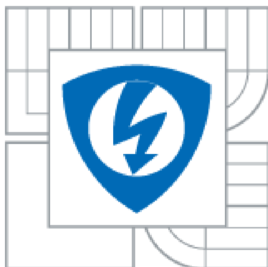
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

# IMPLEMENTACE SYMETRICKÉ BLOKOVÉ ŠIFRY AES NA MODERNÍCH PROCESORECH

IMPLEMENTATION OF SYMMETRIC BLOCK CIPHER AES IN MODERN PROCESSORS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

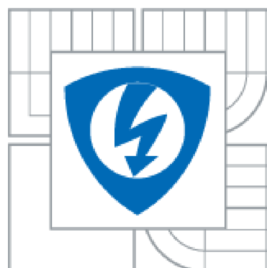
AUTOR PRÁCE Bc. MARTIN ŠKODA

AUTHOR

VEDOUCÍ PRÁCE Ing. ONDŘEJ RÁŠO, Ph.D.

SUPERVISOR

BRNO 2014



**VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ**

**Fakulta elektrotechniky a komunikačních technologií**

**Ústav telekomunikací**

# Diplomová práce

magisterský navazující studijní obor

**Telekomunikační a informační technika**

| | | | |
|---|---|---|---|
| ***Student:*** | Bc. Martin Škoda | ***ID:*** | 125658 |
| ***Ročník*** | 2 | ***Akademický rok:*** | 2013/2014 |

**NÁZEV TÉMATU:**

**Implementace symetrické blokové šifry AES na moderních procesorech.**

**POKYNY PRO VYPRACOVÁNÍ:**

1) Popište princip symetrických blokových šifer vč. jejich režimů.
2) Proveďte rozbor nových instrukcí z instrukční sady i5, které lze použít pro implementaci šifry AES na osobních počítačích. Tuto rešerši doplňte o komentované příklady.
3) V jazyce symbolických instrukcí implementujte šifru AES pro velikost klíče 128, 192 a 256 bitů. Dále implementujte vybrané typy režimů této šifry. Zdrojové kódy opět doplňte detailními komentáři.
4) Vytvořte dynamickou knihovnu, která bude poskytovat funkce pro šifrování podle bodu č. 3.

**DOPORUČENÁ LITERATURA:**

[1] Intel® Advanced Encryption Standard (AES) Instructions Set, White Paper, Intel Mobility Group, 2012, Rev 3.01 , dostupné z www: http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/
[2] RÁŠO, O. Programování funkcí v jazyce symbolických adres podle STDCALL konvence. Elektrorevue - Internetový časopis (http://www.elektrorevue.cz), 2009, roč. 2009, č. 8, s. 1-4. ISSN: 1213- 1539.

| | | | |
|---|---|---|---|
| ***Termín zadání:*** | 10.2.2014 | ***Termín odevzdání:*** | 28.5.2014 |

***Vedoucí práce:*** Ing. Ondřej Rášo
***Konzultanti semestrální práce:***

**doc. Ing. Jiří Mišurec, CSc.**
Předseda oborové rady

**UPOZORNĚNÍ:**

Autor semestrální práce nesmí při vytváření semestrální práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

## Abstrakt

Hlavním cílem diplomové práce je využití nových instrukcí z instrukční sady Intel® Advanced Encryption Standard New Instructions (AES-NI), která je dostupná na procesorech s kódovým označením Westmere a novějších. V teoretické části jsou popsány symetrické blokové šifry a jejich operační módy. Šifra AES je popsána podrobně, zejména používané blokové transformace, expanze klíče a ekvivalentní inverzní šifra. Dále jsou popsány instrukce z instrukční sady AES-NI – je vysvětlena jejich funkčnost pomocí pseudokódů a jsou uvedeny příklady jejich použití. Následně je vytvořena dynamická knihovna, která implementuje šifru AES pro velikost klíče 128, 192 a 256 bitů a implementuje operační módy popsané v teoretické práci. Funkce knihovny jsou volány z prostředí Matlab pomocí skriptů a je ověřena jejich funkčnost porovnáním výsledků funkcí s testovacími vektory, které poskytuje v publikacích Národního institutu standardů a technologie.



## Klíčová slova

Symetrické blokové šifry, operační módy blokových šifer, Advanced Encryption Standard, Intel Advanced Encryption Standard New Instruction, jazyk symbolických instrukcí, dynamicky linkovaná knihovna.



## Abstrakt

The main aim of master's thesis is usage of new instructions from instruction set called Intel® Advanced Encryption Standard New Instructions (AES-NI), which is available on processors with code name Westmere and newer. In theoretical part, there are described symmetric block ciphers and their operational modes. Cipher AES is described in details, especially used block transformations, key expansion and equivalent inverse cipher. Next topic is description of instructions of AES-NI instruction set – their function is explained using pseudo-codes of instructions and there are examples of their usage in code. Further in work, dynamic-link library is created, which implements cipher AES with key sizes 128, 192 and 256 bites and implements operational modes described in theoretical part. Library functions are called from Matlab by scripts and their functionality is proved by checking test vectors values, which are provided in publications of National Institute of Standards and Technology.



## KeyWords

Symmetric block cipher, block cipher modes of operation, Advanced Encryption Standard, Intel Advanced Encryption Standard New Instruction, assembly language, dynamic-link library.

## Bibliografická citace práce:

ŠKODA, M. *Implementace symetrické blokové šifry AES na moderních procesorech.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2014. 87 s. Vedoucí semestrální práce Ing. Ondřej Rášo, Ph.D..

## Prohlášení

Prohlašuji, že svou semestrální práci na téma Implementace symetrické blokové šifry AES na moderních procesorech jsem vypracoval samostatně pod vedením vedoucího semestrální práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené semestrální práce dále prohlašuji, že v souvislosti s vytvořením této semestrální práce jsem neporušil(-a) autorská práva třetích osob, zejména jsem nezasáhl(-a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(-a) následku porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonu (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne ............... ...........................................

podpis autora

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Technicka 12, CZ-61600 Brno
Czech Republic
http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v tomto diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

(podpis autora)

## Obsah


**1 Úvod ..... 6**

**2 Symetrické šifry ..... 7**

2.1 Blokové šifry ..... 7
2.2 Operační módy blokových šifer ..... 8

**3 Advanced Encryption Standard ..... 15**

3.1 Algoritmus šifrování a dešifrování ..... 15
3.2 Expanze klíče ..... 17
3.3 Používané transformace ..... 17
3.4 Ekvivalentní inverzní šifra ..... 19

**4 Architektura Intel AES New Instruction ..... 21**

4.1 Formát dat ..... 21
4.2 Rundovní instrukce ..... 23
4.3 Instrukce pro generování klíčů ..... 26

**5 Implementace šifry Advanced Encryption Standard ..... 31**

5.1 Používané instrukce ..... 31
5.2 Expanze klíče ..... 34
5.3 Šifrování a dešifrování AES ..... 41
5.4 Operační módy AES ..... 42
5.5 Vytvoření knihovny ..... 50
5.6 Ukázky výsledků ..... 57

**6 Závěr ..... 66**

**7 Seznam použité literatury ..... 67**

**8 Seznam použitých zkratek ..... 68**

**9 Seznam příloh ..... 69**

# 1 Úvod

V roce 1997 americký Národní institut standardů a technologie (NIST) vydal požadavek na nový standard pokročilého šifrování (anglicky Advanced Encryption Standard), který by měl nahradit dosavadní algoritmus DES, resp. 3DES. Ze všech patnácti přihlášených návrhů byl v roce 2001 jako nejlepší ohodnocen a vybrán algoritmus Rijndael [rejndál] (název algoritmu vznikl přesmyčkou jmen autorů). Autoři jsou dva kryptografové z Belgie – Dr. Joan Daemen a Dr. Vincent Rijmen. V dnešní době se algoritmus Rijndael označuje jako Advanced Encryption Standard (AES). Dnes se AES používá například u šifrování dat pevných disků a pro zabezpečení bezdrátových sítí (Wi-Fi Protected Access II - WPA2).

Tato práce se zabývá implementací šifry AES pomocí instrukcí označovaných Intel® AES New Instruction. Jde o sadu instrukcí dostupnou s příchodem rodiny procesorů Intel® Core™ na 32 nm Intel® mikroarchitektůře s kódovým jménem Westmere. Jde o šest hardwarově podporovaných instrukcí, jejíchž účelem je poskytnout rychlé a bezpečné implementace šifry AES.

V teoretické části (body zadání 1 a 2) bude na začátku popsán princip symetrických šifer a jejich operačních režimů (módů). Hlavním cílem teoretické části je podrobněji popsat šifru AES a instrukce z instrukční sady Intel® AES New Instruction.

Zbylé body zadání (3 a 4) tvoří praktickou část, v jejímž rámci bude implementována šifra AES pro velikosti klíče 128, 192 a 256 bitů v jazyce symbolických instrukcí. Následně budou implementovány zadané operační režimy symetrických šifer (režim elektronické kódové knihy, řetězení bloků šifry, zpětné vazby ze šifry, zpětné vazby z výstupu a režim čítače). Posledním úkolem je vytvořit dynamickou knihovnu poskytující funkce implementující šifru AES. Funkce vytvořené knihovny budou volány a zkoušeny pomocí skriptů z prostředí Matlab.

# 2 Symetrické šifry

Schéma symetrických šifer, jak je popsáno v [1], se skládá z pěti součástí:

- **Čistý text** – původní srozumitelná zpráva, která je vstupem šifrovacího algoritmu.
- **Šifrovací algoritmus** – algoritmus provádějící transformace nad čistým textem.
- **Tajný klíč** – vstup šifrovacího algoritmu (šifrovací klíč) hodnotami nezávislí na čistém textu. Pro různé hodnoty tajného klíče je různý výstup šifrovacího algoritmu.
- **Šifrovaný text** – nesrozumitelná zpráva závislá pro daný šifrovací algoritmus na čistém textu a tajném klíči.
- **Dešifrovací algoritmus** – v podstatě šifrovací algoritmus, jehož transformace jdou v obráceném pořadí. Vstupem je šifrovaný text a tajný klíč (dešifrovací klíč), výstupem je čistý text.

Jak je uvedeno v [2], pokud jsou šifrovací a dešifrovací klíče stejné, nebo prakticky od sebe odvoditelné, musí být oba dva tajné (proto tajný klíč). Z hlediska utajení mají pak stejné tzv. symetrické postavení a proto se takové šifry nazývají symetrické. Symetrické šifry se dělí podle zpracovávání dat na:

- **proudové** – zpracovávají proud bitů,
- **blokové** – pracují s bloky pevné velikosti.

Příkladem proudových šifer je šifra RC4, která je popsaná v [1], část 1., kapitola 6.3. Příkladem blokových šifer jsou DES a AES. Šifra AES bude popsána v další kapitole. Pro více informací o DES viz [1], část 1., kapitola 3.

## 2.1 Blokové šifry

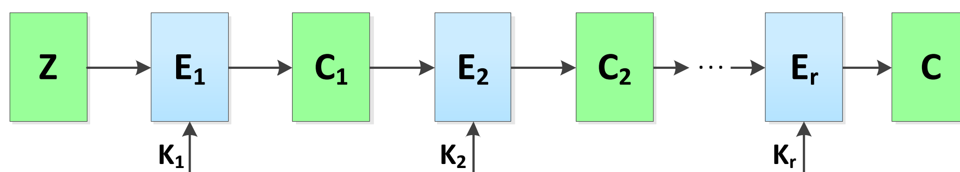
Blokové šifry pracují s bloky pevné velikosti. Autor v [2] uvádí, že: „*Hodnota výstupního bloku kryptogramu C je závislá na všech n bitech vstupního bloku Z (tzv. difuze) a na všech bitech klíče K (tzv. konfuze). Difuze a konfuze značně komplikují útočníkovi odhad hodnoty Z nebo K z hodnoty C a tím výrazně ztěžují kryptoanalýzu blokových šifer.*“

V praxi se podle [2] nejčastěji používají tyto blokové operace:

- **permutace** (prohození *bytů*),
- **substituce** (náhrada *bytů*),
- **aritmetické operace**.

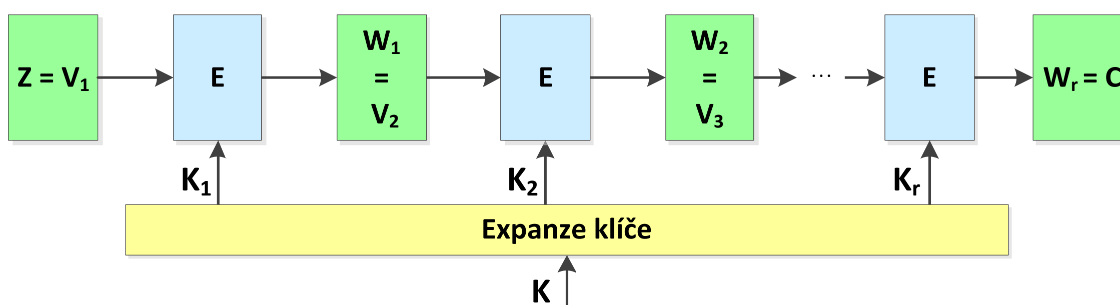
Speciálním případem permutace je rotace – ta se používá u AES. U substituce se nahrazují původní byty hodnotami z tabulky. U aritmetických operací se bloky *bitů* o délce *n* chápou jako čísla, se kterými se provádějí příslušné aritmetické operace a na získaný výsledek se aplikuje operace modulo $2^n$. Např. pro čtyřbitové sčítání 11 + 14 je $n = 4$. Pak můžeme psát $(11+14) \bmod 2^4 = 25 \bmod 16 = 9$. V binární formě $1011 + 1110 = 1001$.

Konstrukce blokových šifer vychází z konceptu kaskády šifer („*product cipher*“). Jde o zřetězení *r* šifer, kde výstupním blokem jedné šifry je vstupní blok následující šifry. Výstupní blok poslední šifry je zároveň výsledným kryptogramem.

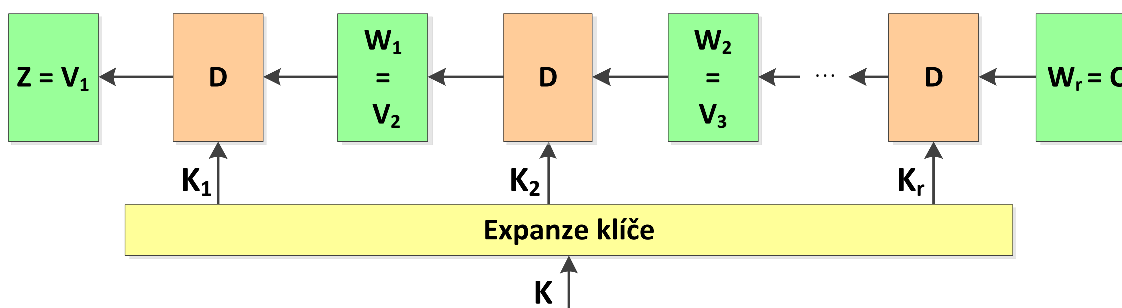



**Obr. 2.1: Kaskáda šifer. [2]**

Podle [2] se v praxi používá zřetězení *r* stejných šifer, tzv. iterovaná šifra (viz Obr. 2.1). V bloku expanze se z šifrovacího klíče *K* odvodí *r* dílčích klíčů ($K_1$ až $K_r$). Tyto klíče jsou šifrovacími klíči pro iterovanou (opakovaně použitou) transformaci *E*. Transformaci *E* se proto budeme v dalším nazývat jako iteraci („*round*" – runda). Iterace je zpravidla relativně jednoduchá šifra, ve které je vstupní blok bitů $V_i$ v závislosti na iteračním klíči $K_i$ transformován nějakou vhodnou kombinací blokových operací (zpravidla substitucí, permutaci a aritmetických operací) do podoby výstupního bloku $W_i$. Výstupem *r*-té iterace E je výstupní blok $W_r = C$, což je výsledný šifrovaný text.




**Obr. 2.2: Obecné schéma blokové šifry při šifrování. [2]**




**Obr. 2.3: Obecné schéma blokové šifry při dešifrování. [2]**

## 2.2 Operační módy blokových šifer

Pro využití blokových šifer v různých aplikacích byly zavedeny operační módy (režimy) blokových šifer. Operační módy vylepšují vlastnosti blokových šifer pro danou aplikaci. Operačních módů je pět:

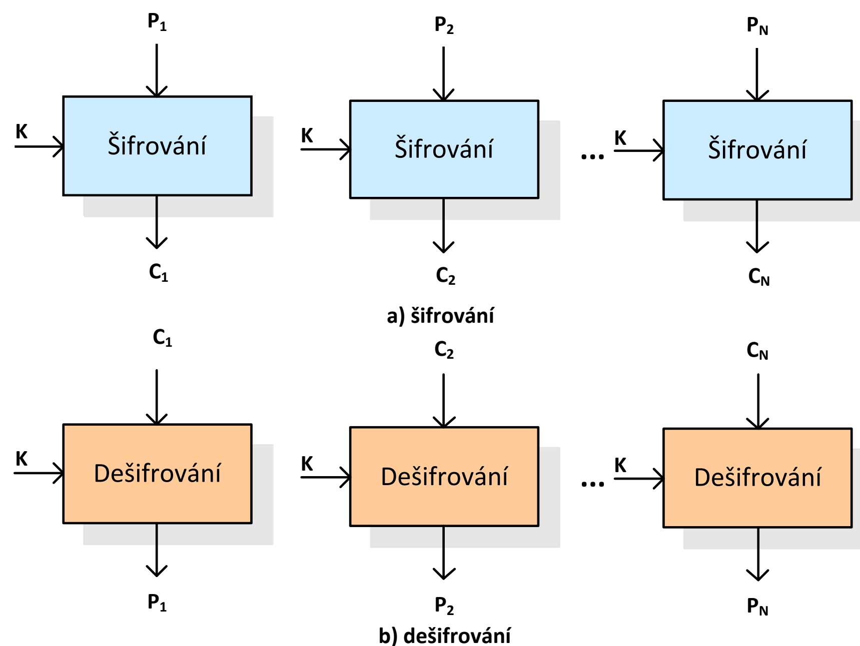
- Electronic Codebook – elektronická kódová kniha,
- Cipher Block Chaining – řetězení šifrovaného textu,
- Cipher Feedback – zpětná vazba ze šifrovaného textu,
- Output Feedback – zpětná vazba z výstupu,
- Counter – čítačový mód.

Tyto módy budou nyní popsány dle [1].

### 2.2.1 Electronic Codebook (ECB)

V tomto módu se jednotlivé bloky šifrují nezávisle na sobě. Každý blok je zašifrovaný stejným klíčem. Pro dva stejné čisté bloky je výstupem stejný šifrovaný blok. Proto označení kódová kniha – pro množinu všech čistých textů existuje množina jim odpovídajících šifrovaných textů.

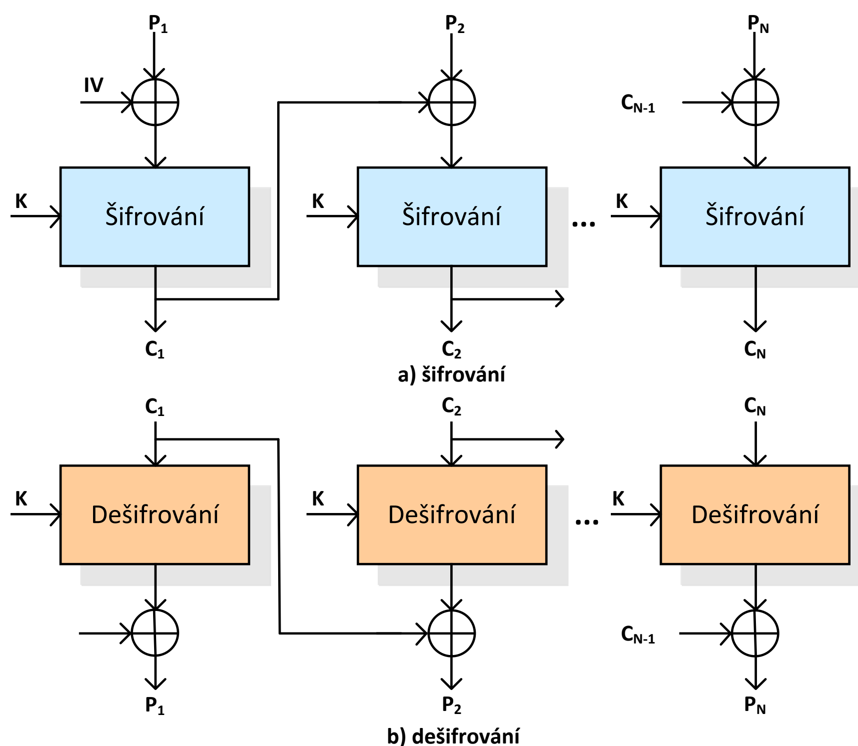
Mód ECB je vhodný pro šifrování krátkých dat, například šifrovacího klíče. Není bezpečný pro šifrování dlouhých zpráv, protože u zpráv s předdefinovaným formátem by útočník mohl získat sobě odpovídající páry čistého a šifrovaného textu. Taktéž při znání kontextu by u zpráv s opakujícími se znaky mohlo dojít k identifikaci čistého textu z šifrovaného.




**Obr. 2.4: Blokové schéma šifrování (a) a dešifrování (b) módu ECB. [1]**

### 2.2.2 Cipher Block Chaining (CBC)

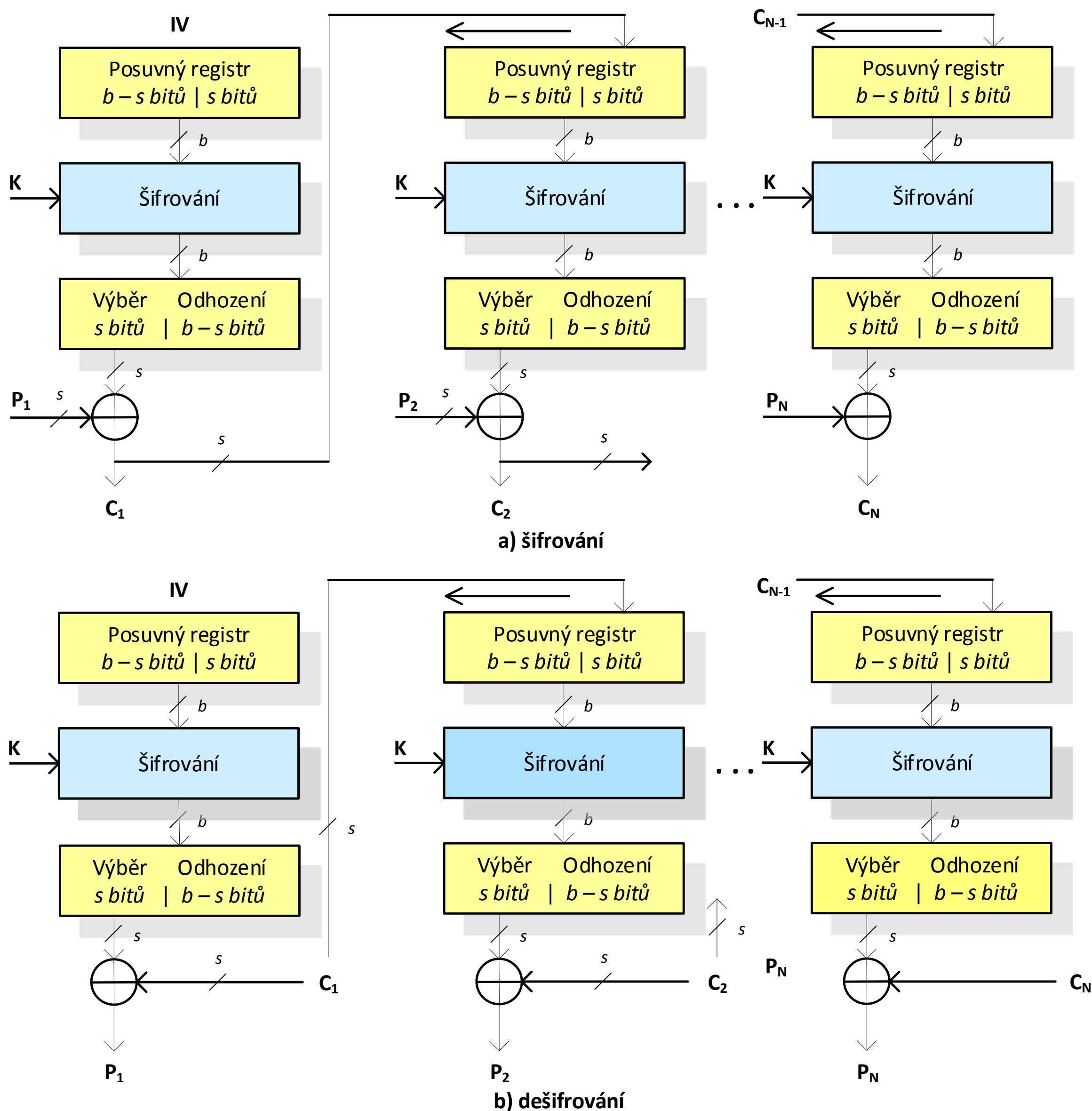
Mód řetězení bloků šifry řeší bezpečnostní nedostatek módu ECB. Mód CBC přináší závislost bloků. To znamená, že při šifrování dvou stejných čistých bloků vzniknou jiné šifrované bloky. K právě šifrovanému čistému bloku se pomocí operace XOR přidává šifrovaný blok předchozího čistého bloku. K vytvoření prvního šifrovaného bloku se používá inicializační vektor (IV), který se „XORuje" s prvním čistým blokem. IV je tajný a musí jej znát obě strany.




**Obr. 2.5: Blokové schéma šifrování (a) a dešifrování (b) módu CBC. [1]**

### 2.2.3 Cipher Feedback (CFB)

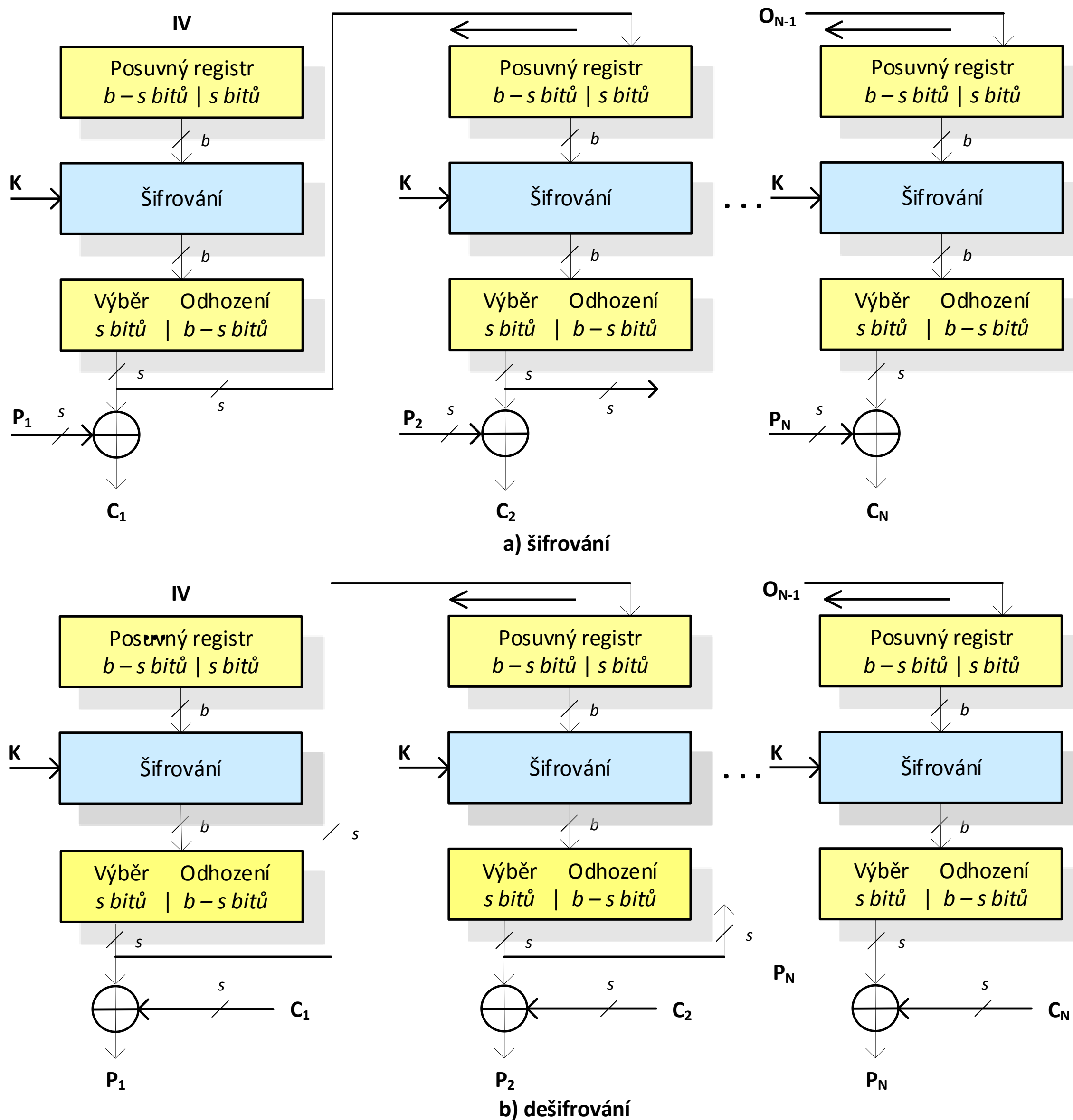
V módu CFB funguje bloková šifra jako proudová. Blokové šifry pracují s bloky o délce $b$ bitů (u šifry Advanced Encryption Standard b = 128). Bloková šifra v módu CFB umožňuješ šifrovat čistý text o délce $s < b$. V prvním kroku je ve vstupním bloku (posuvný registr) inicializační vektor, který se zašifruje. Z výstupu šifrovacího algoritmu se vezme $s$ nejvýznamnějších bitů, které se „XORují" s čistým textem. Výsledkem je $s$ bitů šifrovaného textu, které jdou také do vstupního bloku, jako $s$ nejméně významných bitů.




**Obr. 2.6: Blokové schéma šifrování (a) a dešifrování (b) módu CFB. [1]**

### 2.2.4 Output Feedback (OFB)

Mód OFB je podobný módu CFB. Rozdíl je v tom, že do vstupního bloku jde *s* bitů před provedením operace XOR nad čistým textem. Oproti CFB má OFB výhodu v tom, že pokud se stane chyba v přenosu šifrovaného textu $C_i$, bude touto chybou ovlivněn jenom čistý text $P_i$. U módu CFB (i CBC) se chyba v šifrovaném textu $C_i$ rozšiřuje, protože hodnota $C_i$ slouží také jako vstup do posuvného registru. Nevýhodou je zranitelnost proti útoku změnou šifrovaného textu.

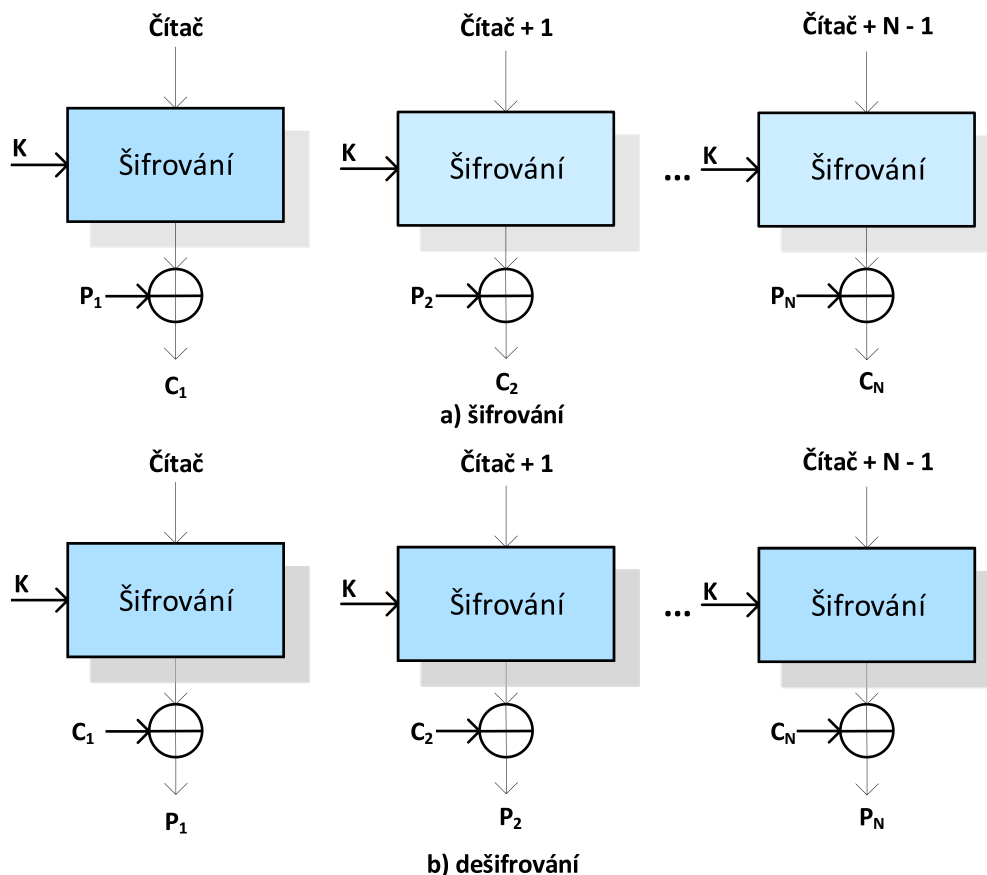



**Obr. 2.7: Blokové schéma šifrování (a) a dešifrování (b) módu OFB. [1]**

### 2.2.5 Counter (CTR)

V čítačovém módu je jako vstupní blok použit čítač, jehož hodnota se na začátku inicializuje a poté se inkrementuje. Hodnota čítače se šifruje a výstupní blok se „XORuje“ s čistým textem.

Absence zřetězení v čítačovém módu umožňuje paralelní zpracovávání více bloků při šifrování i dešifrování. Výhodou je předzpracování výstupních bloků, které se poté „XORují“ s bloky čistých textů. Další výhodou je možnost zpracovávat *i*-tý blok čistého nebo šifrovaného textu.




**Obr. 2.8: Blokové schéma šifrování (a) a dešifrování (b) čítačového módu. [1]**

### 2.2.6 Shrnutí

**Tab. 2.1: Shrnutí operačních módu blokových šifer. [1]**

| Operační mód | Popis | Aplikace |
|---|---|---|
| Elektronická kódová kniha (ECB) | Každý blok čistého textu je šifrovaný nezávisle použitím stejného klíče. | Bezpečný přenos jednoduchých (krátkých) hodnot, např. šifrovacího klíče. |
| Řetězení šifrovaného textu (CBC) | Vstupním blokem šifrovacího algoritmu je hodnota čistého textu „XORovaná“ s předchozím výstupním blokem šifrovacího algoritmu (šifrovaným textem). | Univerzální bloková šifra.<br>Autentizace. |
| Zpětná vazba ze šifrovaného textu (CFB) | Předchozí šifrovaný text je použit jako vstup šifrovacího algoritmu, který vytváří pseudo-náhodný výstup, jenž se „XORuje“ s čistým textem. | Univerzální proudová šifra.<br>Autentizace. |

| | | |
|---|---|---|
| Zpětná vazba z výstupu (OFB) | Podobný CFB. Vstupem je pseudonáhodný výstup šifro-vacího algoritmu před použitím operace XOR nad čistým textem. | Proudová šifra pro přenosy na rušených kanálech, např. satelitní komunikace. |
| Čítačový mód (CTR) | Každý blok čistého textu se XORuje s zašifrovanou hodnotou čítače. Čítač se inkrementuje pro každý následující blok čistého textu. | Univerzální blokový šifra.<br>Pro vysokorychlostní aplikace. |

# 3 Advanced Encryption Standard

AES je symetrická bloková šifra. K šifrování a dešifrování používá stejný klíč o délce 128, 192 a 256 bitů. Blok (vstupní i výstupní) má pevnou velikost 128 bitů (matice 4x4, jedna buňka odpovídá jednomu bytu). Matice se nazývá vnitřní stav šifry. [3]

**Tab. 3.1: Vnitřní stav šifry. [3]**

| | | | |
|---|---|---|---|
| $S_{0,0}$ | $S_{0,1}$ | $S_{0,2}$ | $S_{0,3}$ |
| $S_{1,0}$ | $S_{1,1}$ | $S_{1,2}$ | $S_{1,3}$ |
| $S_{2,0}$ | $S_{2,1}$ | $S_{2,2}$ | $S_{2,3}$ |
| $S_{3,0}$ | $S_{3,1}$ | $S_{3,2}$ | $S_{3,3}$ |

Šifrování a dešifrování probíhá v iteracích (rundách), jejichž počet závisí na délce klíče. Pro každou iteraci se odvozují klíče ze šifrovacího klíče. Tento proces se nazývá expanze klíče a provádí se před samotným šifrováním nebo dešifrováním. Odvozené klíče mají vždy velikost 128 bitů (16 bytů, 4 dvojslova) a nazývají se rundovní klíče („*round keys*“).. V Tab. 3.2 jsou zobrazeny velikosti klíče, expandovaného klíče a počtu iterací. [1]

**Tab. 3.2: Parametry AES. [1]**

| Velikost klíče [bity / byty / dvojslova] | Počet iterací | Velikost expandovaného klíče[1] [byty / dvojslova] |
|---|---|---|
| 128 / 16 / 4 | 10 | 176 / 44 |
| 192 / 24 / 6 | 12 | 208 / 52 |
| 256 / 32 / 8 | 14 | 240 / 60 |

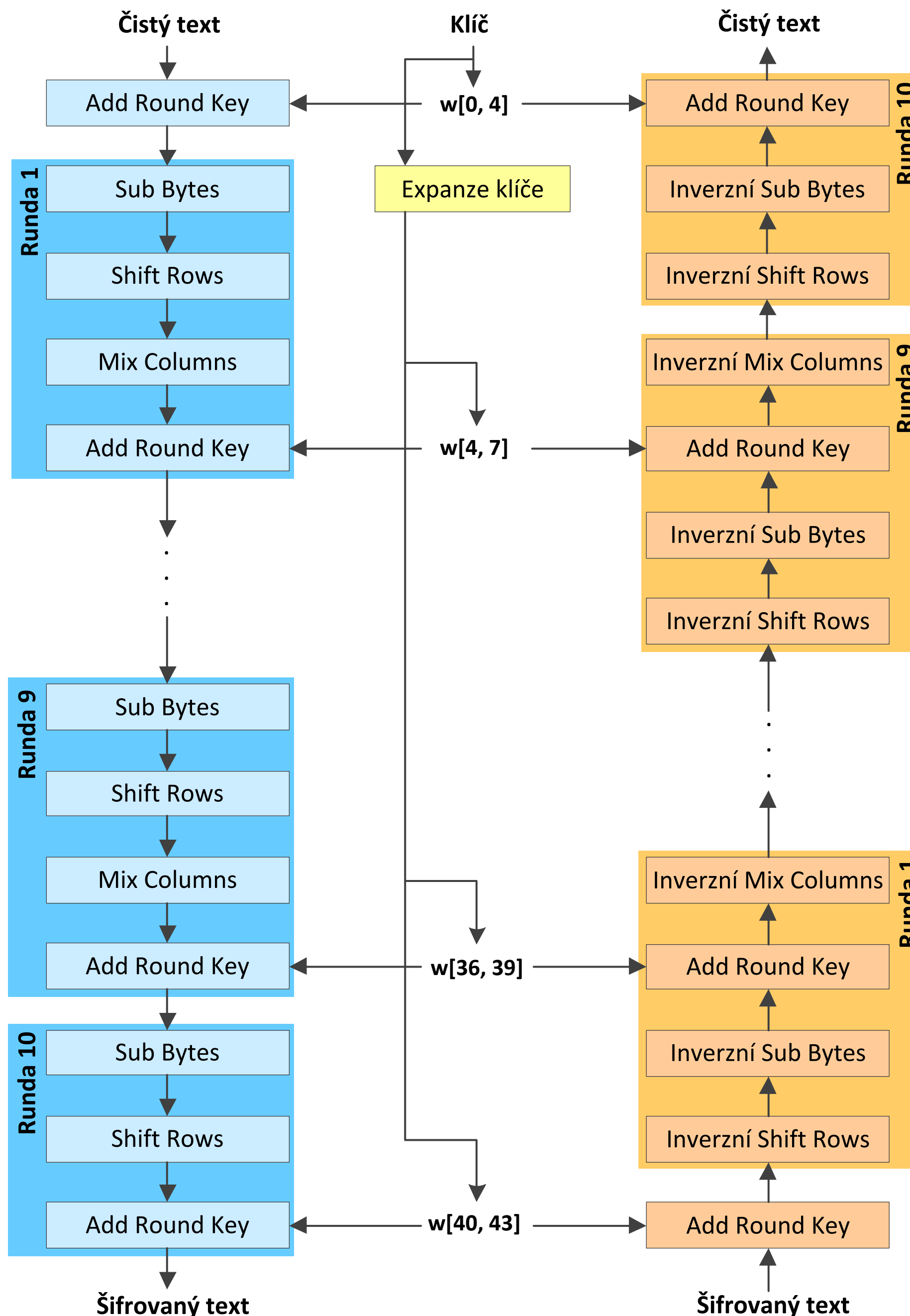
## 3.1 Algoritmus šifrování a dešifrování

Jak bylo zmíněno, před šifrováním i dešifrováním se z hlavního klíče odvodí klíče pro každou iteraci (rundovní klíče). Algoritmus expanze klíče bude popsán v další kapitole. Dle [3] se při šifrovaní se aplikuje tento algoritmus:

- Překopírování čistého textu do vnitřního stavu.
- AddRoundKey (inicializační iterace – runda 0).
- $(N-1)$-krát[2] se provede rundu pomocí transformací:
  - SubBytes,
  - ShiftRows,
  - MixColumns,
  - AddRoundKey.
- Poslední iterace se provede pomocí transformací:
  - SubBytes,
  - ShiftRows,
  - AddRoundKey.

[1] K celkové velikosti klíče je přičten klíč použitý v rundě 0, který má velikost 4 doublewordy.
[2] N je počet iterací.

Na začátku šifrování i dešifrování se provádí inicializační iterace (runda 0), při které se vstupní blok dat „XORuje“ se šifrovacím klíčem. Poté se provádí N iterací, přičemž poslední se liší od předchozích ve vynechání jedné z transformací. Jednotlivé transformace budou popsány dále. Dešifrování se provádí pomocí inverzních operací. Průběh šifrování a dešifrování je znázorněn na Obr. 3.1.




**Obr. 3.1: AES šifrování (vlevo) a dešifrování (vpravo). [1]**

V [1] autor ukazuje na to, že se šifrování a dešifrování od sebe liší (viz Obr. 3.1) v sekvencí transformací, přestože se používá stejný klíčový plán (algoritmus jejich použití). Zmiňuje, že následkem je potřeba oddělených softwarových nebo firmwarových modulů pro šifrování a dešifrování zvlášť. Šifru se stejnou sekvencí transformací - ekvivalentní inverzní šifru - lze

získat změnou klíčového plánu. Ekvivalentní inverzní šifra bude popsána níže, nejprve ale bude vysvětlena expanze klíče a používané transformace.

## 3.2 Expanze klíče

Šifrovací klíče jsou expandovány do rundovních klíčů. Každý rundovní klíč má velikost 128 bitů. Velikost expandovaného klíče se různí podle velikosti šifrovacího klíče, viz Tab. 3.2.

Označme $N_{dw}$ jako počet dvojslov (anglicky *doubleword* šifrovacího klíče (4, 6, 8 pro AES-128, AES-192, AES-256). V [1] je popsán algoritmus expanze klíče pro velikost klíče 128 bitů.

- Šifrovací klíč je zkopírovaný do prvních 0 až $(N_{dw} - 1)$ dvojslov (prvních $N_{dw}$ sloupců).
- Každý další doubleword $W[i]$ závisí na předcházejícím dvojslově $W[i-1]$ (je jeho kopií) a dvojslově o čtyři pozice zpět, $W[i - N_{dw}]$. Ve $(N_{dw} - 1)$ případech ze $N_{dw}$ se používá operace XOR. Platí, že $W[i] = W[i-1] \oplus W[i - N_{dw}]$.
- V případě, že pozice dvojslova je násobkem $N_{dw}$ čísla se provádí tyto operace:
  - Zkopíruje se doubleword, $W[i] = W[i-1]$,
  - Provede se operace $RotWord\ (W[i])$. Operace prohodí vrchní byte dolů, tedy $[B_0, B_1, B_2, B_3] \rightarrow [B_1, B_2, B_3, B_0]$.
  - Provede se operace $SubWord\ (W[i])$. V této operaci se nahradí každý byte v dvojslově pomocí substituční tabulky (Tab. 3.4).
  - Výsledek předchozích kroků je „XORován“ s $W[i-4]$ a rundovní konstantou $RCon[j]$. Výsledek je potom:
    $W[i] = W[i - N_{dw}] \oplus SubWord\left(RotWord\ (W[i-1])\right) \oplus RCon[j]$.
- U AES-256 se používá navíc operace $SubWord\ (W[i])$ pro dvojslova, které jsou násobkem čísla 4 a nejsou násobkem čísla $N_{dw} = 8$.

Rundovní konstanty jsou definovány jako $RCon[j] = [RC[j], 0, 0, 0]$, kde $RC[j] = \{02\}^{j-1}$, pro $i = 1, 2, \ldots, 10$, kde operace jsou prováděny nad Galoisovým tělesem $GF(2^8)$. Pro AES-128 se používá prvních deset hodnot $RC[j]$, viz Tab. 3.3. U AES-192 se používá prvních 8 hodnot a u AES-256 prvních 7 hodnot.

**Tab. 3.3: Hodnoty RC[j]. [1]**

| j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| RC[j] | 0x01 | 0x02 | 0x04 | 0x08 | 0x10 | 0x20 | 0x40 | 0x80 | 0x1B | 0x36 |

## 3.3 Používané transformace

Jak bylo zmíněno, nejprve se při inicializační iteraci (nultá runda) k vnitřnímu stavu „XORuje“ inicializační klíč, který se taky nazývá bělící klíč. Proto se lze potkat s označením nulté rundy jako inicializační nebo bělící runda. Poté se *n*-krát (*n* je počet iterací) provedou transformace (operace) SubBytes, ShiftRows, MixColumns a AddRoundKey. Výjimkou je poslední iterace (runda), ve které se vynechá operace MixColumns.

### 3.3.1 SubBytes

Jedná se o nelineární operaci implementovanou pomocí tabulky pro operaci SubBytes. Každý byte se rozdělí na dvě hexadecimální číslice. Pomocí první se určí řádek a pomocí druhé se určí sloupec v tabulce. Poté se každý byte vnitřního stavu nahradí podle tabulky. V tabulce jsou hodnoty 0-255 právě jednou, proto je možné provádět inverzní operaci vyhledání souřadnic a tím vyhledat původní byte.

**Tab. 3.4: Substituční tabulka. [2]**

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

### 3.3.2 ShiftRows

První řádek zůstane stejný. V druhém řádku se rotuje jeden byte doleva, v třetím se rotuje o dva byte a ve čtvrtém o tři byty.

**Tab. 3.5: Vnitřní stav po operaci ShiftRows.**

| | | | |
|---|---|---|---|
| $S_{0,0}$ | $S_{0,1}$ | $S_{0,2}$ | $S_{0,3}$ |
| $S_{1,1}$ | $S_{1,2}$ | $S_{1,3}$ | $S_{1,0}$ |
| $S_{2,2}$ | $S_{2,3}$ | $S_{2,0}$ | $S_{2,1}$ |
| $S_{3,3}$ | $S_{3,0}$ | $S_{3,1}$ | $S_{3,2}$ |

### 3.3.3 MixColumns

Tato operace pracuje samostatně s každým sloupcem matice. Označíme-li původní hodnotu $i$-tého sloupce $S_{x,i}$ a novou hodnotu $S'_{x,i}$, pak mezi nimi platí vztah:

$$\begin{bmatrix} S'_{0,i} \\ S'_{1,i} \\ S'_{2,i} \\ S'_{3,i} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} S_{0,i} \\ S_{1,i} \\ S_{2,i} \\ S_{3,i} \end{bmatrix} \quad (3.1)$$

Násobení jedničkou znamená ponechání původní hodnoty. Násobení dvojkou znamená posuvem původní hodnoty o jeden bit vlevo. Násobení trojkou znamená posuv původní hodnoty o jeden bit vlevo a následné sečtení (XOR) s původní hodnotou. Příklad výpočtu pro 1. řádek je $S'_{0,i} = 2 \cdot S_{0,i} + 3 \cdot S_{1,i} + 1 \cdot S_{2,i} + 1 \cdot S_{3,i}$. Pokud je výsledek vyšší než 255, pak se k němu ještě přičte hodnota $11B_H$.

### 3.3.4 AddRoundKey

Operace přičítání rundovního klíče pomocí exklusivního logického součtu (XOR).

**Tab. 3.6:Operace AddRoundKey.**

| | | | |
|---|---|---|---|
| $S_{0,0} \oplus K_{0,0}$ | $S_{0,1} \oplus K_{0,1}$ | $S_{0,2} \oplus K_{0,2}$ | $S_{0,3} \oplus K_{0,3}$ |
| $S_{1,0} \oplus K_{1,0}$ | $S_{1,1} \oplus K_{1,1}$ | $S_{1,2} \oplus K_{1,2}$ | $S_{1,3} \oplus K_{1,3}$ |
| $S_{2,0} \oplus K_{2,0}$ | $S_{2,1} \oplus K_{2,1}$ | $S_{2,2} \oplus K_{2,2}$ | $S_{2,3} \oplus K_{2,3}$ |
| $S_{3,0} \oplus K_{3,0}$ | $S_{3,1} \oplus K_{3,1}$ | $S_{3,2} \oplus K_{3,2}$ | $S_{3,3} \oplus K_{3,3}$ |

## 3.4 Ekvivalentní inverzní šifra

Při šifrování se používají transformace v pořadí SubBytes, ShiftRows, MixColumns, AddRounKey. Při dešifrování se používají transformace v pořadí InvShiftRows, InvSubBytes, AddRoundKey, InvMixColumns. Dle [1] je potřeba dvou změn pro dosažení ekvivalentní inverzní šifry. Jde o prohození prvních dvou transformací (InvShiftRows a InvSubBytes) a druhých dvou transformací (AddRoundKey a InvMixColumns).

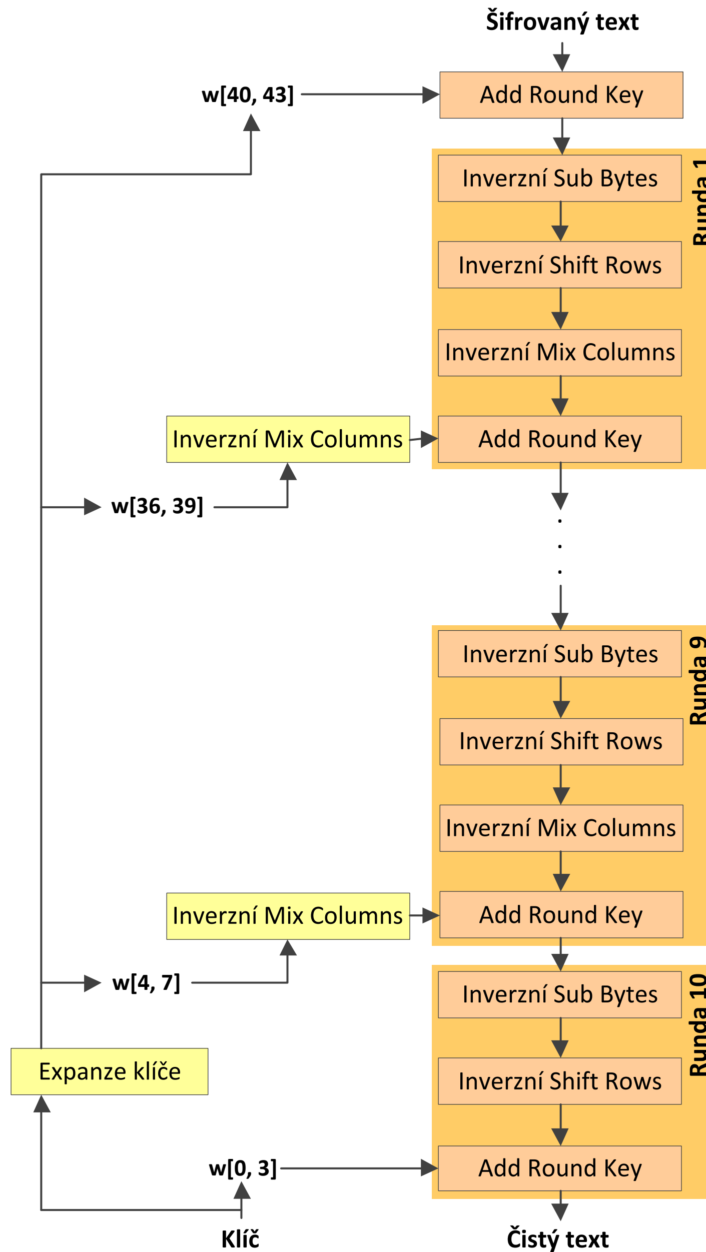
InvShiftRows mění sekvenci bytů, ale nemění jejich obsah, proto nezávisí na obsahu bytů pro provedení této operace. InvSubBytes mění obsah bytů, ale nemění jejich sekvenci, proto nezávisí na sekvenci bytů. Pořadí transformací InvShiftRows a InvSubBytes je tedy možné prohodit. Pro stav $S_i$ platí:

$$InvShiftRows\ [InvSubBytes\ (S_i)] = InvSubBytes\ [InvShiftRows\ (S_i)] \quad (3.2)$$

Operace AddRoundKey a InvMixColumns nemění sekvenci bytů ve stavu, nýbrž jejich obsah. Pro stav $S_i$ a rundovní klíč $w_j$ platí:

$$InvMixColumns\ (S_i \oplus w_j) = [InvMixColumns(S_i)] \oplus [InvMixColumns(w_j)] \quad (3.3)$$

Hodnota $(S_i \oplus w_j)$ je výstup po operaci AddRoundKey, který jde na vstup operace InvMixColums. Z rovnice je patrné, že pořadí transformací lze prohodit, aplikuje-li se nejprve transformace InvMixColumns nad rudnovním klíčem $w_j$. Tím se dosáhne ekvivalentní inverzní šifry. Průběh dešifrování je znázorněn na Obr. 3.2.




**Obr. 3.2: Průběh dešifrování ekvivalentní inverzní šifry. [1]**

# 4 Architektura Intel AES New Instruction

Intel® AES New Instruction (dále jen AES-NI) je sada instrukcí dostupná s příchodem rodiny procesorů Intel® Core™ na 32 nm Intel® mikroarchitektůře s kódovým jménem Westmere. Účelem instrukcí je poskytnutí rychlé a bezpečné implementace šifry AES, která je definována v FIPS publikaci č. 197 (viz [5]). Instrukce, jejich implementace a výsledky jejich výkonu, jsou popsané bílé knize od společnosti Intel, viz [4].

Sada instrukcí AES-NI obsahuje 6 instrukcí, které jsou plně hardwarově podporované. Čtyři poskytují šifrování a dešifrování dat a zbylé dvě zajišťují expanzi klíče. Flexibilita AES instrukcí umožňuje implementaci symetrické blokové šifry AES v různých variantách. Tedy se všechny standardními velikostmi šifrovacích klíčů a standardními módy blokových šifer. Zároveň nabízejí významný zvýšení výkonu v porovnání s klasickou softwarovou implementací. [4]

Dále v [4] uvádějí výhodu větší bezpečnosti. Díky pracování s daty nezávislých na čase a nevyužívajících tabulky jsou eliminovány časové útoky a útoky na cache, které představovali hrozbu u softwaru implementujícího AES založeného na tabulkách.

**Tab. 4.1: Seznam instrukcí Intel AES-NI.**

| Instrukce | Celý název | Funkce |
|---|---|---|
| AESENC | AES Encrypt Round | Zašifrování rundy |
| AESENCLAST | AES Encrypt Last Round | Zašifrování poslední rundy |
| AESDEC | AES Dencrypt Round | Dešifrování rundy |
| AESDECLAST | AES Dencrypt Last Round | Dešifrování poslední rundy |
| AESIMC | AES Inverse Mix Columns | Transofrmace InvMixColumns |
| AESKEYGENASSIST | AES Key Generation Assist | Asistence při generování rundovních klíčů |

Instrukce pracují s jedním nebo dvěma 128 bitovými vstupy. Formát instrukcí je:

```
instrukce xmm1, xmm2/m128
```

Xmm1 a xmm2 jsou dva registry z instrukční sady SSE[3], m128 je místo v paměti (128 bitové). Výsledek je uložen v xmm1.

## 4.1 Formát dat

Podle konvence Intel Architecture popsané v [4] se zapisují hexadecimální řetězce od nižších paměťových míst k vyšším. Tato konvence je analogická ke konvenci „Little Endian“. Máme-li 128 bitový řetězec, čteme byty v tomto pořadí: $[Byte15, Byte14, \dots, Byte1, Byte0]$, kde Byte0 je nejvíce pravý byte a první byte řetězce.

Při práci s daty AES instrukce používají dva 128 bitové registry xmm1 a xmm2, nebo jeden registr xmm1 a 128 bitové místo v paměti (m128). V [4] rozdělují obsah registrů na bity

[3] Streaming SIMD Extensions

(127–0), byty (15–0), dvojslova ($X_3$–$X_0$) a písmeny P-A označují buňky matice 4x4, která reprezentuje stav. V tabulce Tab. 4.2 je znázorněn obsah registru xmm.

**Tab. 4.2: Pořadí bitů, bytů, dvojslov a buněk matice v registru xmm. [4]**

| 127-120 | 119-112 | 111-104 | 103-96 | 95-88 | 87-80 | 79-72 | 71-64 | 63-56 | 55-48 | 47-40 | 39-32 | 31-24 | 23-16 | 15-8 | 7-0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| $X_3$ | | | | $X_2$ | | | | $X_1$ | | | | $X_0$ | | | |
| P | O | N | M | L | K | J | I | H | G | F | E | D | C | B | A |

Uspořádání bytů matice 4x4 v registru xmm je znázorněno v Tab. 4.3.

**Tab. 4.3: Uspořádání bytů registru xmm ve stavu. [4]**

| A | E | I | M |
|---|---|---|---|
| B | F | J | N |
| C | G | K | O |
| D | H | L | P |

### 4.1.1 Příklad zápisu dat

Mějme hexadecimální řetězec o délce 128 bitů (32 hexadecimálních znaků), například šifrovací klíč K:

$$K = 00\ 01\ 02\ 03\ 04\ 05\ 06\ 07\ 08\ 09\ 0A\ 0B\ 0C\ 0D\ 0E\ 0F.$$

V registru xmm pak tato hodnota bude uložena jako:

$$xmm0 = 0F\ 0E\ 0D\ 0C\ 0B\ 0A\ 09\ 08\ 07\ 06\ 05\ 04\ 03\ 02\ 01\ 00.$$

Byte 0 má hodnotu 0x00, byte 15 má hodnotu 0x0F. Pořadí bytů je znázorněno v Tab. 4.4.

**Tab. 4.4: Pořadí bytů hexadecimálního řetězce.**

| Byte | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Buňka | P | O | N | M | L | K | J | I | H | G | F | E | D | C | B | A |
| Hodnota | 0F | 0E | 0D | 0C | 0B | 0A | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |

Odpovídající matice 4x4 je pak znázorněna v Tab. 4.5.

**Tab. 4.5: Hodnoty hexadecimálního řetězce v matici 4x4.**

| 00 | 04 | 08 | 0C |
|---|---|---|---|
| 01 | 05 | 09 | 0D |
| 02 | 06 | 0A | 0E |
| 03 | 07 | 0B | 0F |

## 4.2 Rundovní instrukce

V následujících blocích je zobrazen pseudokód instrukcí AESENC, AESENCLAST pro šifrování pseudokód instrukcí AESDEC a AESDECLAST pro dešifrování, které jsou popsané v [4]. Registry xmm1 a xmm2 jsou jakékoliv dva registry z instrukční sady SSE s velikostí 128 bitů. M128 je místo v paměti s velikostí 128 bitů.

Instrukce AESDEC a AESDECLAST dešifrují podle algoritmu ekvivalentní inverzní šifry popsané v kapitole 3.4. Sled transformací je vidět na Obr. 3.2.

**Pseudokód 4.1: Instrukce pro šifrování AESENC a AESENCLAST. [4]**

```
; Instrukce AESENC
; Tento pseudokód zobrazuje algoritmus instrukce AESENC, která
; provádí rundy šifrování (první až předposlední).

AESENC xmm1, xmm2/m128
                ; formát instrukce, xmm1 obsahuje
                ; vstupní blok, xmm2/m128 obsahuje rundovní klíč
Tmp := xmm1
                ; uložení vstupního bloku do Tmp
Round Key := xmm2/m128
                ; uložení rundovního klíče

; Nyní se provedou transformace podle algoritmu AES.
Tmp := ShiftRows (Tmp)
                ; operace ShiftRows
Tmp := SubBytes (Tmp)
                ; operace SubBytes
Tmp := MixColumns (Tmp)
                ; operace MixColumns
xmm1 := Tmp XOR Round Key
                ; přidání rundovního klíče, výsledek v xmm1

; Instrukce AESENCLAST
; Tento pseudokód zobrazuje algoritmus instrukce AESENCLAST,
; která provádí poslední rundu šifrování.

AESENCLAST xmm1, xmm2/m128
                ; formát instrukce, xmm1 obsahuje
                ; vstupní blok, xmm2/m128 obsahuje rundovní klíč
Tmp := xmm1
                ; uložení vstupního bloku do Tmp
Round Key := xmm2/m128
                ; uložení rundovního klíče

; Nyní se provedou transformace podle algoritmu AES.
; Dle algoritmu se v poslední rundě neprovádí operace MixColumns.

Tmp := ShiftRows (Tmp)
```

```
                ; operace ShiftRows
Tmp := SubBytes (Tmp)
                ; operace SubBytes
xmm1 := Tmp XOR Round Key
                ; přidání rundovního klíče
                ; výsledek v xmm1 je šifrovaný text.
```

**Pseudokód 4.2: Instrukce pro dešifrování AESDEC a AESDECLAST. [4]**

```
; Instrukce AESDEC
; Tento pseudokód zobrazuje algoritmus instrukce AESDEC, která
; provádí rundy dešifrování (první až poslední).

AESDEC xmm1, xmm2/m128
                ; formát instrukce, xmm1 obsahuje
                ; vstupní blok, xmm2/m128 obsahuje  rundovní klíč
Tmp := xmm1
                ; uložení vstupního bloku do Tmp
Round Key := xmm2/m128
                ; uložení rundovního klíče

; Nyní se provedou transformace podle algoritmu, který odpovídá
; ekvivalentní inverzní šifře popsané v kapitole 3.4.

Tmp := InvShiftRows (Tmp)
                ; operace inverzní ShiftRows
Tmp := InvSubBytes (Tmp)
                ; operace inverzní SubBytes
Tmp := InvMixColumns (Tmp)
                ; operace inverzní MixColumns
xmm1 := Tmp XOR Round Key
                ; přidání rundovního klíče

; Instrukce AESDECLAST
; Tento pseudokód zobrazuje algoritmus instrukce AESDECLAST,
; která provádí poslední rundu dešifrování.

AESDECLAST xmm1, xmm2/m128
                ; formát instrukce, xmm1 obsahuje
                ; vstupní blok, xmm2/m128 obsahuje rundovní klíč
Tmp := xmm1
                ; uložení vstupního bloku do Tmp
Round Key := xmm2/m128
                ; uložení rundovního klíče

; Nyní se provedou transformace podle algoritmu AES.
; Dle algoritmu se v poslední rundě neprovádí operace InvMixColumns
```

```
Tmp := InvShiftRows (Tmp)
                ; operace inverzní ShiftRows
Tmp := InvSubBytes (Tmp)
                ; operace inverzní SubBytes
xmm1 := Tmp XOR Round Key
                ; přidání rundovního klíče, výsledek v xmm1
                ; je čistý text
```

Příklad kódu pro šifrování AES s délkou klíče 128 bitů.

**Kód 4.1: Šifrování AES s klíčem o velikosti 128 bitů. [4]**

```
; Sekvence instrukcí pro šifrování AES s klíčem o délce 128b.
; Blok dat je uložený v registru xmm15.
; V registrech xmm0-xmm10 jsou uložený rundovní klíče.
; Výsledek šifrování je uložen v registru xmm15.

PXOR xmm15, xmm0                ; inicializační runda, runda 0
AESENC xmm15, xmm1              ; runda 1
AESENC xmm15, xmm2              ; runda 2
AESENC xmm15, xmm3              ; runda 3
AESENC xmm15, xmm4              ; runda 4
AESENC xmm15, xmm5              ; runda 5
AESENC xmm15, xmm6              ; runda 6
AESENC xmm15, xmm7              ; runda 7
AESENC xmm15, xmm8              ; runda 8
AESENC xmm15, xmm9              ; runda 9
AESENCLAST xmm15, xmm10         ; runda 10, poslední runda
```

Na začátku se provádí inicializační runda, která bývá označována jako runda 0, nebo bělící runda. V tomto kroku se „XORuje" stav s rundovním klíčem 0. Dále se provede 10 rund s klíči 1-10. V poslední rundě je použita instrukce AESDECLAST, ve které se neprovádí transformace MixColumns. Rund je celkově 10 plus inicializační runda, což odpovídá šifrování AES s délkou klíče 128 bitů, viz Tab. 3.2.

Příklad kódu pro dešifrování AES s délkou klíče 192 bitů.

**Kód 4.2: Dešifrování AES s klíčem o velikosti 192 bitů. [4]**

```
; Sekvence příkazů pro dešifrování AES s klíčem o délce 192b.
; Data jsou v registru xmm15.
; V registrech xmm0-xmm12 jsou uložený dešifrovací rundovní
; klíče.
; Výsledek dešifrování je uložen v registru xmm15.

PXOR xmm15, xmm12               ; inicializační runda, runda 0
AESDEC xmm15, xmm11             ; runda 1
AESDEC xmm15, xmm10             ; runda 2
AESDEC xmm15, xmm9              ; runda 3
AESDEC xmm15, xmm8              ; runda 4
AESDEC xmm15, xmm7              ; runda 5
```

```
AESDEC xmm15, xmm6          ; runda 6
AESDEC xmm15, xmm5          ; runda 7
AESDEC xmm15, xmm4          ; runda 8
AESDEC xmm15, xmm3          ; runda 9
AESDEC xmm15, xmm2          ; runda 10
AESDEC xmm15, xmm1          ; runda 11
AESDECLAST xmm15, xmm0          ; runda 12, poslední runda
```

Algoritmus dešifrování odpovídá ekvivalentní inverzní šifře, to znamená že dešifrovací klíče (krom prvního a posledního) jsou odvozeny z šifrovacích pomocí transformace InvMixColumns a až poté jsou aplikovány (při operaci AddRoundKey). Klíče se používají oproti šifrování v obráceném pořadí.

## 4.3 Instrukce pro generování klíčů

AES-NI nabízí dvě instrukce pro generování klíčů. Instrukce AESKEYGENASSIST pomáhá s generování rundovních klíčů k šifrování. Instrukce AESIMC se používá pro transformaci šifrovacích klíčů na klíče používané k dešifrování ve smyslu ekvivalentní inverzní šifry.

**Pseudokód 4.3: Instrukce pro asistenci expanze klíče AESKEYGENASSIST. [4]**

```
; Instrukce AESKEYGENASSIST
; Instrukce napomáhá expanzi klíče.

AESKEYGENASSIST xmm1, xmm2/m128, imm8
                ; v registru xmm1 je uložen výsledek,  v xmm2/m128
                ; je 128 bitový vstup, v imm8 je uložena hodnota
                ; RC[j] (viz Tab. 3.3)
Tmp := xmm2/LOAD(m128)
                ; uložení 128 bit. vstupu do Tmp z registru xmm2
                ; nebo z paměti
X3[31-0] := Tmp[127-96]
                ; uložení bitů 127-96 do doublewordu X3
X2[31-0] := Tmp[95-64]
                ; uložení bitů 95-64 do doublewordu X2
X1[31-0] := Tmp[63-32]
                ; uložení bitů 63-20 do doublewordu X1
X0[31-0] := Tmp[31-0]
                ; uložení bitů 31-0 do doublewordu X0
RCon[7-0] := imm8
                ; uložení RC[j] hodnoty do nejnižšího bytu
                ; doublewordu 0 rundovní konstanty RCon
RCon[31-8] := 0
                ; naplnění ostatních bytů rundovní konstanty
                ; RCon nulou
xmm1 := [   RotWord (SubWord (X3)) XOR RCON,
            SubWord (X3),
            RotWord (SubWord (X1)) XOR RCON,
            SubWord (X1)]
```

```
                ; xmm1 = [X3, X2, X1, X0]
                ; X3 je výsledkem operací SubWord a RotWord nad X3
                ; a následném XORování s rundovní konstantou RCon,
                ; X2 je výsledkem operace SubWord nad X3,
                ; X1 je výsledkem operací SubWord a RotWord nad X1
                ; a následném XORování s rundovní konstantou Rcon,
                ; X0 je výsledkem operace SubWord nad X1.
```

Z pseudokódu instrukce AESKEYGENASSIST je patrné, že instrukce sama o sobě neprovádí expanzi klíče, ale připravuje data pro další zpracování. Pro expanzi klíče je potřeba provést další operace nad 128 bity šifrovacího klíče a výstupními 128 bity instrukce AESKEYGENASSIST. V [4] uvádí, že je několik způsobů pro expanzi klíče s využitím instrukce AESKEYGENASSIST. Jeden ze způsobů je znázorněn a popsán v Kód 4.3, ve kterém je použita instrukce AESKEYGENASSIST pro asistenci generování rundovních klíčů u AES s délkou klíče 128b.

**Kód 4.3: Příklad použití AESKEYGENASSIST pro generování rundovních klíčů. [4]**

```
; Expanze klíče u AES-128.
; Šifrovací klíč je uložen v proměnné Key.
; Klíče jsou ukládány v poli Key_Schedule.

MOVDQU xmm1, XMMWORD PTR Key
                ; nahraje šifrovací klíče do registru xmm1
MOVDQU XMMWORD PTR Key_Schedule, xmm1
                ; nahraje šifrovací klíč v xmm1 do pole Key_Schedule
MOV rcx, OFFSET Key_Schedule+16 ; uloží do registru rcx
                ; (64 bitový registr) offset Key_Schedule+16
                ; (Key_Schedule+128 bitů), pro uložení dalšího
                ; rudnovního klíče za první šifrovací klíč

; Expanze klíče
AESKEYGENASSIST xmm2, xmm1, 0x1
                ; v xmm2 je uložen výsledek, v xmm1 je  uložen klíč
                ; a 3. parametr je rundovní konstanta (viz Tab. 3.3)
CALL key_expansion_128
                ; zavolá podprogram pro dokončení expanze klíče
                ; (podprogram je uveden níže)

; Nyní se opakuje stejný kód pro každý další rundovní klíč.
; Jediným rozdílem je třetí parametr odpovídající rundovní konstantě
; RCon[j], viz Tab. 3.3.
AESKEYGENASSIST xmm2, xmm1, 0x2
CALL key_expansion_128
AESKEYGENASSIST xmm2, xmm1, 0x4
CALL key_expansion_128
AESKEYGENASSIST xmm2, xmm1, 0x8
CALL key_expansion_128
AESKEYGENASSIST xmm2, xmm1, 0x10
```

```
CALL key_expansion_128
AESKEYGENASSIST xmm2, xmm1, 0x20
CALL key_expansion_128
AESKEYGENASSIST xmm2, xmm1, 0x40
CALL key_expansion_128
AESKEYGENASSIST xmm2, xmm1, 0x80
CALL key_expansion_128
AESKEYGENASSIST xmm2, xmm1, 0x1b
CALL key_expansion_128
AESKEYGENASSIST xmm2, xmm1, 0x36
CALL key_expansion_128
JMP END;
     ; instrukce používané v podprogramu jsou popsané v kapitole 0
key_expansion_128:
    PSHUFD xmm2, xmm2, 0xff
                ; doubleword X3 registru xmm2 zkopíruje do
                ; všech doublewordů X0-X3 registru xmm2
    VPSLLDQ xmm3, xmm1, 0x4
                ; do registru xmm3 je zkopírován obsah registru
                ; xmm1, který posunut o 4 byty doleva, nižší
                ; byty jsou vynulovány
    PXOR xmm1, xmm3
                ; operace xmm1 XOR xmm3, výsledek je v xmm1
    VPSLLDQ xmm3, xmm1, 0x4
                ; do registru xmm3 je zkopírován obsah registru
                ; xmm1, který posunut o 4 byty doleva, nižší
                ; byty jsou vynulovány
    PXOR xmm1, xmm3
                ; operace xmm1 XOR xmm3, výsledek je v xmm1
    VPSLLDQ xmm3, xmm1, 0x4
                ; do registru xmm3 je zkopírován obsah registru
                ; xmm1, který posunut o 4 byty doleva, nižší
                ; byty jsou vynulovány
    PXOR xmm1, xmm3
                ; operace xmm1 XOR xmm3, výsledek je v xmm1
    PXOR xmm1, xmm2
                ; operace xmm1 XOR xmm2, výsledkem je  rundovní klíč
                ; v registru xmm1
    MOVDQU XMMWORD PTR [rcx], xmm1
                ; uložení rundovního klíče
    ADD rcx, 0x10
                ; přičte k rcx hodnotu 16 ~ posun o 128 bitů
    ret
END:
```

Výsledkem je pole Key_Schedule s 11 šifrovacími klíči.

$$Key_Schedule = [šifrovací\ klíč_0, rundovní\ klíč_1, rundovní\ klíč_2, \dots, rundovní\ klíč_{10}]$$

První až předposlední rundovní klíč musí, před tím, než je k dešifrování použit, projít operací InvMixColumns. K tomuto se používá instrukce AESIMC.

**Pseudokód 4.4: Instrukce AESIMC. [4]**

```
; Instrukce AESIMC
; Provádí transformaci InvMixColumns nad rundovním klíčem, který
; je uložen v xmm2 nebo m128.

AESIMC xmm1, xmm2/m128
                ; formát instrukce
RoundKey := xmm2/m128
                ; uloží rundovní klíč
xmm1 := InvMixColumns (RoundKey)
                ; provede transformaci InvMixColumns nad rundovním
                ; klíčem a výsledek uloží do xmm1
```

Příklad použití instrukce AESIMC:

**Kód 4.4: Použití instrukce AESIMC pro transformaci šifrovacích klíčů na dešifrovací. [6]**

```
; V poli Key_Schedule jsou uložený rundovní klíče 0-10 (AES-128).
; Do pole Key_Schedule_Decrypt budou ukládány dešifrovací klíče.
; Podle klíčového plánu ekvivalentní inverzní šifry bude
; transformace InvMixColumns bude použita na rundovní klíče 1-9.

MOV rdx, OFFSET Key_Schedule
                ; uložení adresy v paměti, kde začíná Key_Schedule
                ; do registru RDX
MOV rax, OFFSET Key_Schedule_Decrypt
                ; uložení adresy v paměti, kde začíná
                ; Key_Schedule_Decrypt do registru RAX
MOVDQU xmm1, XMMWORD PTR [rdx]
                ; uložení šifrovacího klíče (prvních 128 bitů
                ; v paměti) z pole klíčů Key_Schedule do xmm1
MOVDQU XMMWORD PTR [rax], xmm1
                ; uložení šifrovacího klíče do prvních 128 bitů
                ; Key_Schedule_Decrypt
aDD rdx, 0x10
                ; Přidá k RDX hodnotu 16 bytů (128 bitů)
ADD rax, 0x10
                ; Přidá k RAX hodnotu 16 bytů (128 bitů)
MOV ecx, 9
                ; do ECX se uloží počet rundovních klíčů, na které
                ; se bude aplikovat InvMixColumns (9 pro AES-128,
                ; 11 pro AES-192, 13 pro AES-256)

; Smyčka, která provádí InvMixColumns pomocí AESIMC instrukce na
; daných klíčech (Nr-1)-krát (Nr = počet rund)
repeat_Nr_minus_one_times:
  MOVDQU xmm1, XMMWORD PTR [rdx]
```

```
                ; načte do registru xmm1 1. rundovní klíč z pole
                ; Key_Schedule
  AESIMC xmm1, xmm1
                ; provede transformaci InvMixColumns nad 1.
                ; rundovním klíčem uloženým v xmm1 a uloží výsledek
                ; do xmm1
  MOVDQU XMMWORD PTR [rax], xmm1
                ; uloží transformovaný klíč do pole
                ; Key_Schedule_Decrypt
  ADD rdx, 0x10
                ; Přidá k RDX hodnotu 16 bytů (128 bitů)
  ADD rax, 0x10
                ; Přidá k RAX hodnotu 16 bytů (128 bitů)
loop repeat_Nr_minus_one_times
                ; po každém skoku zpět na repeat_Nr_minus_one_times
                ; je registr ECX dekrementován o 1

MOVDQU xmm1, XMMWORD PTR [rdx]
                ; uložení posledního rundovního klíče 10 z pole
                ; Key_Schedule do xmm1
MOVDQU XMMWORD PTR [rax], xmm1
                ; uložení posledního rundovního klíče 10 z xmm1
                ; do pole Key_Schedule_decrypt
```

Výsledkem je pole Key_Schedule_decrypt s 11 dešifrovacími klíči. Nultý a poslední je stejný jako v Key_Schedule. Klíče 1-9 prošli transformací InvColumnsMix (IRK).

$$Key_Schedule_Decrypt = [\check{s}ifrovac\acute{i}\ kl\acute{i}\check{c}_0, IRK_1, IRK_2, \dots, IRK_9, rundovn\acute{i}\ kl\acute{i}\check{c}_{10}]$$

Při dešifrování se klíče z pole Key_Schedule_Decrypt používají v obráceném pořadí, tedy první je použit rundovní $klíč_{10}$, poslední je použit šifrovací $klíč_0$.

# 5 Implementace šifry Advanced Encryption Standard

Úkolem praktické části diplomové práce je implementovat v jazyce symbolických instrukcí šifru AES pro velikosti klíče 128, 192 a 256 bitů, pomocí instrukcí z instrukční sady AES-NI. Implementované jsou i operační režimy blokových šifer ECB, CBC, CTR, CFB a OFB, které jsou popsané v kapitole 2.2. Cílem je vytvořit dynamickou knihovnu, které bude poskytovat funkce implementující šifru AES a zmíněné operační režimy.

Nejprve budou popsány instrukce použité při tvorbě funkcí knihovny. Dále bude popsána expanze šifrovacího klíče a odvození dešifrovacích klíčů. Následně bude popsáno šifrování a dešifrování AES a implementace jednotlivých operačních režimů. Nakonec bude popsáno vytvoření dynamické knihovny.

Knihovna je vytvořena na rozhraní Win32, což je aplikačně programové rozhraní (API) OS Windows pro 32 bitovou architekturu. Funkce knihovny budou zkoušeny a využívány pomocí skriptů v prostředí Matlab.

V následujícím textu je označováno dvojslovo klíče (32 bitů) písmenem *W* a dvojslovo xmm registru písmenem *X*. *LSB* (*Least Significant Bit*) je nejméně významný bit. *E(P,K)* znamená šifrování čistého textu *P* klíčem *K*. *D(C,K)* znamená dešifrování šifrovaného textu *C* klíčem *K*. Písmeno *I* označuje vstupní blok, *O* označuje výstupní blok a *IV* označuje inicializační vektor.

## 5.1 Používané instrukce

Při tvorbě knihovny byly kromě AES-NI instrukcí použity následující instrukce. Instrukce jsou popsané podle [6].

### 5.1.1 PSHUFD

Packed Shuffle Doublewords (PSHUFD) je instrukce dostupná s příchodem instrukční sady SSE2. Instrukce obmění dvojslova (32 bitů) zdrojového operandu podle masky, kterou představuje třetí operand, a uloží je do cílového operandu.

**Instrukce 5.1: PSHUFD.**

```
PSHUFD xmm1, xmm2/m128, imm8
                ; Přehází dvojslova registru xmm2 nebo 128
                ; bitového místa v paměti podle masky imm8 a uloží
                ; je do registru xmm1.
VPSHUFB xmm1, xmm2/m128, imm8
                ; Stejné jako PSHUFD
```

Maska má velikost 1 byte. Dvojice bitů odpovídají pozici dvojslova cílového operandu. Hodnota dvojice bitů určuje, kolikátý doubleword zdrojového operandu bude uložen v cílovém operandu.

**Tab. 5.1: Maska instrukce PSHUFD.**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Pořadí dvojslov cílového operandu | 3 | | 2 | | 1 | | 0 | |

### 5.1.2 PSHUFB

Packed Shuffle Bytes (PSHUFB) je instrukce dostupná s příchodem instrukční sady SSSE3 (Supplemental Streaming SIMD Extensions 3). Instrukce obmění byty cílového operandu (xmm registr) podle masky, kterou představuje druhý operand. Druhý operand může být xmm registr nebo 128 bitové místo v paměti.

**Instrukce 5.2: PSHUFB.**

```
PSHUFB xmm1, xmm2/m128
                ; Přehází byty registru xmm1 podle obsahu xmm2/m128
VPSHUFB xmm1, xmm2, xmm3/m128
                ; Přehází byty registru xmm2 podle obsahu xmm2/m128
                ; a uloží je do xmm1
```

Pořadí bytu masky odpovídá pořadí bytu cílového operandu. V jednotlivých bytech masky je uložená hodnota, jenž určuje, kolikátý byte (0 až 15) bude přehozen na danou pozici. Hodnota je nastavována prvními čtyřmi bity. Je-li v bytu masky nastaven nejvýznamnější bit na hodnotu 1, byte cílového operandu na odpovídající pozici bude roven nule a to bez ohledu na hodnotu v selektorových bitech.

**Tab. 5.2: Byte masky.**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| | Nulující bit | | | | Selektorové bity | | | |

### 5.1.3 PSLLDQ a PSRLDQ

Packed Shift Left / Right Logical, Double Quadword (PSLLDQ / PSRLDQ) je instrukce dostupná s příchodem instrukční sady SSE2. Instrukce posouvá byty cílového operandu o hodnotu specifikovanou v druhém operandu. Spodní (posun doleva) nebo horní (posun doprava) byty jsou vynulovány.

**Instrukce 5.3: PSLLDQ a PSRLDQ.**

```
PSLLDQ xmm1, imm8
PSRLDQ xmm1, imm8
                ; Posunutí bytů registru xmm1 doleva / doprava
                ; o hodnotu specifikovanou v imm8
VPSLLDQ xmm1, xmm2, imm8
VPSRLDQ xmm1, xmm2, imm8
                ; Posunutí bytů registru xmm2 doleva / doprava
                ; o hodnotu specifikovanou v imm8, výsledek je
                ; uložen v registru xmm1
```

### 5.1.4 PSLLQ a PSRLQ

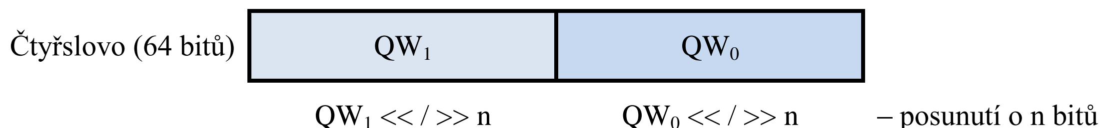
Packed Shift Left / Right Logical, Quadword (PSLLQ / PSRLQ) je instrukce dostupná s příchodem instrukční sady SSE2. Instrukce posouvá bity cílového operandu o hodnotu specifikovanou v druhém operandu. Spodní (posun doleva) nebo horní (posun doprava) bity jsou vynulovány.

**Instrukce 5.4: PSLLQ a PSRLQ.**

```
PSLLQ xmm1, xmm2/m128/imm8
PSRLQ xmm1, xmm2/m128/imm8
                ; Posunutí bitů registru xmm1 doleva / doprava
                ; o hodnotu specifikovanou v xmm2/m128/imm8.
VPSLLQ xmm1, xmm2, xmm3/m128/imm8
VPSRLQ xmm1, xmm2, xmm3/m128/imm8
                ; Posunutí bitů registrů xmm2 doleva / doprava
                ; o hodnotu specifikovanou v xmm2/m128/imm8,
                ; výsledek je uložen v xmm1.
```

Posouvání bitů se provádí odděleně pro obě čtyřslova (64 bitů) xmm registru. To znamená, že při posunutí doleva (k významnějším bitům) se nejvíce významné bity prvního čtyřslova jsou ztraceny – nepřenesou na nejméně významné bity druhého čtyřslova. Stejně tak to funguje při posunutí doprava (k méně významným bitům), kde se ztrácí nejméně významnější bity druhého čtyřslova a nepřenášejí se na nejvíce významné bity prvního čtyřslova.

**Tab. 5.3: Posouvání PSLLQ / PSRLQ.**




### 5.1.5 PXOR

Logical Exclusive OR je instrukce dostupná s příchodem instrukční sady SSE2. Instrukce provádí bitovou exkluzivní disjunkci nad zdrojovým a cílovým operandem. Výsledek je uložen v cílovém operandu.

**Instrukce 5.5: PXOR.**

```
PXOR xmm1, xmm2
                ; Provede operaci xmm1 ⊕ xmm2, výsledek je uložen
                ; v xmm1
```

### 5.1.6 SHLD a SHRD

Double Precision Shift Left / Right je instrukce používána pro posun bitů. Při posunu doleva (SHLD) se posunou bity cílového registru o hodnotu v třetím operandu a nejméně významné bity se nahradí nejvíce významnými bity zdrojového registru. Při posunu doprava se nahradí nejvíce významné bity posouvaného cílového registru nejméně významnými bity zdrojového registru.

Instrukce 5.6: SHLD a SHRD.

```
SHLD r32, r32, imm8
                ; posun bitů cílového registru doleva o hodnotu
                ; uloženou v imm8 s nahrazením bitů ze zdrojového
                ; registru
SHRD r32, r32, imm8
                ; posun bitů cílového registru doleva o hodnotu
                ; uloženou v imm8 s nahrazením bitů ze zdrojového
                ; registru
```

## 5.2 Expanze klíče

V kapitole 4.3 byla popsána instrukce AESKEYGENASSIST pro podporu expanze klíče. Jak bylo zmíněno, instrukce připravuje data pro další zpracování a sama tedy expanzi klíče neprovádí. Dokončení expanze klíče bude popsáno v následujících kapitolách a poté bude popsána implementace odvození dešifrovacích klíčů pomocí instrukce AESIMC.

### 5.2.1 Implementace u AES-128

V této kapitole bude popsán postup implementace expanze klíče AES-128 a bude doplněn a části kódu provádějící expanzi. Při implementaci pro AES-128 bylo vycházeno ze vzorové knihovny dostupné na [7]. Je v ní využito instrukce AESKEYGENASSIST a některých instrukce z instrukční sady SSE.

V případě AES s velikostí klíče 128 bitů (4 dvojslova) má expandovaný klíč velikost 44 dvojslov. První čtyři dvojslova ($W_0$ až $W_3$) tvoří šifrovací klíč. Další čtveřice jsou odvozovány z předchozích čtyř dvojslov dle algoritmu expanze klíče popsaného v kapitole 4.3.

Tab. 5.4: Expandovaný klíč AES-128.

| $W_0$ | $W_1$ | $W_2$ | $W_3$ | $W_4$ | $W_5$ | $W_6$ | $W_7$ | ... | $W_{40}$ | $W_{41}$ | $W_{42}$ | $W_{43}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| | | | | | | | | | | | | |
| Šifrovací klíč | | | | 1. rundovní klíč | | | | | 10. rundovní klíč | | | |

V rovnici (5.1 je odvozen obsah dvojslov prvního rundovního klíče.

$$RK_1 = \begin{bmatrix} W_4 \\ W_5 \\ W_6 \\ W_7 \end{bmatrix} = \begin{bmatrix} W_3' \oplus \mathrm{W}_0 \\ W_4 \oplus W_1 \\ W_5 \oplus W_2 \\ W_6 \oplus W_3 \end{bmatrix} = \begin{bmatrix} RotWord\big(SubWord(W_3)\big) \oplus \mathrm{RCon}(1) \oplus W_0 \\ RotWord\big(SubWord(W_3)\big) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \\ RotWord\big(SubWord(W_3)\big) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \oplus W_2 \\ RotWord\big(SubWord(W_3)\big) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \oplus W_2 \oplus W_3 \end{bmatrix} \tag{5.1}$$

### *Postup implementace*

Nejprve překopírujeme šifrovací klíč na začátek pole s expandovaným klíčem. Dále pomocí AESKEYGENASSIST vytvoříme doubleword $RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1)$. Pro právě takový výstup je potřeba, aby zdrojový operand instrukce AESKEYGENASSIST měl na pozici X3 poslední doubleword šifrovacího klíče ($W_3$), viz Pseudokód 4.3. Výstupní doubleword $RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1)$ poté pomocí instrukce PSHUFD vložíme do všech dvojslov registru xmm.

**Kód 5.1: Nahrání klíče a vytvoření RotWord(SubWord(W₃) ⊕ RCon(1).**

```
MOVDQU xmm1, [Key]
                ; přesune nejnižší 4 dvojslova šifrovacího
                ; klíče do xmm1 registru (W0, W1, W2, W3)
MOVDQU oword [Key_Schedule_128], xmm1
                ; uloží klíč na začátek pole
                ; prefix oword značí velikost vkládaných dat
                ; (osmislovo = 128 bitů) - není povinný
                ; xmm1[X3] = W3
AESKEYGENASSIST xmm1, xmm1, 0x1
                ; xmm1[X3] = RotWord(SubWord(W3)) ⊕ RCon(1)
PSHUFD xmm1, xmm1, 0xff
                ; s maskou 0xFF = 11111111b obmění všechny
                ; dvojslova X0 až X3 registru xmm1 na hodnotu
                ; v xmm3[X3]
                ; xmm1[X0-X3] = RotWord(SubWord(W5)) ⊕RCon(1)
```

Výsledkem bude registr $xmm_1$ obsahující:

$$xmm_1 = \begin{bmatrix} X0 \\ X1 \\ X2 \\ X3 \end{bmatrix} = \begin{bmatrix} RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \\ RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \\ RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \\ RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \end{bmatrix} \quad (5.2)$$

Pro dosažení hodnot v rovnici (5.1) je potřeba provést operaci exkluzivní disjunkce (XOR) nad registrem $xmm_1$ (rovnice (5.2)) a registrem $xmm_2$, jenž bude obsahovat hodnoty, které jsou v následující rovnici (5.3).

$$xmm_2 = \begin{bmatrix} X0 \\ X1 \\ X2 \\ X3 \end{bmatrix} = \begin{bmatrix} W_0 \\ W_0 \oplus W_1 \\ W_0 \oplus W_1 \oplus W_2 \\ W_0 \oplus W_1 \oplus W_2 \oplus W_3 \end{bmatrix} \quad (5.3)$$

Matematické odvození je vidět na rovnici (5.4). V každém kroku se provádí operace XOR nad dvěma xmm registry (každý představuje jedna matice), druhý je od prvního posunutý o jeden doubleword právě pomocí instrukce PSLLDQ.

$$xmm_2 = \begin{bmatrix} W_0 \\ W_1 \\ W_2 \\ W_3 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ W_0 \\ W_1 \\ W_2 \end{bmatrix} = \begin{bmatrix} W_0 \\ W_0 \oplus W_1 \\ W_1 \oplus W_2 \\ W_2 \oplus W_3 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ W_0 \\ W_0 \oplus W_1 \\ W_1 \oplus W_2 \end{bmatrix} = \begin{bmatrix} W_0 \\ W_1 \\ W_0 \oplus W_2 \\ W_1 \oplus W_3 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ W_0 \\ W_1 \\ W_0 \oplus W_2 \end{bmatrix} = \\ = \begin{bmatrix} W_0 \\ W_0 \oplus W_1 \\ W_0 \oplus W_1 \oplus W_2 \\ W_0 \oplus W_1 \oplus W_2 \oplus W_3 \end{bmatrix} \tag{5.4}$$

Tohoto lze dosáhnout uložením čtyř dvojslov šifrovacího klíče $W_0$ až $W_3$ do $xmm_2$ registru a postupnými úpravami pomocí instrukcí PSLLDQ a PXOR.

**Kód 5.2: Dokončení odvození klíče.**

```
                    ; xmm2=[W3,W2,W1,W0]
VPSLLDQ xmm3, xmm2, 0x4
                    ; xmm3=[W2,W1,W0,0]
PXOR xmm2, xmm3
                    ; xmm2 = [W2 ⊕ W3, W1 ⊕ W2, W0 ⊕ W1, W0] - výsledek
                    ; prvního kroku v rovnici (5.4)
VPSLLDQ xmm3, xmm2, 0x4
PXOR xmm2, xmm3
                    ; xmm2 obsahuje výsledek druhého kroku rovnice (5.4)
VPSLLDQ xmm3, xmm2, 0x4
PXOR xmm2, xmm3
                    ; xmm2 obsahuje konečný výsledek rovnice (5.4)
PXOR xmm2, xmm1
                    ; v xmm2 je uložen RK1
MOVDQU oword [Key_Schedule_128+16], xmm1
                    ; uložení RK1 do pole expandovaného klíče
```

Dokončením ($xmm1 \oplus xmm2$) je získán rundovní klíč $RK_1$. Matematicky popsáno v rovnici (5.5).

$$RK_1 = xmm_1 \oplus \mathrm{xmm}_2 = \\ \begin{bmatrix} RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \\ RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \\ RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \\ RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \end{bmatrix} \oplus \begin{bmatrix} W_0 \\ W_0 \oplus W_1 \\ W_0 \oplus W_1 \oplus W_2 \\ W_0 \oplus W_1 \oplus W_2 \oplus W_3 \end{bmatrix} = \\ = \begin{bmatrix} RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \oplus W_0 \\ RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \\ RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \oplus W_2 \\ RotWord(SubWord(W_3)) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \oplus W_2 \oplus W_3 \end{bmatrix} \tag{5.5}$$

Tento postup se opakuje pro odvození dalších rundovních klíčů. Vstupem pro odvození dalšího rundovního klíče je klíč předchozí. Pro následující klíče se používají další rundovní konstanty (Tab. 3.3).

### 5.2.2 Implementace u AES-192

Jelikož je velikost xmm registrů 128 bitů, expanze klíče AES-192 je složitější, než expanze klíče u AES-128. V případě AES-192 má expandovaný klíč velikost 52 dvojslov. Prvních šest dvojslov ($W_0$ až $W_6$) tvoří šifrovací klíč. Další šestice dvojslov jsou odvozovány z předchozích šesti dvojslov dle algoritmu expanze klíče popsaného v kapitole 4.3. Celkový expandovaný klíče je tvořen šifrovacím klíčem (6 dvojslov), potom sedmkrát šesticí dvojslov odvozených z předchozích šestic dvojslov a poté se odvozují čtyři dvojslova. Výsledkem je výše zmíněných 52 dvojslov.

**Tab. 5.5: Expandovaný klíč AES-192.**

| $W_0$ | $W_1$ | $W_2$ | $W_3$ | $W_4$ | $W_5$ | $W_6$ | $W_7$ | $W_8$ | $W_9$ | $W_{10}$ | $W_{11}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |
| | | | | | | | | | | | |

0. rundovní klíč 1. rundovní klíč 2. rundovní klíč

Šifrovací klíč

V rovnici (5.6) je odvozen obsah dvojslov prvního rundovního klíče.

$$\begin{bmatrix} W_6 \\ W_7 \\ \dots \\ W_{11} \end{bmatrix} = \begin{bmatrix} W_5' \oplus W_0 \\ W_6 \oplus W_1 \\ \dots \\ W_{10} \oplus W_5 \end{bmatrix} = \begin{bmatrix} RotWord(SubWord(W_5)) \oplus \mathrm{RCon}(1) \oplus W_0 \\ RotWord(SubWord(W_5)) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \\ \dots \\ RotWord(SubWord(W_5)) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \oplus \dots \oplus W_5 \end{bmatrix} \quad (5.6)$$

#### *Postup implementace*

Stejně jako u AES-128 nejprve překopírujeme šifrovací klíč na začátek pole s expandovaným klíčem.

**Kód 5.3: Nahrání klíče do xmm registru a uložení do pole.**

```
MOVQ xmm2, [Key+16]
                ; přesune nejvyšší dvě dvojslova šifrovacího
                ; klíče do xmm2 registru (W4 a W5)
MOVDQU xmm1, [Key]
                ; přesune nejnižší čtyři dvojslova šifrovacího
                ; klíče do xmm1 registru (W0, W1, W2, W3)
MOVDQU [Key_Schedule_192], xmm1
MOVDQU [Key_Schedule_192+16], xmm2
                ; uložení klíče na začátek pole
```

Dále pomocí AESKEYGENASSIST vytvoříme $RotWord(SubWord(W_5)) \oplus \mathrm{RCon}(1)$. Pro právě takový výstup je potřeba, aby zdrojový operand instrukce AESKEYGENASSIST měl na pozici X3 poslední doubleword šifrovacího klíče ($W_5$). K tomu se využije instrukce PSHUFD. Výstupní doubleword $RotWord(SubWord(W_5)) \oplus \mathrm{RCon}(1)$ poté pomocí instrukce PSHUFD vložíme do všech dvojslov registru xmm.

Kód 5.4: Vytvoření RotWord(SubWord($W_5$) ⊕ RCon(1).

```
                   ; xmm2[X1] = W5
PSHUFD xmm1, xmm2, 01001110b
                   ; xmm1[X3] ← xmm2[X1]
                   ; xmm1[X2] ← xmm2[X0]
                   ; xmm1[X1] ← xmm2[X3]
                   ; xmm1[X0] ← xmm2[X2]
AESKEYGENASSIST xmm3,xmm1,0x1
                   ; xmm3[X3] = RotWord(SubWord(W5)) ⊕ RCon(1)
PSHUFD xmm3, xmm3, 0xff
                   ; s maskou FF = 11111111 obmění všechny dvojslova
                   ; X0 až X3 registru xmm3 na hodnotu v xmm3[X3]
                   ; xmm3[X0-X3] = RotWord(SubWord(W5))⊕ RCon(1)
```

Vytvoření $W_6$ a $W_7$. Na rozdíl od AES-128, kde další klíče se dal odvodit najednou, se u AES-192 další klíč odvozuje po dvou částech. Je to dané velikostí xmm registrů, která činí 128 bitů – 4 dvojslova. Nejprve se odvodí první dvě dvojslova dalšího klíče ($W_6$ a $W_7$) a poté se odvodí zbylé čtyři dvojslova ($W_8$ až $W_{11}$).

Kód 5.5: Odvození dvojslov $W_6$ a $W_7$.

```
MOVQ xmm1, [Key_Schedule_192]
                   ; nahrání prvních dvou dvojslov klíče do registru
                   ; xmm1=[0,0,W1,W0]
VPSLLDQ xmm2, xmm1, 0x4
                   ; posunutí obsahu xmm1 registru o 4 byty
                   ; xmm2=[0,W1,W0,0]
PXOR xmm1, xmm2
                   ; xmm1=[0,W1,W1⊕ W0,W0]
PSHUFD xmm1, xmm1, 11110100b
                   ; vynulování xmm1[X2] obměněním dvojslov
                   ; xmm1=[0,0,W1⊕ W0,W0]
PSHUFD xmm5, xmm1, 01010101b
                   ; příprava registru obsahující ve všech
                   ; dvojslovech W1⊕ W0
                   ; tento registr bude využitý až při odvozování
                   ; dvojslov W8 až W11.
PXOR xmm1, xmm3
                   ; vytvoří první dvě dvojslova dalšího klíče W6 a W7
                   ; xmm1[X0]=RotWord(SubWord(W5))⊕ RCon(1)⊕ W0
                   ; xmm1[X1]=RotWord(SubWord(W5))⊕ RCon(1) ⊕ W0 ⊕ W1
MOVQ [Key_Schedule_192+24], xmm1
                   ; uložení W6 a W7 do pole s klíči
```

Postup odvození dvojslov $W_8$ až $W_{11}$ je stejný, jak postup u odvozování klíčů u AES-128, s jedním rozdílem. K registru se po sekvenci posouvání dvojslov a následných operací XOR ještě „přiXORuje“ obsah registru $xmm_5$, jenž obsahuje hodnotu $W_0 \oplus W_1$ ve všech dvojslovech.

**Kód 5.6: Odvození dvojslov $W_8$ až $W_{11}$.**

```
MOVDQU xmm1, [Key_Schedule_192+8]
                ; nahrání W2 až W5 do registru xmm1
                ; následuje stejná sekvence operací jako při expanzi
                ; AES-128
     ; sekvence posouvání dvojslov
VPSLLDQ xmm4, xmm1, 0x4
PXOR xmm1, xmm4
VPSLLDQ xmm4, xmm1, 0x4
PXOR xmm1, xmm4
VPSLLDQ xmm4, xmm1, 0x4
PXOR xmm1, xmm4
PXOR xmm1, xmm3
          ; zde končí zmíněná sekvence
          ; xmm3 obsahuje RotWord(SubWord(W5))⊕RCon(1) ve všech
          ; dvojslovech
     ; xmm1[X0]=RotWord(SubWord(W5))⊕RCon(1) ⊕W2
     ; xmm1[X1]=RotWord(SubWord(W5))⊕RCon(1) ⊕W2 ⊕W3
     ; xmm1[X2]=RotWord(SubWord(W5))⊕RCon(1) ⊕W2 ⊕W3 ⊕W4
     ; xmm1[X3]=RotWord(SubWord(W5))⊕RCon(1) ⊕W2 ⊕W3 ⊕W4 ⊕W5
PXOR xmm1, xmm5
                ; přidáním xmm5=[W0 ⊕W1,W0 ⊕W1,W0 ⊕W1,W0 ⊕W1]
                ; vzniknou poslední 4 doublewordy W8 až W11
MOVDQU [Key_Schedule_192+32], xmm1
                ; uložení výsledku do pole s klíči
```

Postupem právě popsaným se odvozuje sedmkrát šest dvojslov. Po odvození těchto dvojslov se odvozují poslední čtyři dvojslova. Při jejich odvozování se postupuje stejně, jako při odvozování klíče se čtyřmi dvojslovy u AES-128, viz Kód 5.2.

### 5.2.3 Implementace u AES-256

V případě AES-256 má celkový expandovaný klíč 60 dvojslov. Klíče se odvozují po osmi dvojslovech. Prvních osm dvojslov je tvořeno samotných šifrovacích klíčem, poté je odvozeno dalších šest osmic a nakonec jsou odvozeny poslední čtyři dvojslova. Jak je napsáno v kapitole 3.2 o expanzi klíče, u AES-256 se používá navíc operace $SubWord\ (W[i])$ pro dvojslova, které jsou násobkem čísla 4 a nejsou násobkem čísla 8.

**Tab. 5.6: Expandovaný klíč AES-256.**

| $W_0$ | $W_1$ | $W_2$ | $W_3$ | $W_4$ | $W_5$ | $W_5$ | $W_7$ | $W_8$ | $W_9$ | $W_{10}$ | $W_{11}$ | $W_{12}$ | $W_{13}$ | $W_{14}$ | $W_{15}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| | | | | | | | | | | | | | | | |
| 0. rundovní klíč | | | | 1. rundovní klíč | | | | 2. rundovní klíč | | | | 3. rundovní klíč | | | |

Šifrovací klíč

Expanze probíhá tak, že se prvních 128 bitů klíče nahraje do registru $xmm_1$ a druhých 128 bitů do registru $xmm_2$. Prvních 128 bitů se odvodí stejně jako u AES-128. Pomocí instrukce AESKEYGENASSIST se odvodí $RotWord(SubWord(W_7)) \oplus \mathrm{RCon}(1)$ (zdrojový operand je $xmm_2$). Poté se pomocí kombinace instrukcí PSSLDQ a PXOR dojde k výsledku, který je v rovnici (5.7).

$$\begin{bmatrix} W_8 \\ W_9 \\ W_{10} \\ W_{11} \end{bmatrix} = \begin{bmatrix} W_7' \oplus W_0 \\ W_8 \oplus W_1 \\ W_9 \oplus W_2 \\ W_{10} \oplus W_3 \end{bmatrix} = \begin{bmatrix} RotWord(SubWord(W_7)) \oplus \mathrm{RCon}(1) \oplus W_0 \\ RotWord(SubWord(W_7)) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \\ RotWord(SubWord(W_7)) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \oplus W_2 \\ RotWord(SubWord(W_7)) \oplus \mathrm{RCon}(1) \oplus W_0 \oplus W_1 \oplus W_2 \oplus W_3 \end{bmatrix} \quad \textbf{(5.7)}$$

Při odvozování dalších 128 bitů se do postupu vkládá operace SubWord(). Doubleword $W_{12}$ je na pozici dělitelné 4 a nedělitelné 8, proto se před operací $W_{12} = W_{11} \oplus \mathrm{W}_4$ aplikuje operace $SubWord(W_{11})$. Matematické odvození dalších 128 bitů je znázorněno v rovnici (5.8). Pro získání hodnoty $SubWord\ (W[i])$ je využito instrukce AESKEYGENASSIST, viz Pseudokód 4.3.

$$\begin{bmatrix} W_{12} \\ W_{13} \\ W_{14} \\ W_{15} \end{bmatrix} = \begin{bmatrix} W_{11}' \oplus W_4 \\ W_{12} \oplus W_5 \\ W_{13} \oplus W_6 \\ W_{14} \oplus W_7 \end{bmatrix} = \begin{bmatrix} SubWord(W_{11}) \oplus W_4 \\ SubWord(W_{11}) \oplus W_4 \oplus W_5 \\ SubWord(W_{11}) \oplus W_4 \oplus W_5 \oplus W_6 \\ SubWord(W_{11}) \oplus W_4 \oplus W_5 \oplus W_6 \oplus W_7 \end{bmatrix} \quad \textbf{(5.8)}$$

Poslední 4 dvojslova (128 bitů) expandovaného klíče se odvozuje podle stejného postupu, jako se odvozují klíče u AES-128.

### 5.2.4 Odvození dešifrovacích klíčů

Další instrukcí popsanou v kapitole 4.3 je instrukce AESIMC a používá se k odvození dešifrovacích klíčů ze šifrovacích. Instrukce pro dešifrování AESDEC funguje ve smyslu ekvivalentní inverzní šifry, tzn. klíče pro dešifrování se používají v obráceném pořadí, než klíče pro šifrování a první a poslední klíč zůstává stejný. Odvozuje se tedy 9 klíčů u AES-128, 11 klíčů u AES-192 a 13 klíčů AES-256.

#### *Postup implementace*

Postup implementace je jednoduchý, nejdůležitější částí je tento kód.

**Kód 5.7: Smyčka odvozující dešifrovací klíče.**

```
; Smyčka pro odvození dešifrovacího klíče.
; Registr edx je použit pro adresování pole šifrovacích klíčů,
; registr eax je použit pro adresování pole dešifrovacích klíčů.
; N = 9 pro AES-128, N = 11 pro AES-192, N = 13 pro AES-256.
.repeat_N_times:
            MOVDQU xmm1, oword [edx]
                 ; nahrání šifrovacího klíče
            AESIMC xmm1, xmm1
                 ; provedení operace InvColumnMix nad šifr. klíčem
            MOVDQU oword [eax], xmm1
```

```
                ; uložení dešifrovacího klíče
            ADD edx, 0x10
            ADD eax, 0x10
                ; posunutí ukazatelů na pole klíčů o 128 bitů
        LOOP .repeat_N_times
```

## 5.3 Šifrování a dešifrování AES

Šifrování a dešifrování AES se provádí po expanzi klíčů. Nyní, když máme expandované šifrovací klíče a odvozené dešifrovací klíče, můžeme šifrovat vlastní data (vždy po 128 bitových blocích). K šifrování slouží instrukce AESENC a AESENCLAST a k dešifrování AESDEC a AESDECLAST. Následující kód provádí šifrování AES pro velikost klíče 128 bitů.

**Kód 5.8: Šifrování AES-128.**

```
MOVDQU xmm0, [PlainText]
            ; nahrání 128 bitů čistého textu do registru xmm0
PXOR xmm0, [Key_Schedule_128+16*0]
            ; provedení inicializační (nulté) rundy
            ; použit nultý klíč, který odpovídá šifrovacímu klíči
; vykonání (N-1) šifrovacích rund
AESENC xmm0, [Key_Schedule_128+16*1]
AESENC xmm0, [Key_Schedule_128+16*2]
AESENC xmm0, [Key_Schedule_128+16*3]
AESENC xmm0, [Key_Schedule_128+16*4]
AESENC xmm0, [Key_Schedule_128+16*5]
AESENC xmm0, [Key_Schedule_128+16*6]
AESENC xmm0, [Key_Schedule_128+16*7]
AESENC xmm0, [Key_Schedule_128+16*8]
AESENC xmm0, [Key_Schedule_128+16*9]

AESENCLAST xmm0, [Key_Schedule_128+16*10]
            ; provedení poslední N-té rundy pomocí instrukce
            ; AESENCLAST
MOVDQU [CipherText], xmm0
            ; uložení šifrovaného textu do paměti
```

Kód pro ostatní velikosti klíčů je obdobný. Rozdíl je v počtu iterací (rund) a v poli $Key_Schedule$, které obsahuje expandovaný klíč Následující kód provádí dešifrování AES pro velikost klíče 128 bitů.

**Kód 5.9: Dešifrování AES-128.**

```
MOVDQU xmm0, [CipherText]
            ; nahrání šifrovaného textu do registru xmm0
PXOR xmm0, [Key_Schedule_Decrypt_128+16*10]
            ; provedení inicializační rundy
; vykonání (N-1) dešifrovacích rund
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*9]
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*8]
```

```
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*7]
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*6]
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*5]
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*4]
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*3]
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*2]
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*1]

AESDECLAST xmm0, [Key_Schedule_Decrypt_128+16*0]
            ; provedení poslední N-té rundy pomocí instrukce
            ; AESDECLAST
MOVDQU [PlainText], xmm0
            ; uložení šifrovaného textu do paměti
```

Stejně jako u šifrování, kód pro ostatní velikosti klíčů je obdobný. Rozdíl je opět v počtu iterací (rund) a v poli $Key_Schedule_Decrypt$. V kódu pro dešifrování si lze všimnout obráceného pořadí odvozených klíčů. Toto pořadí odpovídá dešifrování ve smyslu ekvivalentní inverzní šifry, viz Obr. 3.2.

## 5.4 Operační módy AES

V rámci práce je implementováno pět operačních módů blokových šifer. Jsou to módy ECB, CBC, CFB, OFB a CTR. Tyto módy jsou popsané v kapitole 2.2 podle [1]. Módy jsou definovány (šifrování a dešifrování) podle [8]. Funkce implementující operační módy byly vyzkoušeny pomocí hodnot obsažených v [8] (příloha F, příkladové vektory).

### 5.4.1 Mód ECB

V módu elektronické kódová kniha (Electronic Codebook, ECB) se jednotlivé vloky šifrují nezávisle na sobě. Jak je popsáno v kapitole 2.2.1, každý blok čistého textu je zašifrovaný stejným klíčem a pro dva stejné bloky čistého textu je výstupem stejný šifrovaný text. V rovnici (5.9) je definováno ECB šifrování a dešifrování.

$$\begin{aligned} &\text{ECB šifrování:} \quad && C_i = E(P_i, K) \\ &\text{ECB dešifrování:} \quad && P_i = D(C_i, K) \end{aligned} \tag{5.9}$$

Kód 5.8 a Kód 5.9 implementuje šifrování a dešifrování módu ECB.

### 5.4.2 Mód CBC

V mód řetězení šifrovaného textu (Cipher Block Chaining, CBC) je právě šifrovaný (dešifrovaný) blok závislý na předchozím bloku šifrovaného textu. Pro první šifrovaný i dešifrovaný blok se používá inicializační vektor. Více o CBC módu v kapitole 2.2.2. V rovnici (5.10) je definováno CBC šifrování a dešifrování.

CBC šifrování: $C_1 = E(P_1 \oplus IV, K)$

$C_i = E(P_i \oplus C_{i-1}, K)$ pro $i = 2 \dots n$

CBC dešifrování: $P_1 = D(C_1, K) \oplus IV$

$P_i = D(C_i, K) \oplus C_{i-1}$ pro $i = 2 \dots n$

**(5.10)**

Implementace módu CBC je v následujícím kódu.

**Kód 5.10: Implementace CBC módu pro velikosti klíče 128 bitů.**

```
; Šifrování CBC
MOVDQU xmm1, [PlainText]
MOVDQU xmm0, [PreviousCipherText]
PXOR xmm0, xmm1
          ; xmm0 = Pi ⊕ Ci-1
; následuje šifrování AES dle postupu popsaného v kódu 5.8
PXOR xmm0, [Key_Schedule_128+16*0]          ; inicializační runda
AESENC xmm0, [Key_Schedule_128+16*1]        ; první runda
          ...
AESENC xmm0, [Key_Schedule_128+16*9]        ; devátá runda
AESENCLAST xmm0, [Key_Schedule_128+16*10]   ; poslední runda
          ; xmm0 = E(Pi ⊕ Ci-1, K) = Ci
MOVDQU [CipherText], xmm0
          ; uložení šifrovaného textu do paměti

; Dešifrování CBC
MOVDQU xmm1, [PreviousCipherText]
MOVDQU xmm0, [CipherText]
          ; nahrání šifrovaného textu a předchozího šifrovaného
          ; textu do registrů
; následuje dešifrování AES dle postupu popsaného v kódu 5.9
PXOR xmm0, [Key_Schedule_Decrypt_128+16*10]
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*9]
          ...
AESDEC xmm0, [Key_Schedule_Decrypt_128+16*1]
AESDECLAST xmm0, [Key_Schedule_Decrypt_128+16*0]
          ; xmm0 = D(Ci, K)
PXOR xmm0, xmm1
          ; xmm0 = D(Ci, K) ⊕ Ci-1 = Pi
MOVDQU [PlainText], xmm0
          ; uložení čistého textu do paměti
```

### 5.4.3 Mód CTR

V módu čítače (Counter, CTR) se čistý text nešifruje. Místo čistého textu se šifruje hodnota čítače a výsledný šifrovaný text je pak výsledkem operace XOR nad čistým textem a zašifrovanou hodnotou čítače, jak je definováno v rovnici (5.11). Mód čítače je podrobněji popsaný v kapitole 2.2.5.

$$\text{CTR šifrování:} \quad C_i = P_i \oplus E(CTR_i, K)$$
$$\text{CTR dešifrování:} \quad P_i = C_i \oplus E(CTR_i, K) \tag{5.11}$$

**Kód 5.11: Implementace CTR módu pro velikost klíče 128 bitů.**

```
; Šifrování CTR
MOVDQU xmm1, [PlainText]
MOVDQU xmm0, [Counter]
           ; nahrání čistého textu a hodnoty čítače do registrů xmm
; následuje šifrování AES dle postupu popsaného v kódu 5.8
PXOR xmm0, [Key_Schedule_128+16*0]
AESENC xmm0, [Key_Schedule_128+16*1]
           ...
AESENC xmm0, [Key_Schedule_128+16*9]
AESENCLAST xmm0, [Key_Schedule_128+16*10]
           ; zašifrování hodnoty čítače
           ; xmm0 = E(CTRi,K)
PXOR xmm0, xmm1
           ; xmm0 = Pi ⊕ E(CTRi,K) = Ci
MOVDQU [CipherText], xmm0
           ; uložení šifrovaného textu do paměti

; Dešifrování CTR
MOVDQU xmm1, [CipherText]
MOVDQU xmm0, [Counter]
           ; nahrání šifrovaného textu a hodnoty čítače do registrů
; následuje dešifrování AES dle postupu popsaného v kódu 5.9
PXOR xmm0, [Key_Schedule_128+16*0]
AESENC xmm0, [Key_Schedule_128+16*1]
           ...
AESENC xmm0, [Key_Schedule_128+16*9]
AESENCLAST xmm0, [Key_Schedule_128+16*10]
           ; xmm0 = D(CTRi,K)
PXOR xmm0, xmm1
           ; xmm0 = Ci ⊕ D(CTRi,K) = Pi
MOVDQU [PlainText], xmm0
           ; uložení čistého textu do paměti
```

### 5.4.4 Mód CFB

V módu zpětné vazby ze šifrovaného textu je možné pracovat s bloky dat o menší velikosti než 128 bitů. Vstupní i výstupní blok má stále velikost 128 bitů, ale pro šifrování se používá $s$ nejvíce významných bitů výstupního toku, které se „XORují" s $s$ bity čistého textu. Zpětná vazba je tvořena $s$ bity šifrovaného textu, které se vládají na $s$ nejméně významných bitů vstupního bloku. Vstupní blok je tedy tvořen posuvným registrem. CFB mód je více popsán v kapitole 2.2.3. V rovnici (5.12) je definován CFB mód.

$$
\begin{aligned}
&\text{CFB šifrování:} && I_i = IV && \text{pro } i = 1 \\
& && I_i = LSB_{b-s}(I_{i-1})|C_{i-1} && \text{pro } i = 2 \ldots n \\
& && C_i = P_i \oplus E(I_i, K) && \\
&\text{CFB dešifrování:} && I_i = IV && \text{pro } i = 1 \\
& && I_i = LSB_{b-s}(I_{i-1})|C_{i-1} && \text{pro } i = 2 \ldots n \\
& && P_i = C_i \oplus E(I_i, K) &&
\end{aligned}
\quad \textbf{(5.12)}
$$

Při implementaci CFB módu byla největší překážka implementace 128 bitového posuvného registru. Instrukce PSLLDQ a PSLLQ, které jsou popsané v kapitole 5.1, neumožňují jednoduchou implementaci 128 bitového posuvného registru. Instrukce PSLLDQ posouvá v rámci celého 128 bitového xmm registru o zadaný počet bytů. Pomocí této instrukce není možné implementovat posuvný registr pro všechny velikosti $s \in (1, 128)$ bitů, ale jen pro $s = k \cdot 8$ $(k = 1,2, \ldots 16)$ bitů (násobky bytů). Instrukce PSLLQ posouvá o zadaný počet bitů, ale v rámci jednoho čtyřslova (64 bitů). Při posunutí doleva se nejvíce významné bity prvního čtyřslova nepřenášejí na nejméně významné bity druhé čtyřslova. Stejně tak to platí u posunu doprava (PSRLQ), kde se nejméně významné bity druhého čtyřslova nepřesouvají na nejvíce významné bity prvního čtyřslova.

Při implementaci 128 bitového registru je použita instrukce SHLD a SHRD popsaná v kapitole 5.1.6 a je rozdělena na dvě části. V první části se posunují bity vstupního bloku o $s$ bitů doprava (k méně významným bitům) a v druhé části se posunují bity šifrovaného textu o $128 - s$ bitů doleva (k významnějším bitům). Nakonec se provede operace XOR nad posunutým vstupním blokem a posunutým šifrovaným textem a tím vznikne nový vstupní blok.

**Kód 5.12: Šifrování CFB.**

```
MOVDQU xmm0, [InputBlock]
          ; nahrání vstupního bloku do registru
PXOR xmm0, [Key_Schedule_128+16*0]
AESENC xmm0, [Key_Schedule_128+16*1]
          ...
AESENC xmm0, [Key_Schedule_128+16*9]
AESENCLAST xmm0, [Key_Schedule_128+16*10]
          ; xmm0 obsahuje zašifrovaný vstupní blok
MOVDQU [OutputBlock], xmm0
          ; výstupní blok uložen do paměti
; Doposud je kód stejný pro šifrování i dešifrování CFB

; Šifrování CFB
MOVDQU xmm1, [PlainText]
          ; nahrání čistého textu do registru
PXOR xmm0, xmm1
          ; v xmm0 je šifrovaný text
          ; xmm0 = Ci = Pi ⊕ E(Ii, K)
MOVDQU [CipherText], xmm0
```

```
            ; uložení šifrovaného textu do paměti

; Dešifrování CFB
MOVDQU xmm1, [CipherText]
            ; nahrání šifrovaného textu do registru
PXOR xmm0, xmm1
            ; v xmm0 je šifrovaný text
            ; xmm0 = Pi = Ci ⊕ E(Ii, K)
MOVDQU [PlainText], xmm0
            ; uložení čistého textu do paměti
```

Jelikož se stále pracuje se 128 bitovými registry, v tuto chvíli registr xmm0 obsahuje $s$ bitů šifrovaného textu (na $s$ nejméně významných bitech) a $128 - s$ bitů výstupního bloku.

Následuje vytvoření nového vstupního bloku, který se zašifruje a použije při šifrování dalšího čistého textu. Je-li $s = 128$, celý šifrovaný text uložený v xmm0 tvoří nový vstupní blok.

**Kód 5.13: Vytvoření nového vstupní bloku pro s = 128.**

```
MOV eax, [s_bits]
CMP eax, 128
            ; porovnání s_bits s velikostí 128
JE .end_128
            ; je-li s_bits = 128, přeskočí vytváření vstupního bloku
            ; pro s = (1,127)
.tvorba_noveho_vstupniho_bloku:
        …
.end_128:
        MOVDQU [InputBlock], xmm0
            ; uložení šifrovaného textu do paměti
```

Je-li $s \in (1, 127)$, pak nový vstupní blok je tvořen $s$ bity šifrovaného textu a $128 - s$ bity vstupního bloku. Implementace vytváření nového vstupního bloku je v kódu rozdílná pro tři intervaly, do kterých $s$ patří. Tyto tři intervaly jsou rozdílné pro posouvání doprava (posouvání vstupního bloku) a pro posouvání doleva (posouvání šifrovaného textu). Postupy jsou popsané pro každý z intervalů v následujících tabulkách.

**Tab. 5.7: Postup implementace při posouvání vstupního bloku.**

| s | Postup implementace |
|---|---|
| $(1, 31)$ | 1. Posunutí o $s$ bitů pomocí instrukce PSRLQ (ztráta bitů)<br>2. Obnovení bitů z druhého a třetího dvojslova |
| $(32, 63)$ | 1. Posunutí o 32 bitů pomocí instrukce PSRLDQ (bez ztráty bitů)<br>2. Posunutí o $s - 32$ bitů pomocí instrukce PSRLQ (ztráta bitů)<br>3. Obnovení bitů z prvního a druhého dvojslova |
| $(64, 127)$ | 1. Posunutí o 64 bitů pomocí instrukce PSRLDQ (bez ztráty bitů)<br>2. Posunutí o $s - 32$ bitů pomocí instrukce PSRLQ (bez ztráty bitů) |

Tab. 5.8: Postup implementace při posouvání šifrovaného textu.

| s | Postup implementace |
|---|---|
| $(0, 63)$ | 1. Posunutí o 64 bitů pomocí instrukce PSLLDQ (bez ztráty bitů)<br>2. Posunutí o $64 - s$ bitů pomocí instrukce PSLLQ (bez ztráty bitů) |
| $(64, 95)$ | 1. Posunutí o 32 bitů pomocí instrukce PSLLDQ (bez ztráty bitů)<br>2. Posunutí o $96 - s$ bitů pomocí instrukce PSLLQ (ztráta bitů)<br>3. Obnovení bitů z prvního a druhého dvojslova |
| $(96, 127)$ | 1. Posunutí o $s\ mod$ 32 bitů pomocí instrukce PSLLQ (ztráta bitů)<br>2. Obnovení bitů z druhého a třetího dvojslova |

Pro určování intervalu je použito instrukce pro dělení DIV. Instrukce DIV má jeden argument, jímž je dělitel. Dělitelem dělí dělenec uložený v registru EAX. Po vykonání instrukce je v EAX uložen kvocient a v registru EDX zbytek po dělení. Registr EDX je nutné před dělením vynulovat.

Kód 5.14: Dělení hodnoty *s*.

```
MOV eax, [s_bits]        ; nahrání hodnoty s do eax – dělenec
XOR edx, edx             ; vynulování edx
MOV ecx, 32              ; nahrání dělitele
DIV ecx

MOV [Quotient], eax      ; uložení kvocientu do paměti
MOV [Remainder], edx     ; uložení zbytku po dělení do paměti
```

Jelikož je instrukce DIV značně pomalá, dělení hodnoty *s* se provádí ve funkci, ve které se hodnota *s* nastavuje (ukládá do paměti), a neprovádí se v samotné funkci určené pro šifrování (a dešifrování) CFB. Funkce nastavující hodnotu *s* bude popsána v kapitole 5.5.5. Pomocí hodnoty kvocientu, instrukce CMP na porovnávání hodnot a podmíněných skoků se určí do jakého intervalu *s* patří a jaký kód se vykoná.

Kód 5.15: Řízení toku program podle hodnoty kvocientu.

```
MOV edx, [Remainder]
                    ; nahrání zbytku po dělení do edx
MOV eax, [Quotient]
                    ; nahrání kvocientu z paměti do registru eax
CMP eax, 0
JE .s1_31           ; skok na kód pro interval 1-31
CMP eax, 1
JE .s32_63          ; skok na kód pro interval 32-63
 CMP eax, 2
JGE .s64_127        ; skok na kód pro interval 64-127
```

Nyní bude popsán kód posouvání vstupního bloku a šifrovaného textu pro $s \in (1, 31)$. Kód implementující posuny pro ostatní intervaly zde popsán nebude, neboť je dlouhý. Pro celý kód posouvání viz zdrojový kód knihovny AES.

**Kód 5.16: Vytvoření vstupní bloku pro s ∈ (1, 31).**

```
; První část – posunutí vstupního bloku.
.shifting_InputBlock:
            ; zde se provádí Kód 5.15
.s1_31:
MOV [Offset], edx
            ; uložení edx = s mod 32 ( = s pro s ∈ (1,31)) do paměti
MOVDQU xmm1, [InputBlock]
            ; nahrání vstupního bloku do registru xmm1
PSRLQ xmm1, [Offset]
            ; posunutí vstupního bloku doprava o s bitů,
            ; bity přenášející se přes hranici čtyřslova (64. bit)
            ; se ztrácejí – jedná se o s nejméně významných bitů
            ; třetího dvojslova, které poté chybí ve druhém dvojslově
MOV eax, [InputBlock+4]
            ; nahrání druhého dvojslova vst. bloku do registru eax
            ; (číslováno od LSB 1., 2., 3., 4. dvojslovo)
MOV edx, [InputBlock+8]
            ; nahrání třetího dvojslova vst. bloku do registru edx
MOVDQU [InputBlock], xmm1
            ; uložení posunutého vstupního bloku do paměti
            ; v druhém dvojslově chybí s bitů z třetího dvojslova
MOV cl, [Offset]
            ; nahrání hodnoty s do registru cl (8 bit. registr)
SHRD eax, edx, cl
            ; posunutí cílového registru o cl bitů doprava, na cl
            ; nejvíce významných bitů cílového registru přemístí cl
            ; nejméně významných bitů zdrojového registru
MOV dword [InputBlock+4], eax
            ; uložení druhého dvojslova s bity z třetího dvojslova
MOVDQU xmm1, [InputBlock]
            ; uložení posunutého vstupního bloku o s bitů do registru
JMP  .adding_CipherText
            ; skok na druhou část vytvoření vstupního bloku – přidání
            ; posunutého šifrovaného textu
.s64_95:
      …     ; provedení posunu vstupního bloku pro s ∈ (64,95)
      JMP  .adding_CipherText

.s96_127:
      …     ; provedeni posunu vstupního bloku pro s ∈ (96,127)
      ;JMP  .adding_CipherText

; Druhá část – posunutí šifrovaného textu.
; V registru xmm0 je uložen šifrovaný text.
.adding_CipherText:
            ; zde se také provádí řízení toku programu ve smyslu
```

```
            ; předchozího kódu (Kód 5.15)
.s0_63:
MOV edx, 64
SUB edx, [s_bits]
            ; edx = 64 - s_bits
MOV [Offset], edx
            ; uložení edx do paměti
PSLLDQ xmm0, 8
            ; posunutí o 64 bitů doleva – bez ztráty bitů
PSLLQ xmm0, [Offset]
            ; posunutí o hodnotu Offset doleva – bez ztráty bitů
            ; blok je posunut celkově o (64 + edx - s_bits) =
            ; (128 - s_bits) bitů
JMP .end
            ; skok na konec pro dokončení vytvoření nového vstupního
            ; bloku
.s64_95:
      …     ; provedeni posunu šifrovaného textu pro s ∈ (64,95)
      JMP .end
.s96_127:
      …     ; provedeni posunu šifrovaného textu pro s ∈ (96,127)
.end:
PXOR xmm0, xmm1
            ; vytvoření nového vstupního bloku pomocí operace XOR nad
            ; posunutým vstupním blokem o s bitů doprava a posunutým
            ; šifrovaným textem o 128-s bitů doleva
.end_128:
MOVDQU [InputBlock], xmm0
            ; uložení nového vstupního bloku do paměti
```

Podle definic (5.12), vstupní blok $I_i$ je tvořen stejným způsobem u šifrování i dešifrování.

### 5.4.5 Mód OFB

Mód zpětná vazby z výstupu dovoluje, stejně jako mód CFB, pracovat s bloky dat o velikosti menší než 128. Mód OFB je podobný módu CFB. Rozdíl mezi módy CFB a OFB je ve zpětné vazbě. V CFB je zpětná vazba tvořena *s* bity šifrovaného textu, v módu OFB je zpětná vazba tvořena *s* bity výstupního bloku. Vstupní blok je také 128 bitový registr. Mód OFB je definován podle (5.13).

$$
\begin{aligned}
&\text{OFB šifrování:} && I_i = IV && \text{pro } i = 1 \\
& && I_i = LSB_{b-s}(I_{i-1})|O_{i-1} && \text{pro } i = 2 \dots n \\
& && C_i = P_i \oplus E(I_i, K) && \\
&\text{OFB dešifrování:} && I_i = IV && \text{pro } i = 1 \\
& && I_i = LSB_{b-s}(I_{i-1})|O_{i-1} && \text{pro } i = 2 \dots n \\
& && P_i = C_i \oplus E(I_i, K) &&
\end{aligned}
\tag{5.13}
$$

Kód implementace OFB módu je skoro stejný, jako kód implementace módu CFB. Rozdíl je při tvorbě nového vstupního bloku, který je použit u šifrování dalšího čistého textu. Nový vstupní blok se tvoří opět na dvě části. V první části se stejně jako v módu CFB posouvá vstupní blok. V druhé části se namísto šifrovaného textu (CFB mód) posouvá výstupní blok (zašifrovaný vstupní blok). Následně se obě hodnoty „XORují" a tím vznikne nový vstupní blok.

## 5.5 Vytvoření knihovny

V posledním bodě pokynů pro vypracování diplomové práce je vytvoření dynamické knihovny poskytující funkce implementující šifru AES pro velikost klíče 128, 192 a 256 bitů a také módy (režimy) této šifry.

### 5.5.1 Konvence volání funkcí

Volání funkcí se řídí konvencí stdcall, která je popsaná v [9] a [10]. Při popisování pravidel, jimiž se řídí volání budou používány následující pojmy: volající (funkce, která provádí volání – skript v Matlabu) a volaný (funkce, který je volána – funkce z knihovny aes.dll). Pravidla volání, podle [9], jsou:

1) Volající ukládá na zásobník parametry volané funkce jeden po druhém – u stdcall konvence zprava doleva (první parametr je uložen poslední a poslední parametr je uložen první).
2) Volající spouští instrukci CALL, čímž předá kontrolu volanému.
3) Volaný uloží obsah registru EBP (*Base Pointer*) na zásobník a zkopíruje obsah registru ESP (*Stack Pointer*) do registru EBP. EBP se použije jako bázový ukazatel pro práci se zásobníkem.
4) Volaný přistupuje k parametrům pomocí registru EBP od adresy [EBP+8]. Na adrese [EBP] je původní hodnota EBP a na adrese [EBP+4] je návratová adresa volání.
5) Volaný může snížit hodnotu ESP, čímž rezervuje místo pro lokální proměnné, které mají adresy se záporným přírůstkem od [EBP-4].
6) Volaný může vracet hodnotu volajícímu prostřednictvím registru EAX:EDX, EAX, AX nebo AL.
7) Po dokončení práce volaný obnoví obsah registru ESP z EBP a obnoví obsah registru EBP a provede návrat – v stdcall konvenci volaný provede úklid zásobníku.

Jak je uvedeno v [10], NASM používá dvě direktivy pro tvorbu knihoven – GLOBAL a EXPORT. Direktiva GLOBAL říká překladači, že funkce (nebo nějaký symbol) bude přístupná ostatním modulům při linkování programu. Direktiva EXPORT říká překladači, že funkce bude přístupná ostatním programům a bude součástí knihovny DLL.

V následujícím kódu je popsaná funkce pro nastavení velikosti klíče.

**Kód 5.17: Funkce pro nastavení velikosti klíče.**

```
GLOBAL key_size_setup          ; globální funkce
EXPORT key_size_setup          ; externí funkce
```

```
key_size_setup:
        PUSH dword ebp          ; uložení ebp na zásobník
        MOV dword ebp, esp      ; použití esp jako nový ebp

        MOV eax, [ebp+8]        ; přístup k prvnímu parametru
        MOV [KeySize], eax      ; uložení parametru do paměti

        MOV dword esp, ebp      ; obnovení esp z ebp
        POP dword ebp           ; obnovení ebp
RET 4                           ; úklid zásobníku - RET 4*počet parametrů
                                ; funkce
```

### 5.5.2 Kompilace a linkování knihovny

Ke kompilaci je použit překladač Netwide Assembler (NASM verze 2.09.10), který převede zdrojový kód *aesdll.asm* do objektového souboru *aesdll.obj*. K linkování linker ALINK (verze 1.6), který z objektového souboru *aesdll.obj* vytvoří dynamickou knihovnu *aes.dll*.

Volba -f obj specifikuje výstupní formát, -oPE specifikuje výstupní typ souboru (Win32 Portable Executable), pomocí -dll je vytvořena dynamická knihovna a volba -o specifikuje název výsledného souboru.

```
nams.exe aesdll.asm -f obj
alink.exe aesdll.obj -oPE -dll -o aes.dll
```

### 5.5.3 Volání funkcí z prostředí Matlab

Pro využívání a zkoušení funkcí knihovny bylo zvoleno prostředí Matlab. Postup použití knihovny je načtení knihovny do paměti, volání funkcí a nakonec vymazání knihovny z paměti.

Jak je uvedeno v [10], před samotným načtením knihovny je nutné vytvořit hlavičkový soubor, který obsahuje funkční prototypy všech funkcí v externích knihovně. Funkční prototypy jsou zapisovány ve formě:

```
návratová_hodnota konvence_volání název_funkce(vstupní parametry);
```

**Kód 5.18: Výpis z hlavičkového souboru aesdll.h.**

```
void _stdcall key_size_setup(int keySize);
void _stdcall key_expansion(uint *pKey);
int _stdcall encryption_cfb8(int plainText);
```

Pro celý seznam funkčních prototypů hlavičkového souboru *aesdll.h* viz přílohu B.

V prostředí Matlab se knihovny načítají pomocí funkce `loadlibrary`. Prvním argumentem této funkce je cesta ke knihovně, druhým argumentem je cesta k hlavičkovému souboru a třetí argument generuje prototypy načítaných funkcí.

K volání funkcí z načtené knihovny slouží funkce `calllib`. První argumentem této funkce je název knihovny, druhým argumentem je název funkce a další argumenty jsou proměnné, které budou předávány funkci.

Po ukončení práce se knihovna z paměti vymaže pomocí funkce `unloadlibrary`, jejíž jediným parametrem je název knihovny.

**Kód 5.19: Volání funkce z knihovny aes.dll v prostředí Matlab.**

```
% nastavené cesty k hlavičce knihovny
hfile=['aesdll.h'];
% načtení knihovny
loadlibrary('aesdll.dll',hfile,'mfilename','aes_ctr_m');
M = 4;         % udává velikost klíče v doublewordech
calllib('aesdll', 'key_size_setup', M);
               % zavolá funkci key_size_setup a předá ji parametr M = 4
% vymazání knihovny z paměti
unloadlibrary aesdll
```

### 5.5.4 Předávání parametrů

Šifra AES pracuje s bloky dat 128 bitů (nebo menšími u módů CFB a OFB). Proto se pro předávání 128 bitových řetězců v rozhraní Win32 musí použít ukazatelů na proměnné.

V prostředí Matlab se ukazatel na proměnnou vytvoří pomocí funkce `libpointer`. První argument této funkce je datový typ pointeru, druhý parametr je název proměnné. Předávané 128 bitové bloky dat jsou v prostředí Matlab definované jako pole čtyř 32 bitových hodnot typu uint32. Zapisujeme-li každé dvojslovo tak, že nejvíce levý byte odpovídá nejméně významnému bytu daného dvojslova, musíme použít funkci `swapbytes` (viz Kód 5.20), aby funkce převzala správnou hodnotu.

**Kód 5.20: Předávání hodnot při šifrování – část v prostředí Matlab.**

```
% PlainText = 00112233445566778899AABBCCDDEEFF,
% kde 00 je nejméně významný byte
PlainText = [
        swapbytes(uint32(hex2dec('00112233')))
        swapbytes(uint32(hex2dec('44556677')))
        swapbytes(uint32(hex2dec('8899AABB')))
        swapbytes(uint32(hex2dec('CCDDEEFF')))];
pPlainText = libpointer('uint32Ptr',PlainText);
               % vytvoření ukazatele na proměnnou PlainText
CipherArray=uint32([0 0 0 0]);
               % vytvoření pole CipherText
pCipherArray = libpointer('uint32Ptr',CipherText);
               % vytvoření ukazatele na proměnnou CipherText
M = 4;         % nastavení velikosti klíče 128 bitů
calllib('aesdll', 'key_size_setup', M);
               % funkce nastavující velikost klíče
calllib('aesdll', 'encryption_ecb', pPlainText, pCipherArray);
               % funkce provádějící šifrování
```

**Kód 5.21: Předávání hodnot při šifrování – část v zdrojovém kódu funkce.**

```
GLOBAL encryption_ecb
EXPORT encryption_ecb
```

```
encryption_ecb:
        PUSH dword ebp
        MOV dword ebp, esp
        ; nahrání čistého textu
        MOV edi, [ebp+8]
                ; nahrání ukazatele na čistý text do registru EDI
                ; (Destination Index)
        XOR edx, edx
        MOV ecx, 4
                ; vynulování EDX, nastavení ECX = 4
        ; nyní se nahrají 4 dvojslova čistého textu do paměti
        ; smyčka se provede 4x (hodnota v ECX)
        .loading_plaintext
                MOV dword eax, [edi+edx]
                MOV dword [PlainText+edx], eax
                add edx, 4
        loop .loading_plaintext
                ; při každém skoku zpět se dekrementuje ECX o 1
        ; nyní se rozhoduje podle velikosti klíče, jak se bude šifrovat
        MOV ecx, [KeySize]
                ; nahrání hodnoty KeySize do ECX, tato hodnota byla
                ; nastavena funkcí key_size_setup
        CMP ecx, 4
        JZ .enc128
        CMP ecx, 6
        JZ .enc192
        CMP ecx, 8
        JZ .enc256
.enc128:
        …       ; šifrování AES pro velikost klíče 128 bitů
        jmp .end
.enc192:
        …       ; šifrování AES pro velikost klíče 192 bitů
        jmp .end
.enc256:
        …       ; šifrování AES pro velikost klíče 256 bitů
.end:
        ; nyní se vrátí šifrovaný text volajícímu
        MOV esi, [ebp+12]
                ; nahrání ukazatele na šifrovaný text do registru ESI
                ; (Source Index)
        XOR edx, edx
        MOV ecx, 4
                        ; vynulování EDX, nastavení ECX = 4
        .returning_ciphertext:
                MOV dword eax, [CipherText+edx]
                MOV dword [esi+edx], eax
                ADD edx, 4
```

```
        loop .returning_ciphertext

        MOV dword esp, ebp
        POP dword ebp
RET 8
```

Následuje zobrazení hodnot v prostředí Matlab:

**Kód 5.22: Zobrazení šifrovaného textu v prostředí Matlab.**

```
CipherText_128=get(pCipherArray, 'Value');
CipherText_128=dec2hex(swapbytes(Ciphertext_128()))
```

### 5.5.5 Funkce knihovny

V této kapitole budou popsány funkce knihovny *aes.dll*. Jejich funkční prototypy jsou obsaženy v hlavičkovém souboru *aesdll.h*, jehož výpis je v příloze B. Funkce lze rozdělit do tří skupin:

1. funkce pro nastavení parametrů šifry a módů,
2. funkce pro expanzi šifrovacích klíčů a odvození dešifrovacích klíčů,
3. funkce pro šifrování a dešifrování v různým operačních módech.

Funkce pro nastavení parametrů šifry a módu slouží k uložení hodnot do paměti. Tyto hodnoty jsou používaný v dalších funkcích (bod 2. a 3.) a proto musí být nastaveny jako první. Funkce nevrací žádnou hodnotu (návratová hodnota typu `void`).

**Tab. 5.9: Funkce pro nastavování parametrů šifry a módů.**

| Název funkce | Vstupní parametr | Návratová hodnota |
|---|---|---|
| `key_size_setup` | `int keySize` | `void` |
| `initvector_setup` | `uint *pInitVector` | `void` |
| `s_bits_setup` | `int s` | `void` |

1. **`key_size_setup`** – funkci je předávána hodnota velikosti klíče v dvojslovech (4 pro AES-128, 6 pro AES-192, 8 pro AES-256). Hodnota je předávána vstupním parametrem `keySize` datového typu `int`. Tato hodnota je funkcí uložena do paměti – do proměnné `KeySize`. Hodnota `KeySize` je používaná pro rozlišení velikosti šifrovacího klíče při expanzi šifrovacího klíče, odvození dešifrovacích klíčů, šifrování a dešifrování.
2. **`initvector_setup`** – funkci je předáván 128 bitový inicializační vektor. Hodnoty jsou předávány pomocí ukazatele `*pInitVector` datového typu `uint` (32 bitové kladné číslo). Inicializační vektor je uložen do paměti – do proměnných PreviousCipherText (použití v módu CBC) a InputBlock (použití v módu CFB a OFB).
3. **`s_bits_setup`** – funkci je předávána velikost zpracovávaného bloku $s$. Tato hodnota se používá v rámci módu CFB a OFB. Hodnota je předávána vstupním parametrem $s$ datového typu `int`. Funkce uloží $s$ do paměti a zároveň provádí dělení $s \bmod 32$. Výsledky dělení (kvocient a zbytek po dělení) uloží do paměti jako

proměnné `Quotient` a `Remainder`. Tyto dvě hodnoty jsou používány v módu CFB a OFB při implementaci 128 bitového posuvného registru.

Následující funkce provádějí expanzi šifrovacího klíče a odvození dešifrovacích klíčů. Před voláním těchto funkcí musí být nastavena hodnota `KeySize`. Expanze klíčů a odvození šifrovacích klíčů musí být provedeno před vlastním šifrováním, resp. dešifrováním. Pole šifrovacích a dešifrovacích klíčů se ukládá do paměti. Funkce nevrací žádnou hodnotu.

**Tab. 5.10: Funkce pro expanzi šifrovacích klíčů a odvození dešifrovacích klíčů.**

| Název funkce | Vstupní parametr | Návratová hodnota |
|---|---|---|
| key_expansion | uint *pKey | void |
| key_expansion_dec | – | void |

1. **`key_expansion`** – funkci je předáván šifrovací klíč, který může mít velikosti 128, 192, nebo 256 bitů. Hodnoty jsou předávány pomocí ukazatele `*pKey` datového typu `uint`. Podle hodnoty v `KeySize` funkce nahrává daný počet dvojslov a rozhoduje o tom, pro jakou velikost klíče bude provedena expanze. Expandovaný klíč je uložen do paměti do odpovídajícího pole `Key_Schedule_128/192/256`.
2. **`key_expansion_dec`** – funkci není předávána žádná hodnota – před provedením této funkce je nutné provést expanzi šifrovacího klíče. Funkce odvozuje dešifrovací klíče ze šifrovacích klíčů, uložených polích `Key_Schedule_128/192/256`, a uloží je do odpovídajícího pole `Key_Schedule_Decrypt_128/192/256`.

Tab. 5.11 obsahuje funkce pro šifrování a dešifrování v různých operačních módech šifry AES. Jak bylo právě popsáno, před voláním těchto funkcí je potřeba provést nastavení potřebných parametrů šifry a módů a poté expandovat šifrovací klíče a odvodit dešifrovací klíče. Pro mód CFB a OFB jsou vytvořený tři funkce pro šifrování a k nim odpovídající funkce pro dešifrování. Funkce se liší ve velikosti zpracovávaného bloku $s$. Funkce končící cfb_8, cfb_128 a ofb_8 a ofb_128 jsou implementovány pro velikost $s = 8$, resp. $s = 128$. Funkce končící na cfb a ofb jsou implementovány pro velikost $s = \langle 1{,}128 \rangle$. Dále v práci bude popsán rozdíl v rychlosti mezi funkcemi `encryption_cfb8` a `encryption_cfb` pro $s = 8$.

**Tab. 5.11: Funkce pro šifrování a dešifrování v různých operačních módech šifry AES.**

| Název funkce | Vstupní parametr | Návratová hodnota |
|---|---|---|
| encryption_ecb<br>encryption_cbc<br>encryption_cfb128<br>encryption_ofb128 | uint *pPlainText<br>uint *pCipherText | void |
| decryption_ecb<br>decryption_cbc<br>decryption_cfb128<br>encryption_ofb128 | uint *pCipherText<br>uint *pPlainText | void |

| encryption_ctr | uint *pCounter<br>uint *pPlainText<br>uint *pCipherText | void |
|---|---|---|
| decryption_ctr | uint *pCounter<br>uint *pCipherText<br>uint *pPlainText | void |
| ecnryption_cfb<br>encryption_ofb | uint *pPlainText<br>uint *pCipherText<br>uint *pInputBlock | void |
| decryption_cfb<br>decryption_ofb | uint *pCipherText<br>uint *pPlainText<br>uint *pInputBlock | void |
| encryption_cfb8<br>encryption_ofb8 | int plainText | int |
| decryption_cfb8<br>decryption_ofb8 | int cipherText | int |

Funkce z Tab. 5.11 zjišťují velikost klíče pomocí proměnné `KeySize`. Na základě této hodnoty rozhodnou, zda-li budou šifrovat pro 128, 192, nebo 256 bitový klíč, tzn. kolik bude provedeno iterací (rund) a z jakého pole budou nahrávány šifrovací a dešifrovací klíče. Funkcím pro šifrování jsou předávány bloky čistého textu (až 128 bitů) pomocí ukazatele na proměnnou s čistým textem – vstupní parametr `uint *pPlainText`. Výstup je převzat taktéž pomocí ukazatele (`uint *pCipherText`). Proto funkce pracující s 128 bitovými bloky dat mají návratovou hodnotu typu `void` – žádnou hodnotu nevrací. U funkcí cfb8 a ofb8 se pracuje s bloky dat 8 bitů, proto jsou vstupní parametry a návratová hodnota typu `int`. V následujících odrážkách jsou popsány jednotlivé funkce pro šifrování.

1. **encryption_ecb** – funkce provádí šifrování AES v módu ECB.
2. **encryption_cbc** – funkce provádí šifrování AES v módu CBC. V proměnné `PreviousCipherText` je uložen inicializační vektor pro šifrování prvního bloku čistého textu. Po jeho zašifrování je do této proměnné uložen šifrovaný text, který se použije pro šifrování následujícího bloku čistého textu.
3. **encryption_ctr** – funkce provádí šifrování AES v módu CTR. Hodnota čítače je předávána funkci pomocí vstupního parametru (pomocí ukazatele `*pCounter`). Funkce neřeší inkrementaci čítače – inkrementaci čítače musí řešit volající (skript v prostředí Matlab).
4. **encryption_cfb** – funkce provádí šifrování AES v módu CFB. Tato funkce je schopná zpracovávat bloky o velikosti $s = \langle 1,128 \rangle$. Třetím vstupním parametrem je ukazatel `uint *pInputBlock`, pomocí kterého lze získat hodnotu nového vstupního bloku. Výstupní hodnota získaná pomocí `uint *pCipherText` obsahuje $s$ bitů šifrovaného textu a $128 - s$ bitů výstupního bloku. Volající (skript v Matlabu) řeší získání $s$ bitů šifrovaného textu z navrácených 128 bitů.

5. **encryption_ofb** – funkce provádí šifrování AES v módu OFB. Stejně jako `encryption_cfb` je tato funkce schopná zpracovávat bloky o velikosti $s = \langle 1{,}128 \rangle$. Třetím vstupním parametrem je ukazatel `uint *pInputBlock`, pomocí kterého lze získat hodnotu nového vstupního bloku. Výstupní hodnota získaná pomocí `uint *pCipherText` obsahuje $s$ bitů šifrovaného textu a $128 - s$ bitů výstupního bloku. Volající (skript v Matlabu) řeší získání $s$ bitů šifrovaného textu z navrácených 128 bitů.
6. **encryption_cfb8** – funkce provádí šifrování AES v módu CFB pro velikost zpracovávaného bloku $s = 8$. Tato funkce provádí posun šifrového textu a vstupního bloku pomocí jedné instrukce a návratová hodnota je vždy jeden byte. Volající tedy nemusí řešit získávání $s$ bitů z navrácené hodnoty, jako tomu bylo u `encryption_cfb`.
7. **encryption_cfb128** – funkce provádí šifrování AES v módu CFB pro velikost zpracovávaného bloku $s = 128$. V této funkci nejsou řešeny žádné posuny, protože při $s = 128$ je nový vstupní blok tvořen šifrovaným textem.
8. **encryption_ofb8** – funkce provádí šifrování AES v módu OFB pro velikost zpracovávaného bloku $s = 8$. Stejně jako u cfb8 volající nemusí řešit získávání $s$ bitů z navrácené hodnoty.
9. **encryption_ofb128** – funkce provádí šifrování AES v módu CFB pro velikost zpracovávaného bloku $s = 128$. V této funkci nejsou řešeny žádné posuny, protože při $s = 128$ je nový vstupní blok tvořen 128 bity výstupního bloku.

Funkce pro šifrování a dešifrování v módu CFB a OFB mají vstupní parametr ukazatel `uint *pInputBlock`. Tímto ukazatelem lze z funkce získat hodnotu nového vstupního bloku, který se vytváří pomocí 128 bitového posuvného registru. Hodnota nového vstupního bloku je ve funkci ukládána do paměti a použita pro šifrování následujícího bloku čistého textu Funkci se tato hodnota nepředává a slouží pouze k zobrazení posouvání vstupního bloku.

Funkce pro dešifrování fungují obdobně, jako funkce pro šifrování. Např. u funkcí `decryption_cfb` a `decryption_ofb` výstupní hodnota obsahuje $s$ bitů čistého textu a $128 - s$ bitů výstupního bloku. Tedy zde volající musí řešit získání $s$ bitů čistého textu z navrácených 128 bitů.

## 5.6 Ukázky výsledků

### 5.6.1 Ověření funkčnosti knihovny

Správná funkčnost funkcí knihovny byla ověřena vytvořením skriptů v prostředí Matlab. Ve skriptech jsou definovány vstupní testovací vektory podle publikací [5] a [8], viz příloha C. Návratové hodnoty všech funkcí odpovídají výstupním testovacím vektorům z publikací [5] a [8], čímž je ověřena správná funkčnost funkcí knihovny.

### 5.6.2 Ukázky výsledků některých módů

V této kapitole budou zobrazeny výstupy vektory skriptů volající jednotlivé funkce knihovny *aes.dll*. Výsledné hodnoty jsou zapisovány ve formě čtyř dvojslov v hexadecimální podobě, kde na prvním řádku je nejméně významné dvojslovo a levý byte je nejméně významný.

***ECB-AES128-192-256***

V Tab. 5.12 jsou výstupy ze skriptu *aes_ecb.m*. Skript využívá funkce:

- `key_size_setup` pro nastavení velikosti klíčů,
- `key_expansion` a `key_expansion_dec` pro odvození šifrovacích a dešifrovacích klíčů a
- `encryption_ecb` a `decryption_ecb` pro šifrování a dešifrování ve podle módu ECB.

Vstupními daty byly testovací vektory z publikace [5]. Výsledky uvedené v publikaci se shodují s výsledky dosaženými pomocí funkce `encryption_ecb` a `decryption_ecb`.

**Tab. 5.12: Hodnoty dosažené ve skriptu aes_ecb.m.**

| Velikost klíče [bity]: | 128 | 192 | 256 |
|---|---|---|---|
| Šifrovaný text | 69C4E0D8<br>6A7B0430<br>D8CDB780<br>70B4C55A | DDA97CA4<br>864CDFE0<br>6EAF70A0<br>EC0D7191 | 8EA2B7CA<br>516745BF<br>EAFC4990<br>4B496089 |
| Čistý text | 00112233<br>44556677<br>8899AABB<br>CCDDEEFF | 00112233<br>44556677<br>8899AABB<br>CCDDEEFF | 00112233<br>44556677<br>8899AABB<br>CCDDEEFF |

***CBC-AES128-192-256***

V tabulce 5.13 jsou výstupy ze skriptu *aes_cbc.m*. Skript využívá funkce:

- `key_size_setup` pro nastavení velikosti klíčů,
- `key_expansion` a `key_expansion_dec` pro odvození šifrovacích a dešifrovacích klíčů,
- `initvecotr_setup` pro nastavení inicializačního vektoru a
- `encryption_cbc` a `decryption_cbc` pro šifrování a dešifrování ve podle módu CBC.

Příloha D obsahuje část skriptu *aes_cbc.m*, která implementuje volání funkcí pro mód CBC a velikost klíče 128 bitů. Vstupními daty byly testovací vektory z publikace [8]. Výsledky uvedené v publikaci se shodují s výsledky dosaženými pomocí funkce `encryption_cbc` a `decryption_ecb`.

**Tab. 5.13: Hodnoty dosažené ve skriptu aes_cbc.m.**

| Velikost klíče [bity]: | | 128 | 192 | 256 |
|---|---|---|---|---|
| | Blok č. | | | |
| Šifrovaný text | 1 | 7649ABAC<br>8119B246<br>CEE98E9B<br>12E9197D | 4F021DB2<br>43BC633D<br>7178183A<br>9FA071E8 | F58C4C04<br>D6E5F1BA<br>779EABFB<br>5F7BFBD6 |
| | 2 | 5086CB9B<br>507219EE<br>95DB113A<br>917678B2 | B4D9ADA9<br>AD7DEDF4<br>E5E73876<br>3F69145A | 9CFC4E96<br>7EDB808D<br>679F777B<br>C6702C7D |
| | 3 | 73BED6B8<br>E3C1743B<br>7116E69E<br>22229516 | 571B2420<br>12FB7AE0<br>7FA9BAAC<br>3DF102E0 | 39F23369<br>A9D9BACF<br>A530E263<br>04231461 |
| | 4 | 3FF1CAA1<br>681FAC09<br>120ECA30<br>7586E1A7 | 08B0E279<br>88598881<br>D920A9E6<br>4F5615CD | B2EB05E2<br>C39BE9FC<br>DA6C1907<br>8C6A9D1B |

***CTR-AES128***

V tabulce 5.14 jsou výstupy ze skriptu *aes_ctr.m*. Příloha E obsahuje část skriptu *aes_ctr.m*, implementuje volání funkcí pro mód CTR a velikost klíče 128 bitů. Skript využívá funkce:

- `key_size_setup` pro nastavení velikosti klíčů,
- `key_expansion` pro odvození šifrovacích klíčů a
- `encryption_ctr` a `decryption_ctr` pro šifrování a dešifrování ve podle módu CTR.

V módu čítače se při šifrování i dešifrování používá algoritmus šifrování, proto není potřeba odvozovat dešifrovací klíče pomocí funkce `key_expansion_dec`. Vstupními daty byly testovací vektory z publikace [8]. Výsledky uvedené v publikaci se shodují s výsledky dosaženými pomocí funkce `encryption_ctr` a `decryption_ctr`. Na hodnotách čítače si lze všimnout, že se s každým následujícím blokem se přičítá 1 k nejvýznamnějšímu bitu.

**Tab. 5.14: Hodnoty dosažené ve skriptu aes_ctr.m pro velikost klíče 128 bitů.**

| Blok č. | Hodnota čítače | Šifrovaný text |
|---|---|---|
| 1 | F0F1F2F3<br>F4F5F6F7<br>F8F9FAFB<br>FCFDEFFF | 874D6191<br>B620E326<br>1BEF6864<br>990DB6CE |
| 2 | F0F1F2F3<br>F4F5F6F7<br>F8F9FAFB<br>FCFDFF00 | 9806F66B<br>7970FDFF<br>8617187B<br>B9FFFDFF |
| 3 | F0F1F2F3<br>F4F5F6F7<br>F8F9FAFB<br>FCFDFF01 | 5AE4DF3E<br>DBD5D35E<br>5B4F0902<br>0DB03EAB |
| 4 | F0F1F2F3<br>F4F5F6F7<br>F8F9FAFB<br>FCFDFF02 | 1E031DDA<br>2FBE03D1<br>792170A0<br>F3009CEE |

***CFB8-AES128***

V Tab. 5.15 jsou výstupy ze skriptu *cfb8_aes128.m*. Tento skript šifruje 18 testovacích vektorů (18 bytů čistého textu) pomocí funkcí `encryption_cfb` a `encryption_cfb8` pro velikost klíče 128 bitů. Při použití obou funkcí je dosaženo stejných vektorů šifrovaného textu. Tyto vektory jsou shodné s vektory uvedenými v publikaci [8].

**Tab. 5.15: Hodnoty dosažené ve skriptu cfb8_aes128.m pro velikost klíče 128 bitů.**

| Segment | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| Čistý text | 6b | c1 | be | e2 | 2e | 40 | 9f | 96 | e9 |
| | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| | 3d | 7e | 11 | 73 | 93 | 17 | 2a | ae | 2d |
| Segment | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Šifrovaný text | 3b | 79 | 42 | 4c | 9c | 0d | d4 | 36 | ba |
| | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| | ce | 9e | 0e | d4 | 58 | 6a | 4f | 32 | b9 |

Následující obrázek zachycuje inicializační vektory. Každý řádek představuje jedno dvojslovo (4 bytů). Začíná-li každý řádek (dvojslovo) nulou, v prostředí Matlab se tato nula nezobrazí – proto některé vektory o jeden znak méně. Na vektorech si lze všimnou posouvání vstupního bloku (na začátku je tvořen inicializačním vektorem) o byte k méně významným bitům, kde nejvíce významný byte je tvořen šifrovaným textem.

```
>> cfb8_aes128
i = 0            i = 4         i = 8         i = 12        i = 16
Init_Vector =    InputBlock =  InputBlock =  InputBlock =  InputBlock =

0010203          04050607      08090A0B      0C0D0E0F      3B79424C
4050607          08090A0B      0C0D0E0F      3B79424C      9C0DD436
8090A0B          0C0D0E0F      3B79424C      9C0DD436      BACE9E0E
C0D0E0F          3B79424C      9C0DD436      BACE9E0E      D4586A4F

i = 1            i = 5         i = 9         i = 13        i = 17
InputBlock =     InputBlock =  InputBlock =  InputBlock =  InputBlock =

1020304          05060708      090A0B0C      0D0E0F3B      79424C9C
5060708          090A0B0C      0D0E0F3B      79424C9C      0DD436BA
90A0B0C          0D0E0F3B      79424C9C      0DD436BA      CE9E0ED4
D0E0F3B          79424C9C      0DD436BA      CE9E0ED4      586A4F32

i = 2            i = 6         i = 10        i = 14        i = 18
InputBlock =     InputBlock =  InputBlock =  InputBlock =  InputBlock =

2030405          06070809      0A0B0C0D      0E0F3B79      424C9C0D
6070809          0A0B0C0D      0E0F3B79      424C9C0D      D436BACE
A0B0C0D          0E0F3B79      424C9C0D      D436BACE      9E0ED458
E0F3B79          424C9C0D      D436BACE      9E0ED458      6A4F32B9

i = 3            i = 7         i = 11        i = 15
InputBlock =     InputBlock =  InputBlock =  InputBlock =

3040506          0708090A      0B0C0D0E      0F3B7942
708090A          0B0C0D0E      0F3B7942      4C9C0DD4
B0C0D0E          0F3B7942      4C9C0DD4      36BACE9E
F3B7942          4C9C0DD4      36BACE9E      0ED4586A
```

**Obr. 5.1: Výstup skriptu cfb8_aes128.m – inicializační vektory.**

### *OFB128-AES128*

Obr. 5.2 zobrazuje výsledky po spuštění skriptu *aes_ofb.m* v prostředí Matlab. Skript využívá funkce:

- `key_size_setup` pro nastavení velikosti klíčů,
- `key_expansion` pro odvození šifrovacích klíčů,
- `initvecotr_setup` pro nastavení inicializačního vektoru,
- `s_bits_setup` pro nastavení velikosti zpracovávaného bloku dat a
- `encryption_ofb` a `decryption_ofb` pro šifrování a dešifrování ve podle módu OFB.

Hodnota nového vstupního bloku odpovídá hodnotě výstupního bloku.

```
>> aes_ofb

CipherText_128 =      Inputblock =

3B3FD92E              50FE67CC
B72DAD20              996D32B6
333449F8              DA0937E9
E83CFB4A              9BAFEC60
```

**Obr. 5.2: Výsledný šifrovaný text a nový vstupní blok.**

Tab. 5.16 obsahuje hodnoty vektorů z publikace [8] a slouží pro porovnání s výsledky skriptu, zobrazených na Obr. 5.2.

**Tab. 5.16: Testovací vektory z publikace [8] u šifrování módu OFB-AES128.**

| OFB-AES128.Encrypt | |
|---|---|
| Key | 2b7e151628aed2a6abf7158809cf4f3c |
| IV | 000102030405060708090a0b0c0d0e0f |
| Block #1 | |
| Input Block | 000102030405060708090a0b0c0d0e0f |
| Output Block | 50fe67cc996d32b6da0937e99bafec60 |
| Plaintext | 6bc1bee22e409f96e93d7e117393172a |
| Ciphertext | 3b3fd92eb72dad20333449f8e83cfb4a |
| Block #2 | |
| Input Block | 50fe67cc996d32b6da0937e99bafec60 |

### 5.6.3 Měření rychlosti

Pro měření rychlosti funkcí lze v prostředí Matlab využít funkce `tic` a `toc`. Funkce `tic` zapíná měření času a funkce `toc` měření času vypíná. Výsledkem je čas (v sekundách) od spuštění měření do jeho vypnutí. Těchto funkcí je využito pro porovnání rychlosti funkcí `encryption_cfb` ($s$ = 8) a `encryption_cfb8` při šifrování a dešifrování 18 bloků dat (18 bytů). Ve skriptu jsou pro každou funkci měřeny dva časy.

První čas (CFB_loop_enc a CFB8_loop_enc) měří dobu vykonání celé smyčky pro šifrování. Tato smyčka u funkce `encryption_cfb` obsahuje:

1. definování čistého textu,
2. vytvoření ukazatele na čistý text,
3. volání funkce `encryption_cfb`,
4. uložení výstupních 128 bitů dat,
5. získání $s$ bitů z výstupních 128 bitů dat a
6. uložení $s$ bitů do pole šifrovaných textů.

U funkce `encryption_cfb8` se ve smyčce provádí:

1. definování čistého textu,
2. zavolání funkce `encryption_cfb8` a
3. uložení návratové hodnoty do pole šifrovaných textů.

Druhý čas měří dobu volání a vykonání samotných funkcí `encryption_cfb` a `encryption_cfb8`.

V důsledku rozdílné implementace obou funkcí lze očekávat, že funkce `encryption_cfb` bude pomalejší než funkce `encryption_cfb8`. Vliv na rychlost bude mít i počet předávaných parametrů. U funkce `encryption_cfb` jsou předávány tři 32 bitové ukazatele (ukazatel na čistý text, šifrovaný text a nový inicializační vektor) a u funkce `encryption_cfb8` je předáváno jedno 32 bitové číslo (čistý text). Na dobu trvání šifrování a dešifrování bude mít vliv i rozdílnost právě popsaných smyček. Smyčka pro funkci `encryption_cfb` je výpočtově náročnější, než smyčka pro funkci `encryption_cfb8`.

Skript provádějící měření má název *cfb8_aes128_cmp.m*.

**Kód 5.23: Implementace měření času u ve skriptu cfb8_aes128_cmp.m.**

```
% Implementace pro funkce encryption_cfb
CFB_encryption = 0;
CFB_loop_enc = tic;
      % Zapnutí měření doby trvání smyčky.
for i = 1:j
      % j je počet bloků, j = 18
      % definování čistého textu
      % vytvoření ukazatele na čistý text
      CFB_tic = tic;
            % Zapnutí měření doby vykonání funkce
      calllib('aes', 'encryption_cfb', pPlainText, pCipherText, pVector);
      CFB_encryption = CFB_encryption + toc(CFB_tic);
            % Vypnutí měření doby vykonání funkce a přičtení této
            % hodnoty k celkové době vykonávání funkce
      % uložení výstupních 128 bitů dat
      % získání s bitů šifrovaného textu z výstupu
      % uložení šifrovaného textu do pole šifrovaných textů
end
CFB_loop_enc = toc(CFB_loop_enc);
      % Vypnutí měření doby trvání smyčky.

% Implementace pro funkci encryption_cfb8
CFB8_encryption = 0;
CFB8_loop_enc = tic;
      % Zapnutí měření doby trvání smyčky.
for i = 1:j
      Plaintext = Plaintext_Array(i);
      CFB8_tic = tic;
                  % Zapnutí měření doby vykonání funkce
```

```
        Ciphertext_array(i) = calllib('aes', 'encryption_cfb8', Plaintext);
        CFB8_encryption = CFB8_encryption + toc(CFB8_tic);
            % Vypnutí měření doby vykonání funkce a přičtení této
            % hodnoty k celkové době vykonávání funkce
end
CFB8_loop_enc = toc(CFB8_loop_enc);
    % Vypnutí měření doby trvání smyčky.
```

Implementace u dešifrování je stejná. Průměrné naměřené hodnoty jsou zobrazeny v následující tabulce:

**Tab. 5.17: Naměřené doby trvání pomocí skriptu cfb8_aes128.m.**

| | CFB_loop_enc [$.10^{-4}$ sec] | CFB_encryption [$.10^{-4}$ sec] | CFB8_loop_enc [$.10^{-4}$ sec] | CFB8_encryption [$.10^{-4}$ sec] |
|---|---|---|---|---|
| Šifrování | 64,00 | 3,64 | 8,00 | 2,64 |
| Dešifrování | 45,00 | 3,45 | 8,69 | 2,34 |

V Tab. 5.17 je patrné, že volání a vykonání funkce `encryption_cfb8` je rychlejší než u funkce `encryption_cfb`. Dále je patrné, že u obou funkcí má na dobu trvání největší vliv práce s daty v prostředí Matlab. U funkce `encryption_cfb` je tento vliv značný. V Tab. 5.18 je znázorněno, jaká část z celkové doby náleží vykonávání operací s daty v prostředí Matlab ($t_d$) a jaká části celkové doby náleží volání a vykonávání dané funkce ($t_f$).

**Tab. 5.18: Poměr časů $t_d$ a $t_f$ při šifrování a dešifrování 18 bytů.**

| Použitá funkce | $t_{celk}$ | | $t_d$ | | $t_f$ | |
|---|---|---|---|---|---|---|
| | [sec] | [%] | [sec] | [%] | [sec] | [%] |
| `encryption_cfb` | $64.10^{-4}$ | 100 | $60,36.10^{-4}$ | 94,3 | $3,64.10^{-4}$ | 5,7 |
| `decryption_cfb` | $45.10^{-4}$ | 100 | $41,55.10^{-4}$ | 92,3 | $3,45.10^{-4}$ | 7,6 |
| `encryption_cfb8` | $8,00.10^{-4}$ | 100 | $5,36.10^{-4}$ | 67,0 | $2,64.10^{-4}$ | 33,0 |
| `decryption_cfb8` | $8,69.10^{-4}$ | 100 | $6,35.10^{-4}$ | 73,1 | $2,34.10^{-4}$ | 26,9 |

### 5.6.4 Příklad praktického šifrování

Použitím funkcí `encryption_cfb` a `encryption_cfb8` byl zašifrován audio soubor .mp3 o velikosti 4,5MB (4 517 044 bytů). Při šifrování pomocí funkce `encryption_cfb` byla nastavena velikosti zpracovávaného bloku $s = 8$. Pro otevření souboru v prostředí Matlab slouží funkce `fopen`. Pomocí funkce `fread` získáme obsah otevřeného souboru, který je uložen do proměnné. Tato proměnná představuje pole čistého textu.

**Kód 5.24: Otevření a načtení souboru.**

```
% Otevření a načtení souboru
% Písnička zdarma ke stažení na:
http://freedownloads.last.fm/download/29775084/The%2BLaw.mp3
file_id = fopen ('Museum - The Law.mp3')
Plaintext_Array = fread(file_id);
    % Pole Plaintext_Array obsahuje 4 517 044 bytů.
```

Po otevření souboru a uložení hodnot do proměnné následuje šifrování. Smyčky šifrování (popsané v předešlé kapitole) se provedou 4 517 044-krát. Při šifrování byly měřeny časy stejným způsobem, jaký je popsán v předchozí kapitole. Pro měření byl vytvořen skript *cfb_file_enc.m*, který šifruje 4.5MB nejprve pomocí funkce `encryption_cfb` a poté pomocí funkce `encryption_cfb8` Tab. 5.19 obsahuje naměřené hodnoty.

**Tab. 5.19: Naměřené doby trvání šifrování souboru.**

| | CFB_loop_enc [sec] | CFB_encryption [sec] | CFB8_loop_enc [sec] | CFB8_encryption [sec] |
|---|---|---|---|---|
| Šifrování | 1085 | 75,19 | 50,25 | 25,67 |

Z Tab. 5.19 je patrné, že zašifrování 4.5MB souboru pomocí funkce `encryption_cfb` trvá přibližně 18 minut. Pomocí funkce encryption_cfb8 je 4.5MB soubor zašifrován za přibližně 50 sekund. Šifrování souboru pomocí funkce `encryption_cfb` je tedy nevhodné.

**Tab. 5.20: Poměr časů $t_d$ a $t_f$ při šifrování 4 517 044 bytů.**

| Použitá funkce | $t_{celk}$ | | $t_d$ | | $t_f$ | |
|---|---|---|---|---|---|---|
| | [sec] | [%] | [sec] | [%] | [sec] | [%] |
| `encryption_cfb` | 1085 | 100 | 1009,81 | 93,1 | 75,19 | 6,9 |
| `encryption_cfb8` | 50,25 | 100 | 24,58 | 48,9 | 25,67 | 51,1 |

Doba trvání dešifrování pomocí funkce `decryption_cfb` by byla přibližně stejně dlouhá. Proto pro dešifrování výsledného šifrovaného textu bylo zvoleno využití funkce `decryption_cfb8`.

Pro provedení zašifrování a dešifrování audio souboru byl vytvořen skript *cfb_file_encryption_decryption.m*. Skript používá funkce `encryption_cfb8` a `decryption_cfb8`. Ve skriptu jsou měřený doby trvání smyček pro šifrování a dešifrování.

**5.21: Doba trvání šifrování a dešifrování audio souboru.**

| | Doba trvání [s] |
|---|---|
| Šifrování | 22,12 |
| Dešifrování | 21,66 |

Skript provádí otevření a nahrání audio souboru, jeho zašifrování, dešifrování a uložení výsledku do souboru.

**Kód 5.25: Uložení dešifrovaného textu do souboru.**

```
% Otevření nového souboru.
new_file_id = fopen('out.mp3','w');
% Zápis hodnot PlainText_Array do souboru.
fwrite(new_file_id, PlainText_Array);
```

Dešifrovaný text byl zapsán do souboru out.mp3 a poté spuštěn v audio přehrávači. Audio bylo nepoškozeno a přehráno bez problému, čím že ověřena správná funkčnost funkcí `encryptiption_cfb8` a `decryption_cfb8`.

# 6 Závěr

V rámci prvního bodu zadání diplomové práce byly popsány symetrické blokové šifry a jejich operační módy (režimy). Bylo popsáno pět operačních módů: elektronická kódová kniha (ECB), řetězení šifrovaného textu (CBC), zpětná vazba ze šifrovaného textu (CFB), zpětná vazba z výstupu (OFB) a mód čítače (CTR). Tato část práce se zaměřuje především na popsání šifry Advanced Encryption Standard, zejména na její parametry, algoritmus šifrování a dešifrování, využívané transformace, expanzi klíče a ekvivalentní inverzní šifru.

Následně byly popsány instrukce z instrukční sady Intel® AES New Instruction (AES-NI). Jde o instrukce: AESENC a AESENCLAST pro šifrování, AESDEC a AESDECLAST pro dešifrování, AESKEYGENASSIST a AESIMC pro expanzi šifrovacího klíče a odvození dešifrovacích klíčů. Instrukce byly popsány pomocí pseudokódu a byly uvedeny komentované příklady jejich použití.

Pomocí instrukcí z instrukční sady AES-NI byly vytvořeny kódy v jazyce symbolických instrukcí, které implementovali šifru AES pro velikost klíče 128, 192 a 256 bitů. Následně byly implementovány zmíněné operační módy blokových šifer. Zdrojové kódy jsou opatřeny detailními komentáři. Algoritmus šifrování a dešifrování symetrické blokové šifry v módu ECB odpovídá algoritmu šifrování a dešifrování šifry AES. U módu CBC, CTR, CFB a OFB není řešena problematika generování inicializačních vektorů (hodnoty čítače u CTR). U módu CFB a OFB byl vyřešen 128 bitový registr a funkce implementující CFB a OFB lze použít pro zpracovávání bloku o velikosti $s \in \langle 1,128 \rangle$.

Zdrojové kódy byly využity k vytvoření dynamicky linkované knihovny aes.dll. Funkce této knihovny jsou volány skripty z prostředí Matlab, ve kterém jsou definovány vstupní parametry funkcí. Pomocí těchto skriptů jsou zobrazovány i výsledné hodnoty funkcí knihovny. Funkčnost knihovny byla ověřena pomocí testovacích vektorů uvedených v publikacích FIPS-197 a NIST800-38A. V těchto publikacích jsou zadány testovací vektory a k nim odpovídající výsledky. Pro mód CFB byla funkčnost ověřena praktickým zašifrováním a dešifrováním audio souboru (formát mp3) o velikosti 4,5MB. Audio soubor byl zašifrován a posléze dešifrován. Výstupní soubor je stejný jako vstupní a lze tedy i přehrát. Zároveň byly měřeny doby trvání šifrování a dešifrování pro jednotlivé módy. Z výsledných hodnot lze konstatovat, že největší vliv na dobu trvání má práce s daty v prostředí Matlab.

Vytvořením dynamické knihovny *aes.dll* byl splněn hlavní cíl této diplomové práce.

# 7 Seznam použité literatury

[1] STALLINGS, William. *Cryptography and Network Security Principles and Practices*. 4th Edition. Prentice Hall, 2005. ISBN 0131873164.

[2] BURDA, Karel. *Bezpečnost informačních systémů*. první vydání. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2013.

[3] LEŽÁK, Petr. *Bezpečnost informačních systémů - laboratorní cvičení*. Brno, 2013.

[4] GUERON, Shay. INTEL CORPORATION. *Intel ® Advanced Encryption Standard (AES) New Instructions Set* [online]. Revision 3.01. 2012 [cit. 2013-11-16]. Dostupné z: http://software.intel.com/sites/default/files/article/165683/aes-wp-2012-09-22-v01.pdf

[5] FEDERAL INFORMATION PROCESSING STANDARDS PUBLICATIONS. *Advanced Encryption Standard (AES)* [online]. 26.11. 2001 [cit. 2013-11-29]. Dostupné z: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[6] INTEL CORPORATION. *Intel® 64 and IA-32 Architectures Software Developer's Manual* [online]. Copyright © 1997-2013 [cit. 2013-12-04]. Dostupné z: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[7] Download the Intel AESNI Sample Library. INTEL CORPORATION. *Developer zone* [online]. 2011 [cit. 2014-04-27]. Dostupné z: https://software.intel.com/en-us/articles/download-the-intel-aesni-sample-library

[8] DWORKIN, Morris. NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *Recommendation for Block Cipher Modes of Operation: Methods and Techniques* [online]. 2001 [cit. 2014-05-08]. Dostupné z: http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

[9] ORSÁG, F. Pokročilé asemblery – studijní opora. Elektronické skriptum FIT VUT Brno 2006.

[10] RÁŠO, O. Programování funkcí v jazyce symbolických adres podle STDCALL konvence. Elektrorevue - Internetový časopis (http://www.elektrorevue.cz), 2009, roč. 2009, č. 8, s. 1-4. ISSN: 1213- 1539.

[11] Documentation Center. MATHWORKS. [online]. © 1994-2014 [cit. 2014-05-13]. Dostupné z: http://www.mathworks.com/help/

# 8 Seznam použitých zkratek

| | |
|---|---|
| AES | Advanced Encryption Standard |
| AES-NI | Advanced Encryption Standard New Instruction |
| API | Application Programming Interface |
| CBC | Cipher Block Chaining |
| CFB | Cipher Feedback |
| CTR | Counter |
| DES | Data Encryption Standard |
| ECB | Electronic Code Book |
| FIPS | Federal Information Processing Standards |
| IV | Initialization Vector |
| LSB | Least Significant Bit |
| NIST | National Institute of Standards and Technology |
| OFB | Output Feedback |
| RC4 | Rivest Cipher 4 |
| SIMD | Single Instruction, Multiple Data |
| SSE | Streaming SIMD Extensions |
| SSSE3 | Supplemental Streaming SIMD Extensions 3 |
| WPA2 | Wi-Fi Protected Access II |

# 9 Seznam příloh

A Příloha 1 – příklad expanze klíče ........ 70
B Příloha 2 – hlavičkový soubor aesdll.h ........ 76
C Příloha 3 – testovací vektory ........ 77
D Příloha 4 – skript implementující CBC-AES128 ........ 78
E Příloha 5 – skript implementující CTR-AES128 ........ 81
F Příloha 6 – skript implementující CFB8-AES128 ........ 84
G Příloha 7 – obsah přiloženého CD ........ 87

# A Příloha 1 – příklad expanze klíče

V této příloze bude z šifrovacího klíče o délce 128 bitů odvozen první rundovní klíč. Nejprve podle algoritmu expanze klíče popsaného v kapitole 3.2, poté podle algoritmu využívající instrukci AESKEYGENASSIST, který je popsán v kapitole 4.3 (Kód 4.3).

Mějme šifrovací klíč $K = 3C\ 4F\ CF\ 09\ 88\ 15\ F7\ AB\ A6\ D2\ AE\ 28\ 16\ 15\ 7E\ 2B$.

Řetězec upravíme dle Intel konvence, tedy dle Tab. 4.2 a Tab. 4.3. Matice pak bude vypadat takto:

**Tab. A.1: Šifrovací klíč.**

| 2B | 28 | AB | 09 |
|---|---|---|---|
| 7E | AE | F7 | CF |
| 15 | D2 | 15 | 4F |
| 16 | A6 | 88 | 3C |

***Odvození podle algoritmu AES***

Nejprve odvodíme první rundovní klíč podle algoritmu popsaném v 3.2. Bude postupováno dvojslovo po dvojslově (matice 4x1) zleva doprava.

První dvojslovo prvního rundovního klíče $W_i$ je na pozici 4, tato pozice je násobkem čísla 4, což znamená, že se pro daný sloupec bude aplikovat rovnici:

$$W[i] = W[i-4] \oplus SubWord\left(RotWord\left(W[i-1]\right)\right) \oplus RCon[j].^{4} \tag{A.1}$$

**Tab. A.2: Pole s klíči.**

| | $W_{i-4}$ | | | $W_{i-1}$ | $W_i$ | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 2B | 28 | AB | 09 | | | | | |
| | 7E | AE | F7 | CF | | | | | … |
| | 15 | D2 | 15 | 4F | | | | | |
| | 16 | A6 | 88 | 3C | | | | | |
| Pozice $i$: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |

$$W[4] = \begin{vmatrix} 2B \\ 7E \\ 15 \\ 16 \end{vmatrix} \oplus RotWord\left(SubWord\left(\begin{vmatrix} 09 \\ CF \\ 4D \\ 3C \end{vmatrix}\right)\right) \oplus \begin{vmatrix} 01 \\ 00 \\ 00 \\ 00 \end{vmatrix} = \begin{vmatrix} 2B \\ 7E \\ 15 \\ 16 \end{vmatrix} \oplus \begin{vmatrix} 8A \\ 84 \\ EB \\ 01 \end{vmatrix} \oplus \begin{vmatrix} 01 \\ 00 \\ 00 \\ 00 \end{vmatrix} = \underline{\underline{\begin{vmatrix} A0 \\ FA \\ FE \\ 17 \end{vmatrix}}} \tag{A.2}$$

[4] Pořadí transformací SubWord a RotWord lze zaměnit.

**Tab. A.3: Pole s klíči**

| | $W_{i-4}$ | | | $W_{i-1}$ | $W_i$ | | | |
|---|---|---|---|---|---|---|---|---|
| | 2B | 28 | AB | 09 | A0 | | | |
| | 7E | AE | F7 | CF | FA | | | … |
| | 15 | D2 | 15 | 4F | FE | | | |
| | 16 | A6 | 88 | 3C | 17 | | | |
| Pozice *i*: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Další dvojslova 1. rundovního klíče (dvojslova na pozici 5, 6, 7) se odvozují stejně podle rovnice:

$$W[i] = W[i-1] \oplus W[i-4] \tag{A.3}$$

$$W[5] = W[4] \oplus W[1] = \begin{vmatrix} A0 \\ FA \\ FE \\ 17 \end{vmatrix} \oplus \begin{vmatrix} 28 \\ AE \\ D2 \\ A6 \end{vmatrix} = \underline{\underline{\begin{vmatrix} 88 \\ 54 \\ 2C \\ B1 \end{vmatrix}}} \tag{A.4}$$

$$W[6] = W[5] \oplus W[2] = \begin{vmatrix} 88 \\ 54 \\ 2C \\ B1 \end{vmatrix} \oplus \begin{vmatrix} AB \\ F7 \\ 15 \\ 88 \end{vmatrix} = \underline{\underline{\begin{vmatrix} 23 \\ A3 \\ 39 \\ 39 \end{vmatrix}}} \tag{A.5}$$

$$W[7] = W[6] \oplus W[3] = \begin{vmatrix} 23 \\ A3 \\ 39 \\ 39 \end{vmatrix} \oplus \begin{vmatrix} 09 \\ CF \\ 4F \\ 3C \end{vmatrix} = \underline{\underline{\begin{vmatrix} 2A \\ 6C \\ 76 \\ 05 \end{vmatrix}}} \tag{A.6}$$

**Tab. A.4: Pole klíčů.**

| 2B | 28 | AB | 09 | A0 | 88 | 23 | 2A | |
|---|---|---|---|---|---|---|---|---|
| 7E | AE | F7 | CF | FA | 54 | A3 | 6C | … |
| 15 | D2 | 15 | 4F | FE | 2C | 39 | 76 | |
| 16 | A6 | 88 | 3C | 17 | B1 | 39 | 05 | |
| Šifrovací klíč | | | | 1. rundovní klíč | | | | |

První rundovní klíč je $RK_1 = 05\ 76\ 6C\ 2A\ 39\ 39\ A3\ 23\ B1\ 2C\ 54\ 88\ 17\ FE\ FA\ A0$.

### *Odvození pomocí instrukce AESKEYGENASSIST*

Nejprve si připomeňme formát instrukce. Instrukce má tři argumenty: xmm1, xmm2/m128 a imm8. První argument je výstup instrukce. Druhý argument je vstup – expandovaný klíč. Třetí argument je 8 bitový řetězec, pomocí kterého se instrukci předává rundovní konstanta, která je jiná pro každý rundovní klíč, viz Tab. 3.3.

```
AESKEYGENASSIST xmm1, xmm2/m128, imm8
```

Pro expanzi prvního rundovního klíče mějme kód:

```
AESKEYGENASSIST xmm2, xmm1, 0x1
```

V xmm1 je uložen šifrovací klíč. Na pseudokódu Pseudokód 4.3 je vidno, že se šifrovací klíč uloží do 128 bitové proměnné Tmp. Také se vytvoří dvojslovo obsahující rundovní konstantu RCon, viz tabulka A.5.

**Tab. A.5: Obsah Tmp a RCon.**

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| 2B | 28 | AB | 09 |
| 7E | AE | F7 | CF |
| 15 | D2 | 15 | 4F |
| 16 | A6 | 88 | 3C |

| RCon |
|---|
| 01 |
| 00 |
| 00 |
| 00 |

Výstup instrukce je dán touto rovnicí:

$$xmm1 = [X3, X2, X1, X0] = \begin{array}{l} [\,RotWord(SubWord(X3)) \oplus RCon, \\ SubWord(X3), \\ RotWord(SubWord(X1)) \oplus RCon, \\ SubWord(X1)] \end{array} \tag{A.7}$$

Výsledek je potom:

**Tab. A.6: Výstup po instrukci AESKEYGENASSIST.**

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| AE | E5 | CF | 8B |
| D2 | B5 | 4F | 84 |
| AB | 62 | 3C | EF |
| 28 | 34 | 09 | 01 |

Instrukce AESKEYGENNASSIST sama o sobě expanzi neprovádí. Je nutné expanzi dokončit. V kapitole 4.3 je expanze dokončena pomocí sekvence instrukcí, viz Kód 4.3. V kódu si lze všimnout, že se po provedení instrukce AESKEYGENASSIST zavolá podprogram key_expansion_128, který dokončuje expanzi klíče. Kód (bez komentářů) je zde uveden:

**Kód A.1: Podprogram pro dokončení expanze klíče**

```
key_expansion_128:
        pshufd xmm2, xmm2, 0xff
        vpslldq xmm3, xmm1, 0x4
        pxor xmm1, xmm3
        vpslldq xmm3, xmm1, 0x4
        pxor xmm1, xmm3
        vpslldq xmm3, xmm1, 0x4
        pxor xmm1, xmm3
        pxor xmm1, xmm2
```

```
        movdqu XMMWORD PTR [rcx], xmm1
add rcx, 0x10
ret
```

Nyní bude zobrazen obsah důležitých xmm registrů po instrukcích, které dokončují expanzi klíče. V registru xmm1 je uložen šifrovací klíče a v registru xmm2 je výstup instrukce AESKEYGENASSIST. Obsah registru xmm1 je tedy v tabulce A.1 a registru xmm2 je v tabulce A.6.

1. **PSHUFD XMM2, XMM2, 0xFF** - v této instrukci (Packed Shuffle Doublewords) se třetím argumentem řídí, které dvojslova zdrojového registru budou uložena do dvojslov cílového registru. 0xFF lze rozdělit na čtyři dvojice bitů (dvojice s hodnotami 0-3). Každá dvojice odpovídá jednomu dvojslovu cílového registru (X0-X3). Hodnota dvojice bitů vyjadřuje číslo dvojslovu zdrojového registru (X0-X3), který se bude kopírovat do cílového na odpovídající pozic. 0xFF znamená, že dvojslovo X3 zdrojového registru bude zkopírováno do všech dvojslov cílového registru. Výstupem této instrukce tedy bude registr xmm2, který bude obsahovat hodnoty podle následující tabulky:

**Tab. A.7: Výstup po instrukci PSHUFD.**

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| 8B | 8B | 8B | 8B |
| 84 | 84 | 84 | 84 |
| EF | EF | EF | EF |
| 01 | 01 | 01 | 01 |

2. **VPSLLDQ XMM3, XMM1, 0x4** - instrukce VPSSLDQ (Packed Shift Left Logical Double Quadword) posune obsah zdrojového registru (xmm1) o počet bytů daným třetím argumentem. Spodní bity vynuluje a výsledek uloží do cílového registru (xmm3). Výsledek je pak:

**Tab. A.8: Výstup instrukce VPSLLDQ - poprvé.**

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| 00 | 2B | 28 | AB |
| 00 | 7E | AE | F7 |
| 00 | 15 | D2 | 15 |
| 00 | 16 | A6 | 88 |

3. **PXOR XMM1, XMM3** - instrukce (Logical Exclusive OR) provádí exkluzivní disjunkci xmm1 XOR xmm2. Výsledek je uložen v registru xmm1.

Tab. A.9: Výstup instrukce PXOR - poprvé.

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| 2B | 03 | 83 | A2 |
| 7E | D0 | 59 | 38 |
| 15 | C7 | C7 | 5A |
| 16 | B0 | 2E | B4 |

4. **VPSLLDQ XMM3, XMM1, 0x4** - stejné jako v bodě 2.

Tab. A.10: Výstup po operaci VPSLLDQ - podruhé.

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| 00 | 2B | 03 | 83 |
| 00 | 7E | D0 | 59 |
| 00 | 15 | C7 | C7 |
| 00 | 16 | B0 | 2E |

5. **PXOR XMM1, XMM3** - stejné jako v bodě 3.

Tab. A.11: Výstup instrukce PXOR - podruhé.

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| 2B | 28 | 80 | 21 |
| 7E | AE | 89 | 61 |
| 15 | D2 | 00 | 9D |
| 16 | A6 | 9E | 9A |

6. **VPSLLDQ XMM3, XMM1, 0x4** - stejné jako v bodě 2 a 4.

Tab. A.12: Výstup po operaci VPSLLDQ - potřetí.

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| 00 | 2B | 28 | 80 |
| 00 | 7E | AE | 89 |
| 00 | 15 | D2 | 00 |
| 00 | 16 | A6 | 9E |

7. **PXOR XMM1, XMM3** - stejné jako v bodě 3 a 5.

**Tab. A.13: Výstup instrukce PXOR - potřetí.**

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| 2B | 28 | 80 | 21 |
| 7E | AE | 89 | 61 |
| 15 | D2 | 00 | 9D |
| 16 | A6 | 9E | 9A |

8. **PXOR XMM1, XMM2** - nyní se provede exkluzivní disjunkce nad xmm2 registrem, jehož obsah je v tabulce A.7 a registrem xmm1, jehož obsah je v A.13. Výsledek je uložen v registru xmm1 a představuje 1. rundovní klíč.

**Tab. A.14: Výsledný rundovní klíč.**

| X0 | X1 | X2 | X3 |
|---|---|---|---|
| A0 | 88 | 23 | 2A |
| FA | 54 | A3 | 6C |
| FE | 2C | 39 | 76 |
| 17 | B1 | 39 | 05 |

Tímto vytvoření prvního rundovního klíče hotové, následující instrukce provádějí uložení jeho uložení a posunutí adresy o 0x10 (odpovídá 128 bitům) pro uložení dalšího klíče.

9. **MOVDQU XMMWORD PTR [RCX], XMM1** - instrukce (Move Unaligned Double Quadword) uloží obsah registru xmm1 na místo v paměti s adresou v RCX.
10. **ADD RCX, 0x10** - k hodnotě v RCX přičte hexadecimální hodnotu 0x10.

V kódu následuje návrat z podprogramu, provedení instrukce AESKEYGENASSIST se vstupem v xmm1 registru (nyní první rundovní klíč) a rundovní konstantou pro druhý rundovní klíč. Následuje zavolání podprogramu pro dokončení expanze. Tento postup se aplikuje pro odvození všech rundovních klíčů.

# B Příloha 2 – hlavičkový soubor aesdll.h

V následujícím kódu je obsah celého hlavičkového souboru *aesdll.h*, který obsahuje funkční prototypy funkcí knihovny *aes.dll*. Jak je popsáno v kapitole 5.5.3, funkční prototypy jsou zapisovány ve formě:

```
návratová_hodnota konvence_volání název_funkce(vstupní parametry);
```

**Kód B.1: Obsah hlavičkového souboru aesdll.h**

```
void _stdcall key_size_setup(int keySize);
void _stdcall initvector_setup(uint *pInitVector);
void _stdcall s_bits_setup(int s);

void _stdcall key_expansion(uint *pKey);
void _stdcall key_expansion_dec();

void _stdcall encryption_ecb(uint *pPlainText, uint *pCipherText);
void _stdcall decryption_ecb(uint *pCipherText, uint *pPlainText);

void _stdcall encryption_ctr(uint *pCounter, uint *pPlainText, uint *pCipherText);
void _stdcall decryption_ctr(uint *pCounter, uint *pCipherText, uint *pPlainText);

void _stdcall encryption_cbc(uint *pPlainText, uint *pCipherText);
void _stdcall decryption_cbc(uint *pCipherText, uint *pPlainText);

int _stdcall encryption_cfb8(int plainText);
int _stdcall decryption_cfb8(int CipherText);

void _stdcall encryption_cfb128(uint *pPlainText, uint *pCipherText);
void _stdcall decryption_cfb128(uint *pCipherText, uint *pPlainText);

int _stdcall encryption_ofb8(int plainText);
int _stdcall decryption_ofb8(int CipherText);

void _stdcall encryption_ofb128(uint *pPlainText, uint *pCipherText);
void _stdcall decryption_ofb128(uint *pCipherText, uint *pPlainText);

void _stdcall encryption_cfb(uint *pPlainText, uint *pCipherText, uint *pInputBlock);
void _stdcall decryption_cfb(uint *pCipherText, uint *pPlainText, uint *pInputBlock);

void _stdcall encryption_ofb(uint *pPlainText, uint *pCipherText, uint *pInputBlock);
void _stdcall decryption_ofb(uint *pCipherText, uint *pPlainText, uint *pInputBlock);
```

# C Příloha 3 – testovací vektory

V následujících tabulkách jsou definovány vstupní testovací vektory, které byly použity pro ověření funkcí knihovny. V publikaci FIPS-197 ([5]) jsou definovány testovací vektory, které jsou použity v módu ECB. V publikaci NIST 800-38A ([8]) jsou definovány testovací vektory, které jsou použity v módech CBC, CTR, CFB a OFB.

**Tab C.1: Testovací vektory podle FIPS-197. [5]**

| Vektor | Hodnota |
| --- | --- |
| Čistý text | 00112233 44556677 8899aabb ccddeeff |
| 128 bitový klíč | 00010203 04050607 08090a0b 0c0d0e0f |
| 192 bitový klíč | 00010203 04050607 08090a0b 0c0d0e0f<br>10111213 14151617 |
| 256 bitový klíč | 00010203 04050607 08090a0b 0c0d0e0f<br>10111213 14151617 18191a1b 1c1d1e1f |

**Tab C.2: Testovací vektory podle NIST 800-38A. [8]**

| Vektor | Hodnota |
| --- | --- |
| Čistý text #1 | 6bc1bee2 2e409f96 e93d7e11 7393172a |
| Čistý text #2 | ae2d8a57 1e03ac9c 9eb76fac 45af8e51 |
| Čistý text #3 | 30c81c46 a35ce411 e5fbc119 1a0a52ef |
| Čistý text #4 | f69f2445 df4f9b17 ad2b417b e66c3710 |
| 128 bitový klíč | 2b7e1516 28aed2a6 abf71588 09cf4f3c |
| 192 bitový klíč | 8e73b0f7 da0e6452 c810f32b 809079e5<br>62f8ead2 522c6b7b |
| 256 bitový klíč | 603deb10 15ca71be 2b73aef0 857d7781<br>1f352c07 3b6108d7 2d9810a3 0914dff4 |
| Inicializační vektor | 00010203 04050607 08090a0b 0c0d0e0f |
| Inicializační čítač | f0f1f2f3 f4f5f6f7 f8f9fafb fcfdfeff |

# D Příloha 4 – skript implementující CBC-AES128

Skript *aes_cbc.m* šifruje a dešifruje čtyři 128 bitové bloky čistého textu definovaných podle NIST 800-38A ([8]). Následující kód popisuje implementaci pro velikost klíče 128 bitů.

**Kód D.1: Skript pro použití funkcí k šifrování CBC-AES128.**

```
%% CBC-AES128
% Tento kód provádí šifrování a dešifrování testovacích vektorů
% z publikace NIST 800-38A pro mód řetězení šifrovanéhé textu
% a velikost klíče 128.
close all, clear all

%% Definice klíčů, inicializačního vektoru a čistého textu podle
% dle NIST SP800-38A
Key128 = [swapbytes(uint32(hex2dec('2b7e1516')))
        swapbytes(uint32(hex2dec('28aed2a6')))
        swapbytes(uint32(hex2dec('abf71588')))
        swapbytes(uint32(hex2dec('09cf4f3c')))];
% Key128 = 2b7e1516 28aed2a6 abf71588 09cf4f3c
% Stejným způsobem jsou definovýny následující vektory.
Vector = […];
% Vector = 00010203 04050607 08090a0b 0c0d0e0f
Plaintext1 = […];
% Plaintext1 = 6bc1bee2 2e409f96 e93d7e11 7393172a
Plaintext2 = […];
% Plaintext2 = ae2d8a57 1e03ac9c 9eb76fac 45af8e51
Plaintext3 = […];
% Plaintext3 = 30c81c46 a35ce411 e5fbc119 1a0a52ef
Plaintext4 = […];
% Plaintext4 = f69f2445 df4f9b17 ad2b417b e66c3710
Plaintext = [Plaintext1; Plaintext2; Plaintext3; Plaintext4];
        % Proměnná Plaintext obsahuje všechny bloky čistých textů.

%% Nahrání knihovny
hfile=['aesdll.h'];
[notfound,warnings]=loadlibrary('aes.dll', hfile, 'mfilename', 'aes_cbc_m');

%% Vytvoření ukazatelů
pKey = libpointer('uint32Ptr', Key128);
pVector = libpointer('uint32Ptr', Vector);
        % Vytoření ukazatele na šifrovací klíč a inicializační vektor.
CipherArray=uint32([0 0 0 0]);
pCipherArray = libpointer('uint32Ptr', CipherArray);
        % Vytvoření ukazatele na pole šifrovaného textu, do kterého
        % bude ukládán výstup šifrovací funkce.
%% Expanze klíčů pro CBC-AES128
M = 4;
```

```
      % Udává velikost klíče v doublewordech.
expansion = tic;
calllib('aes', 'key_size_setup', M);
      % Nastavení velikosti klíče.
calllib('aes', 'key_expansion', pKey);
      % Expanze šifrovacího klíče.
calllib('aes', 'key_expansion_dec');
      % Odvození dešifrovacích klíčů.
Key128_expansion_time = toc(expansion);
      % Proměnná obsahuje dobu trvání expanze šifrovacího klíče
      % a odvození dešifrovacích klíčů.
```

```
%% Šifrování a dešifrování CBC-AES128
% Šifrování CBC-AES128.
calllib('aes', 'initvector_setup', pVector);
      % Nahrání inicializačnho vektoru.
CBC_AES128_enc = 0;
      % Inicializace proměnné, která bude použitá k měření doby
      % vykonání šifrovací funkce.
for i = 1:4
   fprintf('i = %d', i)
   pPlainText = libpointer('uint32Ptr', Plaintext(i*4-3:i*4));
      % Ukazatel postupně ukazuje i-tý blok čistého textu.
   f_enc = tic;
      % Zapnutí měření času.
   calllib('aes', 'encryption_cbc', pPlainText, pCipherArray);
      % Volání funkce pro šifrování CBC.
   CBC_AES128_enc = CBC_AES128_enc + toc(f_enc);
      % Přičtení doby vykonání funkce.
   Ciphertext_128=get(pCipherArray, 'Value');
   Ciphertext_128=dec2hex(swapbytes(Ciphertext_128()))
      % Zobrazení šifrovaného textu.
   eval(sprintf('CA_128_%d = Ciphertext_128;',i))
      % Vytvoření čtyr proměnných pro každý šifrovaný text.
end
Ciphertext_128=[CA_128_1; CA_128_2; CA_128_3; CA_128_4];
      % Proměnná Ciphertext_128 obsahuje všechny bloky šifrovaných
      % textů

% Dešifrování CBC-AES128.
calllib('aes', 'initvector_setup',pVector);
      % nahrání inicializačního vekotru.
% Zformatování výstupu šifrování.
Ciphertext=swapbytes(uint32(hex2dec(cellstr(Ciphertext_128))));
CBC_AES128_dec = 0;
      % Inicializace proměnné, která bude použitá k měření doby
      % vykonání dešifrovací funkce.
for i = 1:4
```

```
    fprintf('i = %d', i)
    pCipherArray = libpointer('uint32Ptr', Ciphertext(i*4-3:i*4));
        % Ukazatel postupně ukazuje i-tý blok šifrovaného textu.
    f_dec = tic;
        % Zapnutí měření času.
    calllib('aes','decryption_cbc', pCipherArray, pPlainText);
        % Volání funkce pro dešifrování cbc.
    CBC_AES128_dec = CBC_AES128_dec + toc(f_dec);
        % Přičtení doby vykonání funkce.
    Plaintext_128=get(pPlainText, 'Value');
    Plaintext_128=dec2hex(swapbytes(Plaintext_128()))
        % Zobrazení čistého textu.
end

%% Zobrazení dob trvání
Key128_expansion_time
CBC_AES128_enc
CBC_AES128_dec
```

# E Příloha 5 – skript implementující CTR-AES128

Skript *aes_ctr.m* šifruje a dešifruje čtyři 128 bitové bloky čistého textu definovaných podle NIST 800-38A ([8]). Následující kód popisuje implementaci pro velikost klíče 128 bitů. Generování inicializačního čítače není ve skriptu řešeno – hodnota je převzatá z NIST 800-38A. Ve skriptu se řeší inkrementace čítače. Čítač se inkrementuje přičítáním jedničky k nejvíce významným bitům. Ve skriptu se hodnota inkerementuje tak, že se nejprve obrátí pořadí bytů pomocí `swapbytes`, přičte se jednička a poté se opět obrátí pořadí bytů pomocí `swapbytes`.

**Kód E.1: Skript pro použití funkcí k šifrování CTR-AES128.**

```
%% CTR-AES128
% Tento kód provádí šifrování a dešifrování testovacích vektorů z
% publikace NIST 800-38A pro mód čítače a velikosti klíče 128.
close all, clear all

%% Definice klíčů, inicializačního vektoru a čistého textu podle
% dle NIST SP800-38A
Key128 = [swapbytes(uint32(hex2dec('2b7e1516')))
        swapbytes(uint32(hex2dec('28aed2a6')))
        swapbytes(uint32(hex2dec('abf71588')))
        swapbytes(uint32(hex2dec('09cf4f3c')))];
% Key128 = 2b7e1516 28aed2a6 abf71588 09cf4f3c
% Stejným způsobem jsou definovýny následující vektory.
Init_Counter = […];
% Init_Counter = f0f1f2f3 f4f5f6f7 f8f9fafb fcfdfeff
Plaintext1 = […];
% Plaintext1 = 6bc1bee2 2e409f96 e93d7e11 7393172a
Plaintext2 = […];
% Plaintext2 = ae2d8a57 1e03ac9c 9eb76fac 45af8e51
Plaintext3 = […];
% Plaintext3 = 30c81c46 a35ce411 e5fbc119 1a0a52ef
Plaintext4 = […];
% Plaintext4 = f69f2445 df4f9b17 ad2b417b e66c3710
Plaintext = [Plaintext1;Plaintext2;Plaintext3;Plaintext4];
      % Proměnná Plaintext obsahuje všechny bloky čistých textů.

%% Nahrání knihovny
hfile=['aesdll.h'];
[notfound,warnings]=loadlibrary('aes.dll', hfile, 'mfilename', 'aes_ctr_m');

%% Vytvoření ukazatelů
CipherArray=uint32([0 0 0 0]);
pCipherArray = libpointer('uint32Ptr', CipherArray);
      % Ukazatel na pole šifrovaného textu.
pKey = libpointer('uint32Ptr', Key128);
      % Ukazatel na šifrovací klíč.
```

```
%% Expanze klíčů pro CTR-AES128
M = 4;
calllib('aes', 'key_size_setup', M);
      % Nastavení velikosti klíče.
calllib('aes', 'key_expansion', pKey);
      % Expanze šifrovacího klíče
```

```
%% Šifrování a dešifrování CTR-AES128
% Šifrování CTR-AES128.
Counter = Init_Counter;
      % Do proměnné Counter nahrajeme inicializační čítač
CTR_AES128_enc = 0;
      % Inicializace proměnné, která bude použitá k měření doby
      % vykonání šifrovací funkce.
for i = 1:4
   fprintf('i = %d', i)
   pPlainText = libpointer('uint32Ptr', Plaintext(i*4-3:i*4));
      % Ukazatel postupně ukazuje i-tý blok čistého textu.
   pCounter = libpointer('uint32Ptr',  Counter);
      % Ukazatel na hodnotu čítače.
   f_enc = tic;
      % Zapnutí měření času.
   calllib('aes', 'encryption_ctr', pCounter, pPlainText, pCipherArray);
      % Volání funkce pro šifrování CTR.
   CTR_AES128_enc = CTR_AES128_enc + toc(f_enc);
      % Přičtení doby vykonání funkce.
   Ciphertext_128=get(pCipherArray, 'Value');
   Ciphertext_128=dec2hex(swapbytes(Ciphertext_128()))
      % Získání hodnoty šifrovaného textu a její zobrazení.
   eval(sprintf('CA_128_%d = Ciphertext_128;',i))
      % Vytvoření čtyr proměnných pro každý šifrovaný text.
   Counter(4) = swapbytes(Counter(4))+1;
   Counter(4) = swapbytes(Counter(4));
      % Inkrementace čítače.
   New_Counter = dec2hex(swapbytes(Counter))
      % Zobrazení hodnoty nového čítače.
end
Ciphertext_128=[CA_128_1; CA_128_2; CA_128_3; CA_128_4];
      % Proměnná Ciphertext_128 obsahuje všechny bloky šifrovaných
      % textů

% Dešifrování CTR-AES128.
% Přeformátování výstupu šifrování.
Ciphertext=swapbytes(uint32(hex2dec(cellstr(Ciphertext_128))));
Counter = Init_Counter;
CTR_AES128_dec = 0;
      % Inicializace proměnné, která bude použitá k měření doby
      % vykonání dešifrovací funkce.
```

```
for i = 1:4
   fprintf('i = %d', i)
   pCipherArray = libpointer('uint32Ptr', Ciphertext(i*4-3:i*4));
       % Ukazatel postupně ukazuje i-tý blok šifrovaného textu.
   pCounter = libpointer('uint32Ptr', Counter);
       % Ukazatel na hodnotu čítače.
   f_dec = tic;
       % Zapnutí měření času.
   calllib('aes','decryption_ctr', pCounter, pCipherArray, pPlainText);
       % Volání funkce pro dešifrování CTR.
   CTR_AES128_dec = CTR_AES128_dec + toc(f_dec);
       % Přičtení doby vykonání funkce.
   Plaintext_128=get(pPlainText, 'Value');
   Plaintext_128=dec2hex(swapbytes(Plaintext_128()))
       % Získání hodnoty čistého textu a její zobrazení.
   Counter(4) = swapbytes(Counter(4))+1;
   Counter(4) = swapbytes(Counter(4));
       % Inkrementace čítače.
   New_Counter = dec2hex(swapbytes(Counter))
       % Zobrazení hodnoty nového čítače.
end
```

```
%% Zobrazení dob trvání
CTR_AES128_enc
CTR_AES128_dec
```

# F Příloha 6 – skript implementující CFB8-AES128

Následující skript šifruje testovací vektory čistého textu definovaného v poli Plaintext_Array podle režimu CFB. Všechny definované hodnoty jsou převzaté z publikace NIST 800-38A ([8]). Skript je uložen na CD pod názvem *cfb8_aes128.m*. Ve skriptu je provedeno šifrování nejprve pomocí funkce encryption_cfb a poté pomocí funkce encryption_cfb8.

**Kód C.1: Skript pro použití funkcí k šifrování CFB8-AES128.**

```
%% CFB8-AES128
% Tento kód provádí šifrování a dešifrování testovacích vektorů z
% publikace NIST 800-38A pro mód zpětné vazby ze šifrovaného textu
% a velikost s = 8.
close all, clear all

%% Definování klíče, čistého textu a inicializačního vektoru podle
% NIST 800-38A
Plaintext_Array=hex2dec(['6b'; 'c1'; 'be'; 'e2'; '2e'; '40'; '9f'; '96'; 'e9'; '3d'; '7e'; '11'; '73';
'93'; '17'; '2a'; 'ae'; '2d']);
Key128 = [
   swapbytes(uint32(hex2dec('2b7e1516')))
   swapbytes(uint32(hex2dec('28aed2a6')))
   swapbytes(uint32(hex2dec('abf71588')))
   swapbytes(uint32(hex2dec('09cf4f3c')))];
Vector = [
   swapbytes(uint32(hex2dec('00010203')))
   swapbytes(uint32(hex2dec('04050607')))
   swapbytes(uint32(hex2dec('08090A0B')))
   swapbytes(uint32(hex2dec('0C0D0E0F')))];

%% Načtení knihovny
hfile=['aesdll.h'];
[notfound,warnings]=loadlibrary('aes.dll',hfile, 'mfilename', 'cfb8_aes128_m');

%% Expanze klíče
pKey = libpointer('uint32Ptr', Key128);
      % Vytvoření ukazatele na Key128.
M = 4;
      % Počet dvojslov klíče.
calllib('aes', 'key_size_setup', M);
      % Nastavení velikosti klíče.
calllib('aes','key_expansion', pKey);
      % Provedení expanze 128 bitového šifrovacího klíče.
      % Je volána funkce key_expansion, která přebírá ukazatel na
      % šifrovací klíč a expanduje jej expandovaný klíč je uložen v
      % paměti, funkce žádnou hodnotu nevrací.
```

```
%% Nastavení inicializačního vektoru
pVector = libpointer('uint32Ptr', Vector);
      % Vytvoření ukazatele na inicializační vektor.
calllib('aes','initvector_setup', pVector);
      % Zavolání funkce nastavující inicializační vektor, vstupní
      % parametr je ukazatel na inicializační vektor
      % Inicializační vektor je uložen do paměti, funkce nevrací
      % žádnou hodnotu.
```

```
%% Nastavení velikosti s
s = 8;
      % s = 8 bitů
calllib('aes', 's_bits_setup', s);
      % Volání funkce s_bits_setup, která uloží hodnotu s do paměti.
      % Funkce nevrací žádnou hodnotu.
```

```
%% Vytvoření ukazatele na pole šifrovaného textu
Ciphertext_out = uint32([0 0 0 0]);
pCipherText = libpointer('uint32Ptr', Ciphertext_out);
```

```
%% Šifrování CFB8-AES128 využitím funkce encryption_cfb
Size = size(Plaintext_Array);
j = Size(1);
      % Zjištění počtu bloků čistého textu.
Ciphertext_array = zeros(j,1);
      % Vytvoření pole pro ukládání šifrovaného textu.
i = 0;
fprintf('i = %d', i)
Init_Vector = dec2hex(swapbytes(Vector))
      % Zobrazení inicializačního vektoru.
% Smyčka se provede i-krát = počet bloků čistého textu.
for i = 1:j
   fprintf('i = %d', i)
   Plaintext = Plaintext_Array(i);
           % Uložení i-tého čistého textu do proměnné.
   pPlaintext = libpointer('uint32Ptr', Plaintext);
           % Vytvoření ukazatele na čistý text.
   Plaintext = dec2hex(Plaintext);
           % Zobrazí v command okně hexadecimální hodnotu čistého
           % textu.
   calllib('aes', 'encryption_cfb', pPlaintext, pCipherText, pVector);
           % Zavolání knihovny, která zašifruje čistý text, vrací
           % hodnotu šifrovaného textu a nového inicializačního
           % vektoru, který je použit pro šifrování následujícího
           % čistého textu.
   Output = get(pCipherText, 'Value');
```

```
            % Pomocí get získáme návratovou hodnotu funkce. V Output
            % je nyní uloženo 128 bitů, kde s = 8 bitů tvoří
            % šifrovaný text a 128 - s = 120 bitů tvoří nový vstupní
            % blok.
  % Získání s bitů šifrovaného textu z vrácených 128 bitů.
  Output = dec2hex(Output(1));
      % Output = 1 dvojslovo výstupu (prvních 32 bitů) - 8
      % hexadecimálních znaků.
  CipherText = hex2dec(Output(1,7:8));
      % CipherText = 2 nejvíce pravé hexadecimální znaky výstupu.
  Ciphertext_array(i) = CipherText;
      % Uložení 8 bitů šifrovaného textu do pole šifrovaných textů.
  InputBlock = get(pVector, 'Value');
  InputBlock = dec2hex(swapbytes(InputBlock ()))
      % Získání hodnoty vstupního bloku a zobrazení v hexadecimální
      % podobě.
end
 % Zobrazení čistého a šifrovaného textu.
Plaintext_Array_cfb = dec2hex(Plaintext_Array)
Ciphertext_array_cfb = dec2hex(Ciphertext_array)

%% Šifrování CFB8-AES128 využitím funkce encryption_cfb8
pVector = libpointer('uint32Ptr', Vector);
calllib('aes','initvector_setup', pVector);

Size = size(Plaintext_Array);
j = Size(1);
      % Zjištění počtu bloků čistého textu.
Ciphertext_array = zeros(j,1);
      % Vytvoření pole pro ukládání šifrovaného textu.
for i = 1:j
  % Smyčka se provede i-krát = počet bloků čistého textu.
  Plaintext = Plaintext_Array(i);
      % uložení i-tého čistého textu do proměnné Plaintext
  Ciphertext_array(i)  = calllib('aes', 'encryption_cfb8', Plaintext);
      % návratová hodnota funkce typu int je uložena do pole
      % s šifrovaným textem
end
% Zobrazení čistého a šifrovaného textu.
Plaintext_Array_cfb8 = dec2hex(Plaintext_Array)
Ciphertext_array_cfb8 = dec2hex(Ciphertext_array)

%% Vymazání knihovny z paměti
unloadlibrary aes
```

# G Příloha 7 – obsah přiloženého CD

Přiložené CD obsahuje:

- elektronickou verzi diplomové práce uložené v PDF,
- zdrojový kód knihovny aesdll.asm,
- překladač NASM a linker ALINK,
- dynamickou knihovnu aes.dll,
- hlavičkový soubor aesdll.h,
- skripty do programu Matlab (přípona .m) využívající funkcí knihovny aes.dll.
    - aes_ecb – provádí a měří doby trvání šifrování, dešifrování v módu ECB,
    - aes_cbc – provádí a měří doby trvání šifrování, dešifrování v módu CBC, měří doby trvání expanze klíčů,
    - aes_ctr – provádí a měří doby trvání šifrování, dešifrování v módu CTR, zobrazuje hodnoty čítače,
    - aes_cfb – provádí a měří doby trvání šifrování, dešifrování v módu CFB pro velikost klíče 128 bitů,
    - aes_ofb – provádí a měří doby trvání šifrování, dešifrování v módu OFB a délku zpracovávaných bloku $s = 128$,
    - aes_ofb8 – provádí a měří doby trvání šifrování, dešifrování v módu OFB a délku zpracovávaných bloku $s = 8$,
    - cfb8_aes128 – provádí šifrování v módu CFB a délky zpracovávaných bloků $s = 8$ (pomocí funkcí `encryption_cfb` a `encryption_cfb8`), zobrazuje hodnoty vstupních bloků,
    - cfb8_aes128_cmp – provádí a měří dobu trvání šifrování v módu CFB a délky zpracovávaných bloků $s = 8$ (pomocí funkcí `encryption_cfb` a `encryption_cfb8`),
    - cfb_file_enc – šifruje 4,5MB audio soubor pomocí funkcí `encryption_cfb` a `encryption_cfb8` a měří dobu trvání šifrování,
    - cfb8_file_encryption_decryption – šifruje a dešifruje 4,5MB audio soubor pomocí funkce `encryption_cfb8` a měří dobu trvání šifrování,
    - allionone – spouští všechny předešlé skripty.