbox[b3(d1t)];
  data[2] = (sbox[b1(d2t)]<<24) | (sbox[b0(d2t)]<<16) | (sbox[b3(d2t)]<<8) | sbox[b2(d2t)];
```

```
91    data[3] = (sbox[b0(d3t)]<<24) | (sbox[b3(d3t)]<<16) | (sbox[b2(d3t)]<<8) | sbox[b1(d3t)];
92  }
93
94
95  //Combined invshiftrows and subbytes
96  void invshrow_invsubbytes(UINT32* data){
97    UINT32 d0t, d1t, d2t, d3t;
98    d0t = data[0];
99    d1t = data[1];
100   d2t = data[2];
101   d3t = data[3];
102
103   data[0] = (rsbox[b3(d0t)]<<24) | (rsbox[b2(d0t)]<<16) | (rsbox[b1(d0t)]<<8) | rsbox[b0(d0t)];
104   data[1] = (rsbox[b0(d1t)]<<24) | (rsbox[b3(d1t)]<<16) | (rsbox[b2(d1t)]<<8) | rsbox[b1(d1t)];
105   data[2] = (rsbox[b1(d2t)]<<24) | (rsbox[b0(d2t)]<<16) | (rsbox[b3(d2t)]<<8) | rsbox[b2(d2t)];
106   data[3] = (rsbox[b2(d3t)]<<24) | (rsbox[b1(d3t)]<<16) | (rsbox[b0(d3t)]<<8) | rsbox[b3(d3t)];
107 }
108
109 //Multiplication with 2 in GF(256), byte by byte
110 UINT32 wxtime(UINT32 x){
111   UINT32 tmp;
112   tmp  = x & 0x80808080;
113   tmp |= tmp>>1;
114   tmp |= tmp>>2;
115   tmp |= tmp>>4;
116   tmp ^= tmp<<1;
117   tmp &= 0x1b1b1b1b;
118   return x<<1 ^ tmp;
119 }
120
121 //Multiplication in GF(256), 4 MSBs of y needs to be 0
122 UINT32 wmult(UINT32 x, UINT32 y){
123   UINT32 tmp1, tmp2, tmp3, tmp = 0;
124
125   tmp1 = wxtime(x);
126   tmp2 = wxtime(tmp1);
127   tmp3 = wxtime(tmp2);
128
129   if(y>>0 & 1) tmp = x;
130   if(y>>1 & 1) tmp ^= tmp1;
131   if(y>>2 & 1) tmp ^= tmp2;
132   if(y>>3 & 1) tmp ^= tmp3;
133   return tmp;
134 }
135
136 //MixColumns
137 void MixColumns(UINT32* data){
138   /*
139   //MixColumns with NSR
140   UINT32 d0t, d1t, d2t, d3t, tmp1, tmp2, tmp3, tmp4;
141   d0t = data[0];
142   d1t = data[1];
143   d2t = data[2];
144   d3t = data[3];
145
146   tmp1 = wxtime(d1t) ^ d2t ^ d3t;
147   tmp2 = wxtime(d3t) ^ d0t ^ d1t;
148   tmp3 = wxtime(d0t);
149   tmp4 = wxtime(d2t);
150
151   data[0] = tmp1 ^ tmp3 ^ d1t;
152   data[1] = tmp1 ^ tmp4 ^ d0t;
153   data[2] = tmp2 ^ tmp4 ^ d3t;
154   data[3] = tmp2 ^ tmp3 ^ d2t;*/
155
156   //Starightforward MixColumns
157   UINT32 d0t, d1t, d2t, d3t;
158   d0t = data[0];
159   d1t = data[1];
160   d2t = data[2];
161   d3t = data[3];
162
163   data[0] = wmult(d0t, 0x02) ^ wmult(d1t, 0x03) ^ d2t ^ d3t;
164   data[1] = d0t ^ wmult(d1t, 0x02) ^ wmult(d2t, 0x03) ^ d3t;
165   data[2] = d0t ^ d1t ^ wmult(d2t, 0x02) ^ wmult(d3t, 0x03);
166   data[3] = wmult(d0t, 0x03) ^ d1t ^ d2t ^ wmult(d3t, 0x02);
167 }
168
169
170 //inverse MixColumns
171 void InvMixColumns(UINT32* data){
172   /*
173   //MixColumns with NSR
174   UINT32 d0t, d1t, d2t, d3t, tmp1, tmp2, tmp3, tmp4;
175   d0t = data[0];
176   d1t = data[1];
177   d2t = data[2];
178   d3t = data[3];
179
180   tmp1 = wxtime(wxtime((d0t ^ d2t)));
```

```
    tmp2 = wxtime(wxtime((d1t ^ d3t)));

    d0t ^= tmp1;
    d1t ^= tmp2;
    d2t ^= tmp1;
    d3t ^= tmp2;

    tmp1 = wxtime(d1t) ^ d2t ^ d3t;
    tmp2 = wxtime(d3t) ^ d0t ^ d1t;
    tmp3 = wxtime(d0t);
    tmp4 = wxtime(d2t);

    data[0] = tmp1 ^ tmp3 ^ d1t;
    data[1] = tmp1 ^ tmp4 ^ d0t;
    data[2] = tmp2 ^ tmp4 ^ d3t;
    data[3] = tmp2 ^ tmp3 ^ d2t;*/

    //Straightforward InvMixColumns
    UINT32 d0t, d1t, d2t, d3t;
    d0t = data[0];
    d1t = data[1];
    d2t = data[2];
    d3t = data[3];

    data[0] = wmult(d0t, 0x0e) ^ wmult(d1t, 0x0b) ^ wmult(d2t, 0x0d) ^ wmult(d3t, 0x09);
    data[1] = wmult(d0t, 0x09) ^ wmult(d1t, 0x0e) ^ wmult(d2t, 0x0b) ^ wmult(d3t, 0x0d);
    data[2] = wmult(d0t, 0x0d) ^ wmult(d1t, 0x09) ^ wmult(d2t, 0x0e) ^ wmult(d3t, 0x0b);
    data[3] = wmult(d0t, 0x0b) ^ wmult(d1t, 0x0d) ^ wmult(d2t, 0x09) ^ wmult(d3t, 0x0e);
}

//Encryption, rkc is roundkeys
void encrypt(UINT32* rkc, UINT32* data, int aes256){
    int i, Nr;

    if (aes256) {Nr = 14;}
    else        {Nr = 10;}

    //First round:
    data[0] ^= *(rkc ++);
    data[1] ^= *(rkc ++);
    data[2] ^= *(rkc ++);
    data[3] ^= *(rkc ++);

    //Remaining rounds-1
    for(i=1; i<Nr; i++){
      shrow_subbytes(data);

      MixColumns(data);

      //Addroundkey
      data[0] ^= *(rkc ++);
      data[1] ^= *(rkc ++);
      data[2] ^= *(rkc ++);
      data[3] ^= *(rkc ++);
    }

    //Final round
    shrow_subbytes(data);

    //Addroundkey
    data[0] ^= *(rkc ++);
    data[1] ^= *(rkc ++);
    data[2] ^= *(rkc ++);
    data[3] ^= *(rkc ++);
}



//Decryption, rkc is roundkeys
void decrypt(UINT32* rkc, UINT32* data, int aes256){
    int i, Nr;
    UINT32* pKey;

    if (aes256) {Nr = 14;}
    else        {Nr = 10;}
    pKey = rkc + 4*(Nr + 1) - 1;

    //First round:
    data[3] ^= *(pKey --);
    data[2] ^= *(pKey --);
    data[1] ^= *(pKey --);
    data[0] ^= *(pKey --);

    //Remaining rounds-1
    for(i=Nr-1; i>0; i--){
      invshrow_invsubbytes(data);

      //Addroundkey
      data[3] ^= *(pKey --);
      data[2] ^= *(pKey --);
```

```
271         data[1] ^= *(pKey --);
272         data[0] ^= *(pKey --);
273 
274         InvMixColumns(data);
275       }
276 
277       //Final round
278       invshrow_invsubbytes(data);
279 
280       //Addroundkey
281       data[3] ^= *(pKey --);
282       data[2] ^= *(pKey --);
283       data[1] ^= *(pKey --);
284       data[0] ^= *(pKey --);
285     }
```

D.2 HDL testbenches

```
1  ----------------------------------------------------------------------------
2  --
3  -- Title       : aes_tb
4  -- Creator     : oivind Ekelund
5  -- Date        : 25.03.09
6  --
7  -- Description : Testbench for power estimation
8  --
9  ----------------------------------------------------------------------------
10
11 library ieee;
12 use ieee.std_logic_1164.all;
13 use ieee.std_logic_arith.all;
14
15 entity aes_tb is
16 end aes_tb;
17
18
19 architecture testbench of aes_tb is
20
21 constant PERIOD: time := 31 ns;
22 constant NR_ROUNDS: integer := 100;
23
24
25 signal clk, clk_en, reset_n, aes256, start, decrypt, stop, keybufen : std_logic := '0';
26 signal done, running : std_logic;
27
28 component aes_toplevel
29    port( clk, clk_en, reset_n, aes256, start, decrypt, stop, keybufen : in
30          std_logic;  done, running : out std_logic);
31 end component;
32
33
34 begin
35
36 --AES toplevel module
37 u_aes_toplevel: aes_toplevel
38    port map(
39       clk       => clk,
40       clk_en    => clk_en,
41       reset_n   => reset_n,
42       aes256    => aes256,
43       start     => start,
44       decrypt   => decrypt,
45       stop      => stop,
46       keybufen  => keybufen,
47       done      => done,
48       running   => running
49    );
50
51
52 clock: process is
53 begin
54     wait for PERIOD/2;
55     clk <= '1';
56     wait for PERIOD/2;
57     clk <= '0';
58 end process clock;
59
60 reset: process is
61 begin
62     wait for PERIOD;
63     reset_n <= '1';
64     wait;
65 end process reset;
66
67 stimuli: process is
68 begin
69     clk_en   <= '1';
70
71     aes256   <= '0';
72     decrypt  <= '0';
73     keybufen <= '0';
74
75     --Wait for reset
76     wait for 2*PERIOD;
77
78     --Main loop, doing NR_ROUNDS encryptions
79     for i in 0 to NR_ROUNDS-1 loop
80
81        --Start
82        --start <= '1';
83        wait for PERIOD;
84        start <= '0';
85
86        --Wait for completion
87        wait until done = '1';
```

```
88          wait for PERIOD;
89
90      end loop;
91
92      wait;
93  end process stimuli;
94
95  end testbench;
96
97
98  CONFIGURATION aes_config OF aes_tb IS
99      for testbench
100             for u_aes_toplevel : aes_toplevel
101                 use entity work.aes_toplevel(syn_verilog);
102         end for;
103     end for;
104 END aes_config;
```

```
1  // ******************************************************************************
2  //
3  // Title        : aes_testbench
4  // Creator      : Oivind Ekelund
5  // Date         : 23.05.09
6  //
7  // Description : Verification of AES128 en- and decryption
8  //               Initial data and key is 0
9  //               Encryption is performed NR_ROUND times
10 //               Decryption is performed NR_ROUND times
11 // ******************************************************************************
12
13
14 module aes_testbench();
15
16    parameter PERIOD = 10;
17    parameter NR_ROUNDS = 100;
18
19    reg clk, clk_en, reset_n, aes256, keybufen, decrypt, start, stop;
20    wire done, running;
21
22    reg [127:0] goldendataenc[0 : NR_ROUNDS];
23    reg [127:0] goldenkeyenc[0 : NR_ROUNDS];
24
25    integer i=0;
26
27    //Clock
28    always #(PERIOD/2) begin
29       clk = 1;
30       #(PERIOD/2)
31       clk = 0;
32    end
33
34    //Reset and clock enable
35    initial begin
36       $readmemh("goldendataenc.txt", goldendataenc);
37       $readmemh("goldenkeyenc.txt", goldenkeyenc);
38       clk_en = 1;
39       #PERIOD
40          reset_n = 0;
41       #PERIOD
42          reset_n = 1;
43    end
44
45    //Teststimuli
46    initial begin
47       start        = 0;
48       aes256       = 0;
49       decrypt      = 0;
50       keybufen     = 0;
51       stop         = 0;
52    end
53
54    always @ (negedge done) begin
55       if(!decrypt) begin
56          //Start verification AES128 encryption
57          if (i==0) begin //First round of simulation
58             i = i+1;
59             #(3*PERIOD)
60             start = 1;
61             #(2*PERIOD)
62             start = 0;
63          end
64          else if(i<(NR_ROUNDS) && i>0) begin //Check results
65             if(u_aes_toplevel.data_in != goldendataenc[i])
66               $display("***ERROR***, data_is: %h, should_be: %h", u_aes_toplevel.data_in, goldendataenc[i]);
67             if(u_aes_toplevel.key_in != goldenkeyenc[i])
68               $display("***ERROR***, key_is: %h, should_be: %h", u_aes_toplevel.key_in, goldenkeyenc[i]);
69
70             #(1*PERIOD)
71             start = 1;
72             #(2*PERIOD)
73             start = 0;
74             i = i+1;
75          end
76          else begin //Check last encryption result
77             if(u_aes_toplevel.data_in != goldendataenc[i])
78               $display("***ERROR***, data_is: %h, should_be: %h", u_aes_toplevel.data_in, goldendataenc[i]);
79             if(u_aes_toplevel.key_in != goldenkeyenc[i])
80               $display("***ERROR***, key_is: %h, should_be: %h", u_aes_toplevel.key_in, goldenkeyenc[i]);
81
82             $display("***VERIFICATION, AES128 ENCRYPTION COMPLETE***");
83
84             //Start verification AES128 decryption
85             decrypt = 1;
86             i = i-1;
87             #(1*PERIOD)
88             start = 1;
89             #(2*PERIOD)
90             start = 0;
```

```
91              end
92
93          end
94
95          else if(decrypt) begin
96              if (i==0) begin //Check last result
97                  if(u_aes_toplevel.data_in != goldendataenc[i])
98                    $display("***ERROR***, data_is: %h, should_be: %h", u_aes_toplevel.data_in, goldendataenc[i]);
99                  if(u_aes_toplevel.key_in != goldenkeyenc[i])
100                   $display("***ERROR***, key_is: %h, should_be: %h", u_aes_toplevel.key_in, goldenkeyenc[i]);
101
102                 $display("***VERIFICATION, AES128 DECRYPTION COMPLETE***");
103
104             end
105             else if(i<(NR_ROUNDS) && i>0) begin //Check results
106                 if(u_aes_toplevel.data_in != goldendataenc[i])
107                   $display("***ERROR***, data_is: %h, should_be: %h", u_aes_toplevel.data_in, goldendataenc[i]);
108                 if(u_aes_toplevel.key_in != goldenkeyenc[i])
109                   $display("***ERROR***, key_is: %h, should_be: %h", u_aes_toplevel.key_in, goldenkeyenc[i]);
110                 #(1*PERIOD)
111                 start = 1;
112                 #(2*PERIOD)
113                 start = 0;
114                 i = i-1;
115             end
116         end
117     end
118
119     //DUT
120     aes_toplevel u_aes_toplevel(
121                                 .clk        (clk),
122                                 .reset_n    (reset_n),
123                                 .clk_en     (clk_en),
124                                 .start      (start),
125                                 .stop       (stop),
126                                 .decrypt    (decrypt),
127                                 .keybufen   (keybufen),
128                                 .aes256     (aes256),
129
130                                 .done       (done),
131                                 .running    (running)
132                                 );
133
134 endmodule
```

D.3 Synthesis- and simulation scripts

```
1  ##############################################################
2  # Filename:     synth.tcl
3  # Date:         22/05/09
4  # Author:       Oivind Ekelund
5  # Description: Synthesis script optimizing for low power
6  #               with a 32MHz target frequency.
7  #               Clock gating is also performed
8  ##############################################################
9
10 #Analyze the desing priror to running this script
11
12 #Elaborate and link
13 elaborate aes_toplevel -architecture verilog -library DEFAULT
14 link
15 insert_clock_gating -global
16
17 #Optimize for low power
18 set_max_dynamic_power  0
19 set_max_total_power 0 ""
20
21 #Specify clock
22 create_clock clk -name clock -period 31
23
24 #Compile
25 uplevel #0 compile -map_effort high -area_effort high -incremental_mapping
26
27 #Change names
28 change_names -rules vhdl -hierarchy
29
30 #Check design
31 check_design
```

```
1  ##############################################################
2  # Filename:     runsim.do
3  # Date:         22/05/09
4  # Author:       Oivind Ekelund
5  # Description: Simulation script, running the testbench
6  #               configuration aes_config for 170 us, logging
7  #               activity in u_aes_toplevel and reporting
8  #               to a.vcd.
9  ##############################################################
10
11 vsim -noglitch -t fs -vital2.2b work.aes_config
12 vcd file a.vcd
13 vcd add -r sim:aes_tb/u_aes_toplevel/*
14 run 170000 ns
15 vcd checkpoint
16 quit -sim
```