VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

AKCELERACE VEKTOROVÝCH A KRYTOGRAFICKÝCH OPERACÍ NA PLATFORMĚ X86-64
ACCELERATION OF VECTOR AND CRYPTOGRAPHIC OPERATIONS ON X86-64 PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Samuel Šlenker**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Miroslav Balík, Ph.D.**
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**
Ústav telekomunikací

Student: Bc. Samuel Šlenker ***ID:*** 154887
Ročník: 2 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Akcelerace vektorových a krytografických operací na platformě x86-64

POKYNY PRO VYPRACOVÁNÍ:

Připravte prostředí pro vytváření 64bitových dynamických knihoven v jazyce symbolických instrukcí. Porovnejte funkce pro akceleraci vybraných algebraických operací s vektory a maticemi s využitím jednotek SSE, AVX a FMA v jednoduché a dvojité přesnosti. Připravte laboratorní úlohy, které postupně povedou k vytvoření knihovny s akcelerovaným algoritmem AES-GCM. Součástí úloh bude testování akcelerace AES šifrování pro různé velikosti bloku s použitím AES-NI a naprogramování a test algoritmu AES-GCM s využitím instrukcí AES-NI a CLMUL.

DOPORUČENÁ LITERATURA:

[1] PAAR, Christof; PELZL, Jan. Understanding cryptography: a textbook for students and practitioners. Springer Science & Business Media, 2009. ISBN 3642041019

[2] GUERON, Shay; KOUNAVIS, Michael E. Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode. White Paper, [online], 2010. Dostupné z https://www.intel.ph/content/dam/www/public/us/en/documents/white-papers/carry-less-multiplication-instruction-in-gcm-mode-paper.pdf

Termín zadání: 1.2.2017 ***Termín odevzdání:*** 24.5.2017

Vedoucí práce: Ing. Miroslav Balík, Ph.D.
Konzultant:

doc. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Cieľom práce bolo naštudovať a následne spracovať porovnanie starších a novších vektorových výpočtových jednotiek moderných mikroprocesorov na platforme x86-64. Práca mala poskytnúť prehľad najrýchlejších výpočtov vektorových operácií s maticami a vektormi spolu s príslušnými zdrojovými kódmi. Ďalej bola jej zameraním oblasť autentizovaného šifrovania, konkrétne blokovej šifry AES pracujúcej v operačnom móde Galois Counter Mode a pojednanie o možnostiach inštrukčných sád pre podporu kryptografie.

KĽÚČOVÉ SLOVÁ

SSE, AVX, FMA, SIMD, vektorové spracovanie dát, AES, GCM, AES-NI, CLMUL, SHA Extensions

ABSTRACT

The aim of this thesis was to study and subsequently process a comparison of older and newer SIMD processing units of modern microprocessors on the x86-64 platform. The thesis provides an overview of the fastest computations of vector operations with matrices and vectors, including corresponding source codes. Furthermore, the thesis is focused on authenticated encryption, specifically on block cipher AES operating in Galois Counter Mode, and on a discussion of possibilities of instruction sets for cryptographic support.

KEYWORDS

SSE, AVX, FMA, SIMD, Vector Data Processing, AES, GCM, AES-NI, CLMUL, SHA Extensions

ŠLENKER, Samuel *Akcelerace vektorových a krytografických operací na platformě x86-64*: diplomová práca. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2016. 110 s. Vedúci práce bol Ing. Miroslav Balík, PhD.

PREHLÁSENIE

Prehlasujem, že som svoju diplomovú prácu na tému „Akcelerace vektorových a kryptografických operací na platformě x86-64“ vypracoval samostatne pod vedením vedúceho diplomovej práce, využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej diplomovej práce ďalej prehlasujem, že v súvislosti s vytvorením tejto diplomovej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákoníka č. 40/2009 Sb.

Brno ....

podpis autora

POĎAKOVANIE

Rád by som sa poďakoval vedúcemu diplomovej práce pánovi Ing. Miroslavovi Balíkovi, PhD. za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Brno . . . . . .

. . . . . .

podpis autora

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

POĎAKOVANIE

Výskum popísaný v tejto diplomovej práci bol realizovaný v laboratóriách podporených projektom SIX; registračné číslo CZ.1.05/2.1.00/03.0072, operačný program V zkum a vývoj pro inovace.

Brno . . . . . .

. . . . . .

podpis autora

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OBSAH

ZOZNAM OBRÁZKOV

ZOZNAM TABULIEK

ÚVOD

Prvá časť práce je zameraná na vektorové operácie s maticami a vektormi. Pokročilé operácie využívané najmä v 3D grafike mali dopad na vznik vektorových výpočtov. Keďže v tejto oblasti sa často využívajú maticové a vektorové výpočty, začali sa k vektorovému spracovaniu vyvíjať aj výpočtové jednotky procesorov. Vznik vektorových výpočtových jednotiek priniesol možnosť spracovať v jednom kroku viacero hodnôt.

Zameraním tejto časti práce je poskytnutie porovnania staršej a novšej generácie vektorových výpočtových jednotiek procesorov, a to s dôrazom na platformu x86-64, ale tiež porovnanie s programovím prostredím x86-32. Toto porovnanie je realizované testovaním zdrojových kódov v jazyku symbolických inštrukcií pre operáciu násobenia vektora maticou, resp. matice vektorom. Pri relizovaní testovania sú využité vlastnosti výpočtových jednotiek, ktoré hrali zásadnú úlohu v optimalizácií samotných zdrojových kódov. Tie boli vytvárané so zámerom poskytnúť čo najoptimálnejšie výsledky.

Druhá časť práce sa venuje autentizovanému šifrovaniu, konkrétne blokovej šifre Advanced Encryption Standard (AES) pracujúcej v operačnom móde Galois Counter Mode (GCM). Bloková šifra AES patrí medzi najpoužívanejšie šifry súčasnosti. V tejto práci je popísaný všeobecný koncept symetrickej kryptografie, na ktorý nadväzuje popis operačného módu GCM a popis samotného šifrovania a dešifrovania blokovej šifry AES, spolu s vysvetlením jednotlivých transformácií a operácií používaných ako pri šifrovaní, tak pri dešifrovaní (inverzná šifra). Práca poukazuje na výhody využitia operačného módu GCM, rovnako ako aj na jeho komplexnosť a celkový koncept.

Vývoj vektorových výpočtových jednotiek a kryptografie mal za následok to, že výpočtové jednotky procesorov boli vybavené inštrukciami pre vektorové spracovanie kryptografických operácií, čo malo viesť k zrýchleniu týchto výpočtov. V práci sú uvedené inštrukčné sady, ktoré toto zrýchlenie poskytujú, spolu s prehľadom a popisom pridaných kryptografickcýh inštrukcií.

1 VEKTOROVÉ OPERÁCIE S VYUŽITÍM VÝPOČTOVÝCH SIMD JEDNOTIEK

Časť diplomovej práce o vektorových operáciách s využitím výpočtových SIMD (Single Instruction-Multiple Data) jednotiek priamo nadväzuje na bakalársku prácu [1]. Bakalárska práca bola zameraná na urýchlenie výpočtov vybraných matematických operácií, ktoré v moderných procesoroch zabezpečujú SIMD výpočtové jednotky SSE (Streaming SIMD Extensions), AVX (Advanced Vector Extensions) a FMA (Fused Multiply-Add). Už z ich všeobecného označenia vyplýva, že výpočty realizované pomocou týchto jednotiek sú akcelerované vďaka faktu, že v jednom kroku sa vykoná rovnaká operácia na viacerých dátach. Využité algebraické operácie práci sú stručne popísané v nasledujúcich podkapitolách.

1.1 Transpozícia vektora a matice

Transpozíciu vektora je nevyhnutné použiť pri násobení zľava, tj. pri násobení vektora maticou. Pri tomto type násobenia sa každý stĺpec matice vynásobí s riadkovo orientovaným vektorom. V poslednom kroku sa tieto hodnoty po násobení sčítajú do daného prvku výsledného vektora. Je dôležité uvedomiť si, že počítač tento výpočet realizuje odlišne, ako človek. Pri realizácií tejto operácie pomocou výpočtových jednotiek procesoru je nutné brať v úvahu tri zásadné fakty:

1. Hodnoty vektoru a matice sú v dátových registroch uložené v opačnom poradí, pretože sa načítavajú do registrov od najnižších adries po najvyššie, ktoré sú vpravo. V prípade matice

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

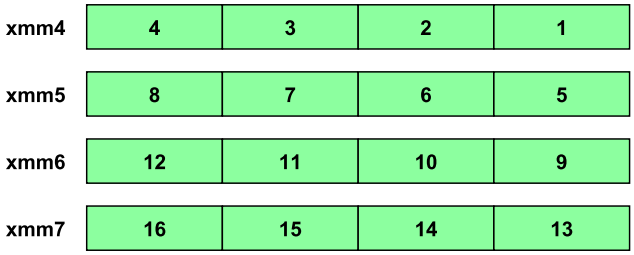
   zobrazuje uloženie jej hodnôt do registrov obrázok 1.1.
2. Pri výpočte pomocou vektorových jednotiek nie sme schopní vykonať násobenie vektora s prvým riadkom matice, pretože prvý riadok matice je uložený až v štyroch registroch.
3. Pri vektorových operáciách platí, že dátové registre, v ktorých sú uložené hodnoty pre výpočet, pracujú „pod sebou“.

Jednou z možností, ako výpočet uskutočniť, je transpozícia vektora. Pri výpočte využijeme skutočnosť, že prvý riadok matice je násobený prvým prvkom vektora, druhý riadok druhým prvkom vektora atď. Pre transponovanie vektora sa využili dva

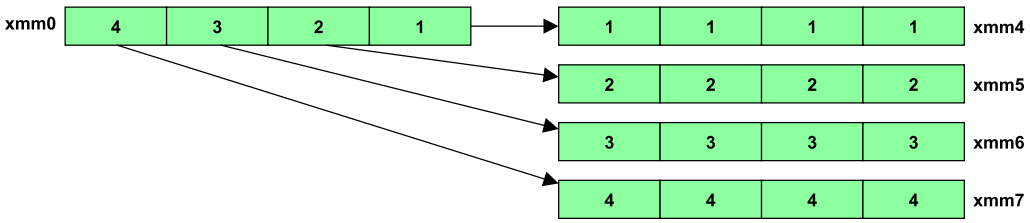
typy inštrukcií pre vektorovú jednotku SSE a jedna špeciálne inštrukcia pre AVX a FMA. Touto špeciálnou inštrukciou výpočtové jednotka SSE nedisponuje:

- SSE - inštrukcia **shufps** (Shuffle Packed Single-Precision Floating-Point Values).
- SSE - inštrukcia **unpcklps/unpckhps** (Unpack and Interleave Low/High Packed Single-Precision Floating-Point Values)
- AVX/FMA - inštrukcia **broadcastss** (Broadcast Floating-Point Data).

Všetky tri typy inštrukcií zabezpečia rovnaký výsledok v prípade, že budú použité správne. Výsledkom je transponovaný vektor v podobe matice, pretože je uložený až v štyroch dátových registroch. Každý prvok vektora zo vstupného registra sa „natiahne“ do samostatného registra (obr. 1.2). Tento fakt je potom využitý pri násobení vektora maticou, ktoré je popísané v nasledujúcej podkapitole. Podrobný popis transpozície vektora poskytuje [1] (kap. 6.1.1, s. 36).



Obr. 1.1: Uloženie hodnôt matice v XMM registroch



Obr. 1.2: Transponovaný vektor v štyroch dátových registroch

Transpozícia matice je z hľadiska vektorových jednotiek časovo náročná operácia a je dobré sa jej vyhnúť. Pri tejto operácii sa riadky matice vymenia s jej stĺpcami.

$$A = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}, A^T = \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

Pre transponovanie matice pri výpočtovej jednotke SSE boli využité tzv. rozbaľovacie inštrukcie (unpack). Inštrukčné sady AVX a novšie poskytujú permutačné inštrukcie, ktoré je taktiež možné pri tejto operácii použiť. Pri vhodne zvolenej postupnosti týchto inštrukcií je z počiatočnej matice odvodená jej transponovaná verzia [1].

1.2 Násobenie vektora maticou

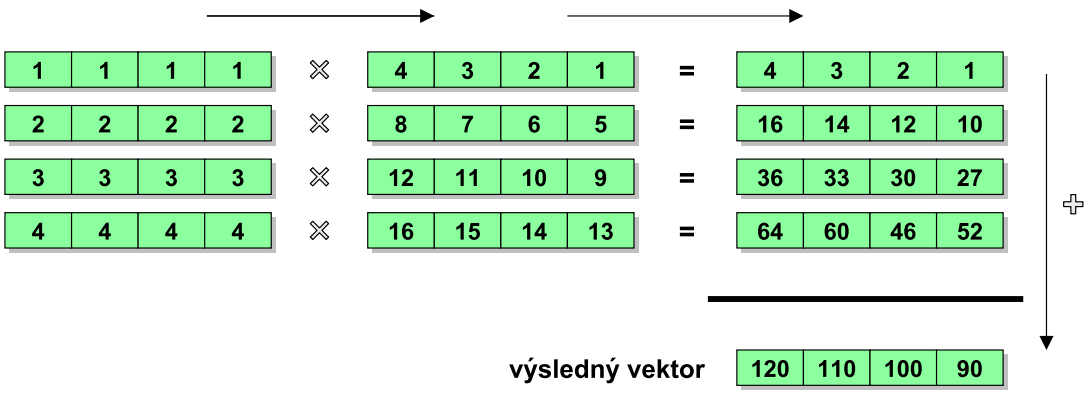
Ukážkový príklad a zápis násobenia zľava:

$$\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix} \times \begin{bmatrix}1 & 2 & 3 & 4\\ 5 & 6 & 7 & 8\\ 9 & 10 & 11 & 12\\ 13 & 14 & 15 & 16\end{bmatrix}$$

Výsledkom operácie je riadkovo orientovaný vektor. Platí, že prvý prvok vektora (číslo 1) sa násobí s každým prvkom prvého riadku matice (čísla 1, 2, 3, 4), druhý prvok vektora (číslo 2) sa násobí s každým prvkom druhého riadku matice (čísla 5, 6, 7, 8) atď. Preto je možné pri násobení vektora maticou využíť poznatky z predošlej podkapitoly, teda operáciu transpozície vektora. Ako bolo uvedené, transponovaný vektor bude mať z hľadiska uloženia hodnôt v dátových registroch charakter matice a algebraicky je možné zobraziť vektor po transpozícii nasledovne:

$$\begin{bmatrix}1 & 2 & 3 & 4\end{bmatrix} \longrightarrow \begin{bmatrix}1 & 1 & 1 & 1\\ 2 & 2 & 2 & 2\\ 3 & 3 & 3 & 3\\ 4 & 4 & 4 & 4\end{bmatrix}$$

S takto transponovaným vektorom stačí vykonať jednoduchú operáciu vektorového násobenia s maticou pomocou inštrukcií násobenia. Medzivýsledkom tohto kroku je matica násobených hodnôt, ktoré sa sčítavajú „pod sebou“ do výsledného vektora pomocou základných inštrukcií vektorového sčítania. Celý proces výpočtu ilustruje obrázok 1.3. Podrobné vysvetlenie operácie násobenia zľava poskytuje [1] (kap. 6.1.2, s. 38).



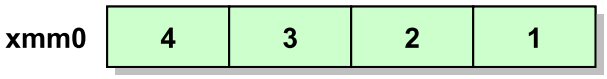
Obr. 1.3: Postup výpočtu násobenia zľava

1.3 Násobenie matice vektorom

Ukážkový príklad násobenia sprava

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix} \times \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

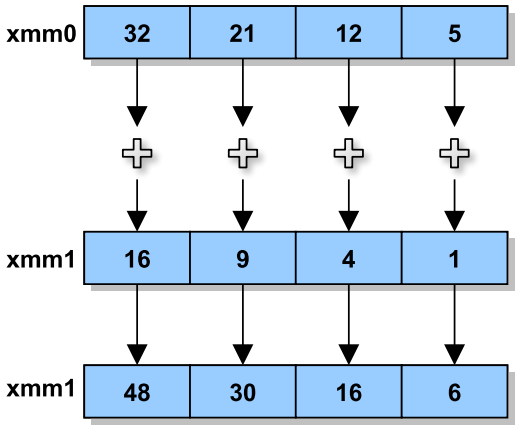
Z pohľadu výpočtových jednotiek od inštrukčnej sady SSE3 je násobenie sprava teoreticky výhodná operácia, pretože si nevyžaduje transpozíciu vektora. Jednotlivé prvky vektora sú nahrávané do jedného dátového registra, ktorý je „horizontálny“, čo zabezpečí jeho okamžitú transpozíciu bez nutnosti používať ďalšie inštrukcie. Uloženie ukážkového stĺpcovo orientovaného vektora je na obrázku 1.4.



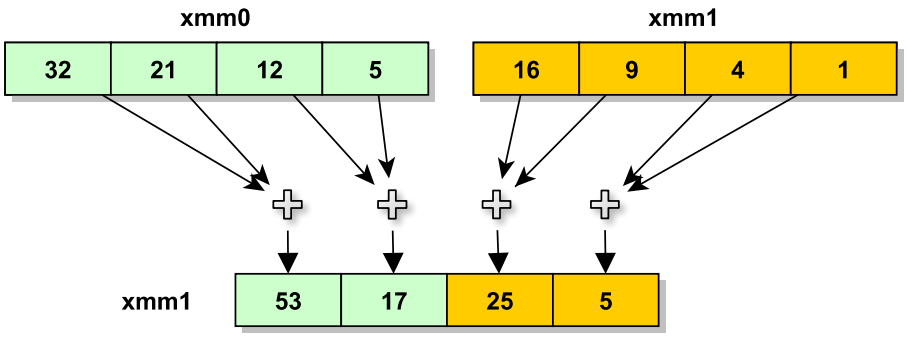
Obr. 1.4: Uloženie stĺpcovo orientovaného vektora v dátovom registri

Z obrázku je zjavná okamžitá transpozícia so stĺpcovo orientovaného vektora na riadkovo orientovaný vektor. Na druhej strane, aby sme dostali správny výsledný vektor, je nutné medzivýslednú maticu po násobení matice vektorom transponovať, čo je časovo náročná operácia. Od inštrukčnej sady SSE3 sú však k dispozícii inštrukcie horizontálneho sčítania, ktoré boli využité pri sčítaní medzihodnôt po násobení a teoreticky by mali transpozíciu medzivýslednej matice urýchliť. Bez týchto inštrukcií by bolo potrebné pridať relatívne veľký počet riadkov do zdrojového kódu,

pretože hodnoty, ktoré sa sčítavajú po medzikroku násobenia, nie sú pod sebou, ale vedľa seba. Inštrukcia pre horizontálne sčítanie má názov **haddps** (Packed Single-FP Horizontal Add) a jej znázornenie, spolu s porovnaním vertikálneho sčítania, je na obrázku 1.5. Detailný popis násobenia sprava je možné nájsť v [1] (kap. 6.1.3, s. 38).



a) Vertikálne sčítanie



b) Horizontálne sčítanie

Obr. 1.5: Postup sčítavania pomocou vertikálneho a horizontálneho sčítania

1.4 Výpočty optimalizované pre výpočtové jednotky SSE, AVX a FMA

Je nutné uviesť, že nižšie uvedené zdrojové kódy sú z [1], kde sa programovalo na platforme x86-32. Od nich sa odvíjali zdrojové kódy v diplomovej práci pre platformu x86-64. Aj keď v samotných zdrojových kódoch výpočtov neboli podstatné rozdiely medzi oboma platformami, niektoré zmeny je možné vidieť pri porovnaní nižšie uvedených výpočtov s tými novo vytvorenými, ktoré sú uvedené v praktickej časti tejto práce.

Táto kapitola poskytuje optimalizované zdrojové kódy pre výpočtové jednotky SSE, AVX a FMA pre výpočet:

- násobenia matice vektorom v jednoduchej presnosti,
- násobenia matice vektorom v dvojitej presnosti,
- násobenia vektoru maticou v jednoduchej presnosti,
- násobenia vektoru maticou v dvojitej presnosti.

Teoretické urýchlenie výpočtu pomocou výpočtovej jednotky FMA je možné aplikovať len na výpočet násobenia vektora maticou. Podmienkou využitia jednotky FMA je, že bezprostredne po sebe nasledujú operácia násobenia a sčítania. Táto podmienka je splnená len v prípade násobenia zľava. Preto pre násobenie matice vektorom táto výpočtová jednotka nebola použitá. Nižšie uvedené výpočty boli vybraté na základe výsledkov testov v [1] spomedzi viacerých testovaných zdrojových kódov.

Kvôli obmedzeniu veľkosti dátových registrov pracujú všetky zdrojové kódy s veľkosťou matice 4×4 a s veľkosťou vektora 4×1, resp. 1×4. Tieto rozmery sú dané veľkosťou dátových registrov a použitých presností. Dátové registre s označením XMM majú veľkosť 128 bitov. Jedno číslo v jednoduchej presnosti má veľkosť 32 bitov. Tým pádom je možné do jedného XMM registra uložiť štyri čísla v jednoduchej presnosti. Od výpočtovej jednotky AVX sú k dispozícii okrem XMM registrov aj dátové registre veľkosti 256 bitov, ktoré majú označenie YMM. Jedno číslo v dvojitej presnosti má veľkosť 64 bitov a preto sa do jedného YMM registra zmestia štyri čísla v dvojitej presnosti. Podrobnejší popis dátových registrov a dátových typov pre SSE poskytuje [1] kapitola 2.1 (s. 15) a pre AVX kapitola 2.2 (s. 17).

Výpočet je realizovaný pomocou vnoreného cyklu. V prípade vstupného vektoru a matice väčšej veľkosti sa v jednom cykle pracuje so submaticou a subvektorom 4×4. Nasledujúce zdrojové kódy demonštrujú výpočet v jednom cykle. Detailný popis každého kroku zdrojového kódu, inštrukcií a celkového konceptu výpočtov je možné nájsť v [1] od kapitoly 7 (s. 42).

1.4.1 Násobenie matice vektorom v jednoduchej presnosti

Výpočtová jednotka SSE

Pri násobení sprava výpočtovou jednotkou SSE bol najrýchlejší výpočet pomocou inštrukcií horizontálneho sčítania. Tieto inštrukcie sú k dispozícii od inštrukčnej sady SSE3. Nasleduje zdrojový kód výpočtu.

```
; xmm0 = subvektor
; xmm4 až xmm7 = submatica

mulps xmm4,xmm0
mulps xmm5,xmm0
mulps xmm6,xmm0
mulps xmm7,xmm0

haddps xmm4,xmm5
haddps xmm6,xmm7
haddps xmm4,xmm6
movups [edi],xmm4 ; uložíme výsledok, ktorý je v xmm4
```

Výpočtová jednotka AVX

U výpočtovej jednotky AVX bol výsledok rovnaký, ako u SSE. Najrýchlejší výpočet poskytovali inštrukcie horizontálneho sčítania. Nasleduje zdrojový kód výpočtu.

```
; xmm0 = subvektor
; xmm4 až xmm7 = submatica

vmulps xmm4,xmm4,xmm0
vmulps xmm5,xmm5,xmm0
vmulps xmm6,xmm6,xmm0
vmulps xmm7,xmm7,xmm0

vhaddps xmm4,xmm4,xmm5
vhaddps xmm6,xmm6,xmm7
vhaddps xmm4,xmm4,xmm6
vaddps xmm2, xmm2, xmm4
vmovups [edi],xmm2 ; výsledok v xmm2
```

1.4.2 Násobenie matice vektorom v dvojitej presnosti

V dvojitej presnosti bolo násobenie sprava realizované len pomocou jednotky AVX. Ako už bolo zmienené, inštrukcie FMA neposkytujú žiadne urýchlenie tohoto typu výpočtu. Realizovať bolo možné len jeden zdrojový kód pre výpočtovú jednotku AVX. Výpočet bol realizovaný pomocou transpozície matice po násobení s využitím rozbaľovacích a permutačných inštrukcií.

```
; ymm7 = subvektor
; ymm0 až ymm3 = submatica
vmulpd ymm0,ymm0,ymm7
vmulpd ymm1,ymm1,ymm7
vmulpd ymm2,ymm2,ymm7
vmulpd ymm3,ymm3,ymm7

vunpcklpd ymm4, ymm0, ymm1
vunpckhpd ymm0, ymm0, ymm1
vunpcklpd ymm1, ymm2, ymm3
vunpckhpd ymm2, ymm2, ymm3

vperm2f128 ymm3, ymm4, ymm1, 32
vperm2f128 ymm4, ymm4, ymm1, 49
vperm2f128 ymm1, ymm0, ymm2, 32
vperm2f128 ymm2, ymm0, ymm2, 49

vaddpd ymm3,ymm3,ymm4
vaddpd ymm1,ymm1,ymm2
vaddpd ymm1,ymm1,ymm3
vaddpd ymm6,ymm6,ymm1

vmovupd [edi],ymm6 ; výsledok v ymm6
```

1.4.3 Násobenie vektoru maticou v jednoduchej presnosti

Výpočtová jednotka SSE

U výpočtovej jednotky sa realizovalo niekoľko výpočtov násobenia zľava. Najlepší výsledok dosiahol výpočet pomocou rozbaľovacích inštrukcií, ktorého zdrojový kód je nižšie.

```
; xmm0 = subvektor
; xmm4 až xmm7 = submatica
movups xmm1,xmm0
unpcklps xmm1, xmm1 ; xmm1 = v2, v2, v1, v1
movups xmm2,xmm1
unpcklps xmm1, xmm1
unpckhps xmm2, xmm2
mulps xmm1, xmm4
addps xmm3, xmm1
mulps xmm2, xmm5
addps xmm3, xmm2

movups xmm1,xmm0
unpckhps xmm1, xmm0
movups xmm2,xmm1
unpcklps xmm1, xmm1
unpckhps xmm2, xmm2
mulps xmm1, xmm6
addps xmm3, xmm1
mulps xmm2, xmm7
addps xmm3, xmm2
movups [edi],xmm3 ; uloženie výsledku v xmm3
```

Výpočtová jednotka AVX

Výpočty pomocou AVX poskytli oproti SSE urýchlenie, aj keď nie výrazné. Najrýchlejší spôsob výpočtu bol pomocou inštrukcií pomiešania. Nasleduje zdrojový kód.

```
; xmm0 = subvektor
; xmm4 až xmm7 = submatica
vshufps xmm1,xmm0,xmm0,0x00
vmulps xmm4,xmm1,xmm4
vaddps xmm2,xmm2,xmm4
vshufps xmm1,xmm0,xmm0,0x55
vmulps xmm5,xmm1,xmm5
vaddps xmm2,xmm2,xmm5
vshufps xmm1,xmm0,xmm0,0xAA
vmulps xmm6,xmm1,xmm6
```

```
vaddps xmm2,xmm2,xmm6
vshufps xmm1,xmm0,xmm0,0xFF
vmulps xmm7,xmm1,xmm7
vaddps xmm2,xmm2,xmm7
movups [edi],xmm2 ; výsledok v xmm2
```

Výpočtová jednotka FMA

U výpočtovej jednotky boli realizované dva rôzne výpočty násobenia zľava v jednoduchej presnosti. Podobne ako u AVX poskytli najrýchlejší výpočet inštrukcie pomiešania. I keď teoreticky by mal výpočet trvať kratšie, ako u jednotky AVX, jednotka FMA bola o niekoľko stotín sekundy pomalšia, a to aj napriek faktu, že jedna inštrukcia FMA nahradzuje dve inštrukcie AVX (inštrukcia násobenia a inštrukcia sčítania je nahradená jednou inštrukciou, ktorá vykoná obe tieto operácie súčasne).

```
; xmm0 = subvektor
; xmm4 až xmm7 = submatica
vshufps xmm1,xmm0,xmm0,0x00
vfmadd231ps xmm2,xmm4,xmm1

vshufps xmm1,xmm0,xmm0,0x55
vfmadd231ps xmm2,xmm5,xmm1

vshufps xmm1,xmm0,xmm0,0xAA
vfmadd231ps xmm2,xmm6,xmm1

vshufps xmm1,xmm0,xmm0,0xFF
vfmadd231ps xmm2,xmm7,xmm1
movups [edi],xmm2 ; výsledok v xmm2
```

1.4.4 Násobenie vektoru maticou v dvojitej presnosti

Výpočtová jednotka AVX

V dvojitej presnosti bolo násobenie zľava testované u AVX formou dvoch typov inštrukcií. Rýchlejší spôsob výpočtu poskytla kombinácia rozbaľovacích a permutačných inštrukcií. Nasleduje zdrojový kód.

```
; ymm7 = subvektor
; ymm0 až ymm3 = submatica
vunpcklpd ymm4,ymm7,ymm7
```

```
vperm2f128 ymm4,ymm4,ymm4,0
vmulpd ymm0, ymm0, ymm4
vaddpd ymm6, ymm6, ymm0

vunpckhpd ymm4,ymm7,ymm7
vperm2f128 ymm4,ymm4,ymm4,0
vmulpd ymm1, ymm1, ymm4
vaddpd ymm6, ymm6, ymm1

vunpcklpd ymm4,ymm7,ymm7
vperm2f128 ymm4,ymm4,ymm4,17
vmulpd ymm2, ymm2, ymm4
vaddpd ymm6, ymm6, ymm2

vunpckhpd ymm4,ymm7,ymm7
vperm2f128 ymm4,ymm4,ymm4,17
vmulpd ymm3, ymm3, ymm4
vaddpd ymm6, ymm6, ymm3
vmovupd [edi],ymm6
```

Výpočtová jednotka FMA

Pomocou výpočtovej jednotky FMA bol oproti jednotke AVX výpočet násobenia zľava v dvojitej presnosti zrýchlený. Realizovaný bol jeden zdrojový kód, ktorý je uvedený nižšie.

```
; ymm7 = subvektor
; ymm0 až ymm3 = submatica
; esi = ukazateľ na vstupný vektor
vbroadcastsd ymm7, [esi]
add esi,8
vfmadd231pd ymm6,ymm0,ymm7

vbroadcastsd ymm7, [esi]
add esi,8
vfmadd231pd ymm6,ymm1,ymm7

vbroadcastsd ymm7, [esi]
add esi,8
```

```
vfmadd231pd ymm6,ymm2,ymm7

vbroadcastsd ymm7, [esi]
add esi,8
vfmadd231pd ymm6,ymm3,ymm7
vmovupd [edi],ymm6 ; uložíme výsledok z ymm6
```

2 KONCEPT SYMETRICKEJ KRYPTOGRAFIE S BLOKOVOU ŠIFROU

Symetrický systém šifrovania sa skladá z piatich zložiek [2]:

1. **Nešifrovaný text**: Jedná sa o zrozumiteľnú správu alebo dáta, ktoré sú privádzané na vstup šifrovacieho algoritmu.
2. **Šifrovací algoritmus**: Šifrovací algoritmus vykonáva rôzne substitúcie a transformácie na nešifrovanom texte.
3. **Tajný kľúč**: Tajný kľúč je taktiež vstupom šifrovacieho algoritmu a predstavuje hodnotu nezávislú na nešifrovanom texte a algoritme. Výstup algoritmu sa líši v závislosti na aktuálne použitom kľúči. Substitúcie a transformácie vykonané algoritmom závisia na kľúči.
4. **Šifrovaný text**: Predstavuje kódovanú správu, ktorá je na výstupe. Šifrovaný text je závislý na nešifrovanom texte a tajnom kľúči. Pre jednu správu teda dva rôzne kľúče vyprodukujú dva rôzne šifrované texty. Šifrovaný text je zdanlivo náhodný tok dát a je nezrozumiteľný.
5. **Dešifrovací algoritmus**: V podstate sa jedná o šifrovací algoritmus, ktorý pracuje inverzne. Tento algoritmus vezme šifrovaný text a tajný kľúč a vyprodukuje originálny nešifrovaný text.

V prípade symetrického kryptosystému sú kľúč na strane pôvodcu a kľúč na strane adresáta prakticky odvoditeľné jeden z druhého. Preto je nutné, aby boli tieto kľúče utajované pred neoprávenými osobami. Hodnoty kľúča pôvodcu a kľúča adresáta môžu byť rôzne, ale v praxi sa takmer výhradne využívajú kryptosystémy, ktoré sú oba kľúče rovnaké [3]. Z hľadiska utajenia teda majú symetrické postavenie, z čoho je odvodený aj samotný názov symetrického šifrovania, resp. symetrickej šifry. Rozlišujeme dva typy symetrických šifier, podľa spracovania dát. Sú to

- **prúdové šifry**, ktoré spracovávajú prúd bitov,
- **blokové šifry**, ktoré pracujú s blokom pevnej veľkosti.

Táto práca je zameraná na blokové šifry a konkrétne na šifru AES, ktorej sa bude venovať nasledujúca kapitola. Viac informácií o ďalších blokových šífrách, a taktiež o prúdových šifrách, je možné nájsť v [2] [3] [4].

V prípade blokovej šifry sa šifrovanie vykonáva po blokoch o pevnej dĺžke n bitov Každý vstupný blok $Z = z_1, z_2, ..., z_n$ je prakticky n bitové číslo, ktorému funkcia šifrovania priraďuje n bitové číslo $C = c_1, c_2, ..., c_n$ tak, že

$$C = \mathrm{E}(Z, K).$$

Funkcia šifrovania E je pritom konštruovaná tak, že hodnota každého bitu výstupného bloku C závisí zložitým spôsobom na všetkých n bitoch vstupného bloku

Z (tzv. difúzia) a na všetkých k bitoch kľúča K (tzv. konfúzia). Difúzia a konfúzia útočníkovi značne komplikujú odhad hodnoty Z alebo K z hodnoty C a tým sťažujú kryptoanalýzu blokových šifier [3].

Základom blokových šifier sú blokové operácie. V praxi sa najčastejšie stretávame s týmito typmi [3]:

- permutácie,
- substitúcie,
- aritmetické operácie.

Z hľadiska tejto práce je najvýznamnejším typom blokovej operácie špeciálny prípad prostej permutácie, ktorým je **rotácia**. Táto sa používa u šifry AES. Existujú dva typy rotácie [3]:

- rotácia vľavo o k bitov,
- rotácia vpravo o k bitov,

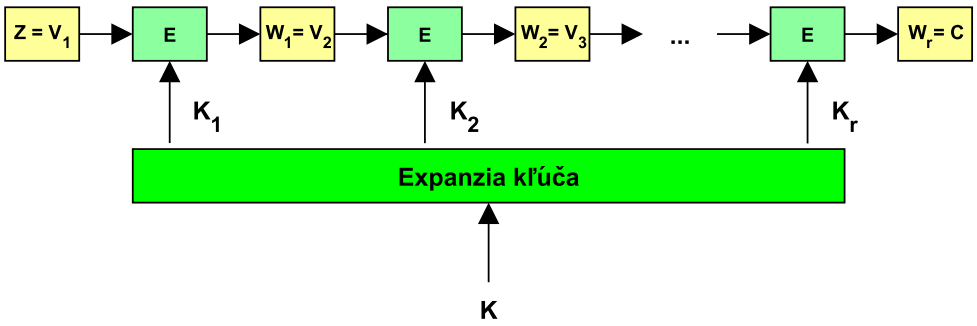
pričom pre premennú k platí, že $0 \leq k < n$.

V [3] je uvedené, že rotácia bloku V o k bitov vľavo sa spravidla označuje $\mathrm{ROL}^k(V)$ podľa anglického „Rotation On Left“. Neformálne je možné túto operáciu definovať tak, že ľavá k-tica bitov vstupného bloku je z bloku odobraná a následne premiestnená na pravú stranu zvyšku bloku. Takto vzniknutý nový blok je výstupom operácie (obr).

Ďalej sa v [3] uvádza, že rotácia bloku V o k bitov vpravo sa označuje $\mathrm{ROR}^k(V)$ podľa anglického „Rotation On Right“. Neformálne je možné túto operáciu definovať tak, že pravá k-tica bitov vstupného bloku je z bloku odobraná a následne premiestnená na ľavú stranu zvyšku bloku. Pokiaľ sa pozrieme na obrázok ilustrujúci rotáciu vľavo o k bitov, tak vidíme, že premiestnenie ľavej k-tice bitov na pravú stranu je ekvivalentné premiestnenie $(n - k)$ pravých bitov na ľavú stranu. Z toho plynie ekvivalencia oboch typov rotácií, ktorú je možné formálne vyjadriť ako

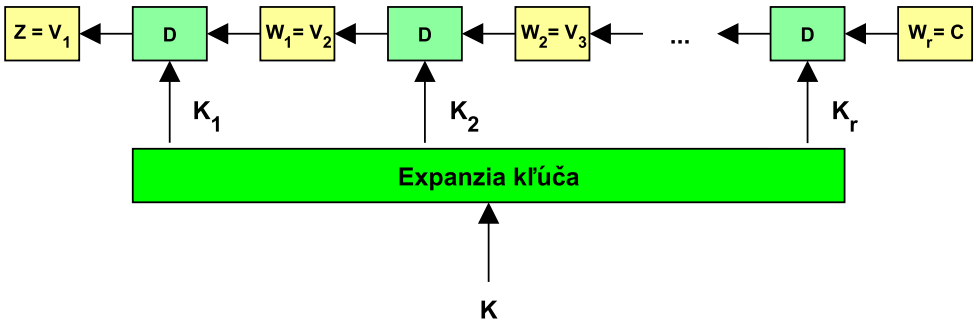
$$\mathrm{ROL}^k(V) = \mathrm{ROR}^{n-k}(V)$$

Podľa [3] sa v praxi často používa variant, kedy je zreťazeno r rovnakých šifier. Tento typ blokovej šifry sa nazýva iterovaná šifra (obr. 2.1). Z šifrovacieho kľúča K sa v bloku expanzie kľúča odvodí r čiastkových (tzv. iteračných) kľúčov. Tieto kľúče K_1 až K_r sú šifrovacími kľúčmi pre iterovanú (opakovane použitú) transformáciu E. Transformáciu E preto budeme ďalej nazývať ako iteráciu („round“). Iterácia je spravidla nejaká relatívne jednoduchá šifra, v ktorej je vstupný blok bitov V_i v závislosti na iteračnom kľúči K_i transformovaný nejakou vhodnou kombináciou blokových operácií do podoby výstupného bloku W_i Vstupný blok šifry Z je privedený na vstupn prvej iterácie E, kde je zašifrovaný do podoby kryptogramu $W_1 = \mathrm{E}(V_1 = Z, K_1)$. Tento blok je vstupom V_2 pre následujúcu iteráciu E. Popísaná procedúra sa opakuje až po poslednú r-tú iteráciu. Výstupom r-tej iterácie E je výstupný blok $W_r = C$.



Obr. 2.1: Všeobecná schéma blokovej šifry pri šifrovaní [3]

Dešifrovanie je u blokovej šifry vykonávané analogicky ako u šifrovania (obr. 2.2) Dešifrovacia transformácia E je inverzná transformácia voči E a inverzné je rovnako aj poradie použitých iteračných kľúčov. V porovnaní s obrázkom 2.1 len jednoducho zmeníme orientáciu vodorovných šípiek.



Obr. 2.2: Všeobecná schéma blokovej šifry pri dešifrovaní [3]

2.1 Operačný mód Counter (CTR) blokovej šifry

Operačných módov blokovej šifry je niekoľko. Z pohľadu tejto práce je však kľúčový operačný mód šifry s názvom Galois Counter Mode (GCM), ktorého šifrovacia časť je založená na princípe operačného módu Counter (CTR). Preto je ešte pred popísaním módu GCM dôležité uviesť princíp šifrovania v operačnom móde CTR. Viac infromácií o všetkých operačných módoch blokovej šifry je možné nájsť v [2] a [4].

Vstupným blokom šifrovania je pri CTR móde čítač. Väčšinou je čítač inicializovaný na nejakú hodnotu a následne sa inkrementuje o 1 pre každý nasledujúci blok (modulo 2^b, kde b je veľkost bloku). Pri šifrovaní je čítač najprv zašifrovaný a potom

sa nad otvoreným textom a zašifrovaným čítačom vykoná exkluzívny logický súčet XOR. Výstupom je blok šifrovaného textu. Pri dešifrovaní sa používa rovnaká sekvencia hodnôt čítača. Každý zašifrovaný čítač je „XORovaný“ s blokom šifrovaného textu, čo vedie k obnoveniu pôvodného bloku otvoreného textu. Z toho vyplýva, že inicializačná hodnota čítača musí byť k dispozícii aj pre dešifrovanie [2].

Jednou z atraktívnych vlastností, ktoré operačný mód CTR ponúka, je možnosť paralelizácie, pretože vo svojej blokovej schéme nemá žiadnu spätnú väzbu, resp. nevyžaduje zreťazenie. Pri šifrovaní a dešifrovaní je tak možné spracovať paralelne viac blokov [4].

2.2 Operačný mód Galois Counter (GCM) blokovej šifry

Operačný mód GCM (Galois Counter Mode) je navrhnutý tak, aby bol paralelizovateľný, čím poskytuje vyššiu priepustnosť a zároveň nižšie oneskorenie, a v celkovom dôsledku teda vyššiu efektivitu [2]. Zvyšok podkapitoly je popísaný podľa [4].

GCM je šifrovací mód, ktorý navyše počíta aj autentizačný kód správy MAC (Message Authentication Code). MAC poskytuje kryptografický kontrolný súčet, vďaka ktorému príjmateľ správy dokáže určiť, že

- originálnu správu skutočne vytvoril odosielateľ,
- behom prenosu nikto nemanipuloval s šifrovaným textom.

Tieto dve vlastnosti teda zaisťujú autentickosť a integritu správy. Pri operačnom móde GCM teda môžeme hovoriť o autentizovanom šifrovaní a dešifrovaní.

GCM chráni dôvernosť otvoreného textu x využitím šifrovania v operačnom móde čítača (CTR), ktorý sa u GCM označuje ako GCTR. GCM navyše chráni aj autentickosť nielen samotného čistého textu x, ale tiež stringu AAD (Additional Authenticated Data). Tieto dodatočné dáta sú v tomto móde, narozdiel od otvoreného textu, ponechané nešifrované. V praxi môže string AAD zahŕňať napr. adresy a parametre v sieťovom protokole.

GCM pozostáva z blokovej šifry, ktorá je základ, a z násobiteľa Galoisovho poľa, ktorým sú realizované funkcie *autentizovaného šifrovania* a *autentizovaného dešifrovania*. Šifra musí mať veľkosť bloku 128 bitov, ako napr. šifra AES. Na strane odosielateľa GCM najskôr zašifruje dáta v móde GCTR a následne vypočíta hodnotu autentizačného kódu správy MAC pomocou funkcie *Galois Hash* (skrátene GHASH). Pri šifrovaní je teda najprv odvodená počiatočná hodnota čítača z inicializačného vektoru [1] a zo sériového čísla. Potom je počiatočná hodnota čítača inkrementovaná

[1] Inicializačný vektor zabezpečuje produkciu rôznych šifrovaných textov z toho istého čistého textu tým, že poskytuje určitú formu náhodnosti.

a táto hodnota je zašifrovaná a „XORovaná“ s prvým blokom nešifrovaného čistého textu. Pre nasledujúce otvorené texty platí, že čítač je inkrementovaný a následne šifrovaný. Základná bloková šifra je použitá len v móde šifrovania. GCM povoluje predvýpočet funkcie blokovej šifry, ak je dopredu známy inicializačný vektor.

Pri autentizácii vykoná GCM zreťazené násobenie Galoisovho poľa. Pre každý otvorený text x_i je odvodený autentizačný parameter g_i. Tento je vypočítaný „XORom“ aktuálneho šifrovaného textu y_i a g_i a následne vynásobený konštantou H. Hodnota H je hešovací podkľúč, ktorý je generovaný šifrovaním nulového vstupu do blokovej šifry. Všetky operácie násobenia sú vykonávané v 128 bitovom Galoisovom poli $GF(2^{128})$ s neredukovateľným polynómom $P(x) = x^{128} + x^7 + x^2 + x + 1$. Keďže pre každú blokovú šifru je potrebné len jedno násobenie, operačný mód GCM pridáva k samotnému procesu šifrovania len veľmi malú réžiu.

3 PROCES ŠIFROVANIA ŠTANDARDU POKROČILÉHO ŠIFROVANIA (AES)

Štandard pokročilého šifrovania (angl. Advanced Encryption Standard) bude popísaný podľa [2].

AES šifra pracuje s blokom otvoreného textu pevnej veľkosti 128 bitov, alebo 16 bytov. Dĺžka šifrovacieho kľúča môža byť 128, 192 alebo 256 bitov. Algoritmy sa potom môžu označovať podľa dĺžky kľúča ako AES-128, AES-192 a AES-256.

Vstupom pre proces šifrovania a dešifrovania je blok pevnej veľkosti 128 bitov, ktorý predstavuje maticu 4×4 (16 bytov). Tento blok je skopírovaný do *stavovej matice*, ktorá je modifikovaná v každom štádiu šifrovania, resp. dešifrovania. V poslednom štádiu je stav matice skopírovaný do výstupnej matice 3.1. Poradie bytov matice je orientované stĺpcovo, tj. prvé štyri bity otvoreného textu sú uložené v prvom stĺpci matice, druhé štyri bity v druhom stĺpci matice atď. To isté platí aj pre expandovaný kľúč.

Šifra pozostáva z N iterácií (rounds), kde ich počet je závislí na dĺžke kľúča:

- 10 iterácií pre kľúč veľkosti 128 bitov (16 bytov).
- 12 iterácií pre kľúč veľkosti 192 bitov (24 bytov).
- 14 iterácií pre kľúč veľkosti 256 bitov (32 bytov).

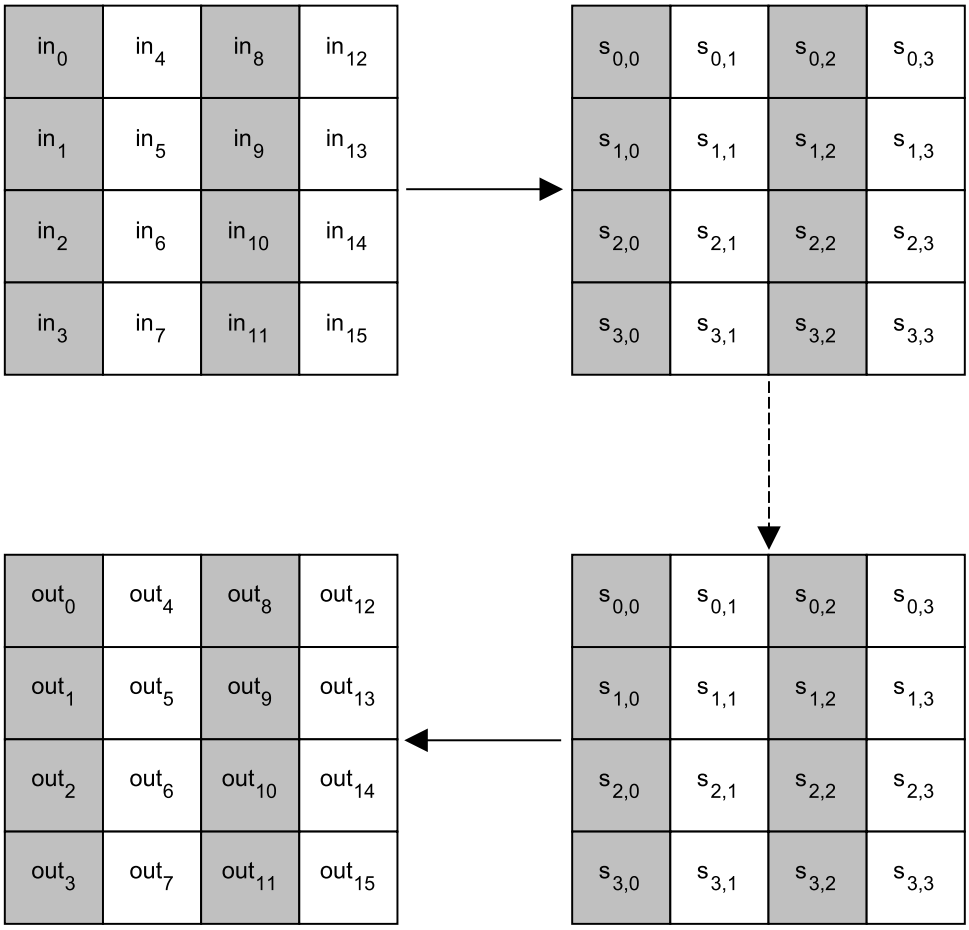
Prvých $N-1$ iterácií pozostáva zo štyroch transformačných funkcií: SubBytes, ShiftRows, MixColumns a AddRoundKey, ktoré budú popísané v nasledujúcej kapitole. Posledná iterácia má už len tri transformácie (bez MixColumns). Okrem toho sa pred prvou iteráciou vykonáva transformačná funkcia AddRoundKey, ktorá môže byť považovaná za nultú iteráciu. Každá transformácia vezme jednu alebo viac matíc veľkosti 4×4 bytov ako vstup a vyprodukuje výstup rovnakej veľkosti. Výstupom poslednej iterácie je šifrovaný text. Proces šifrovania AES zobrazuje obr. 3.2.

3.1 Transformačné funkcie

Ako už bolo zmienené, štandard pokročilého šifrovania AES využíva pri šifrovaní štyri transformačné funkcie. Tieto sú nižšie popísané podľa [2].

3.1.1 Transformácia substitúcie bytov (SubBytes)

Transformácia substitúcie bytov sa skrátene v angličtine označuje ako *SubBytes* (Substitute Bytes). Jedná sa o jednoduchú transformáciu podľa tabuľky. Štandard AES definuje maticu bytových hodnôt rozmeru 16×16 nazývanú *S-box* 3.1, ktorá obsahuje permutácie všetkých možných 8 bitových hodnôt (celkovo 256). Každý



Obr. 3.1: Vstup, stavová matica, výstup [2]

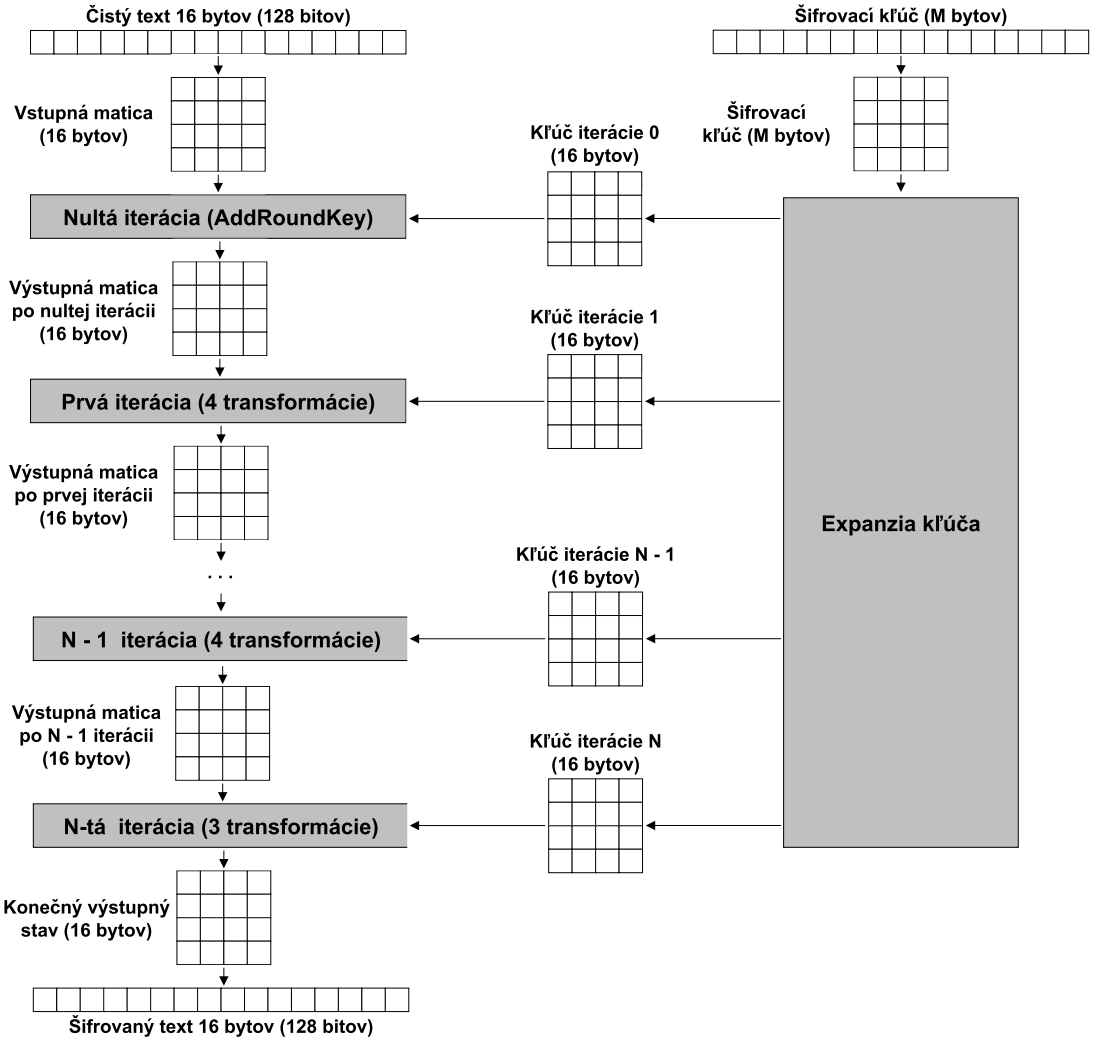
jeden byte stavovej matice je mapovaný na nový byte tak, že štyri bity zľava sú použité ako hodnota riadku a štyri bity sprava ako hodnota stĺpca. Tieto hodnoty slúžia ako index pre výber unikátnej výstupnej 8 bitovej hodnoty z tabuľky S-box.

3.1.2 Transformácia posunu riadkov (ShiftRows)

Transformácia posunu riadkov (ShiftRows) je znázornená na obr. 3.3. Prvý riadok stavovej matice sa nemení. Pre druhý riadok platí posun o jeden byte vľavo. Pre tretí riadok platí posun o dva byty vľavo a pre štvrtý riadok posun o tri byty vľavo.

3.1.3 Transformácia pomiešania stĺpcov (MixColumns)

Táto transformácia pracuje s každým stĺpcom matice samostatne a angl. sa označuje MixColumns. Každý byte stĺpca je mapovaný na novú hodnotu, ktorá je funkciou všetkých štyroch bytov v danom stĺpci. Transformácia pomiešania stĺpcov môže byť



Obr. 3.2: Proces AES šifrovania [2]

definovaná ako nasledovné maticové násobenie stavovej matice.

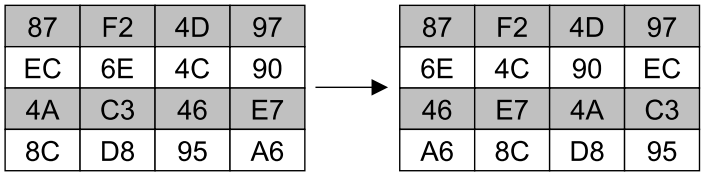
$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

Výber koeficientov matice, ktorou sa násobí stavová matica, bol ovplyvnený úvahami o ich následnom dopade na implementáciu v praxi. Násobenie týchto koeficientov zahŕňa maximálne operácie posunu (shift) a exkluzívny logický súčet (XOR). Koeficienty používané pri dešifrovaní sú iné a z výpočtového hľadiska nie až tak vhodné, avšak šifrovanie je považované za dôležitejšie, ako dešifrovanie.

Vzorec výpočtu transformácie MixColumn pre jeden stĺpec stavovej matice môže

Tab. 3.1: S-box tabuľka [2]

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**S**	**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



Obr. 3.3: Transformácia posunu riadkov [2]

byť vyjadrený ako:

$$s'_{0,i} = (2 \times s_{0,i}) + (3 \times s_{1,i}) + s_{2,i} + s_{3,i}$$

$$s'_{1,i} = s_{0,i} + (2 \times s_{1,i}) + (3 \times s_{2,i}) + s_{3,i}$$

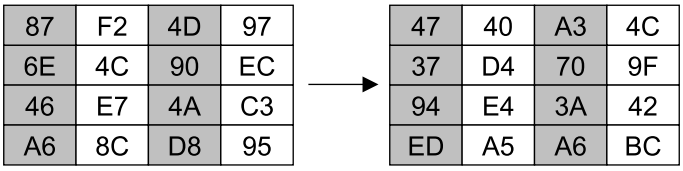
$$s'_{2,i} = s_{0,i} + s_{1,i} + (2 \times s_{2,i}) + (3 \times s_{3,i})$$

$$s'_{3,i} = (3 \times s_{0,i}) + s_{1,i} + s_{2,i} + (2 \times s_{3,i})$$

Príklad transformácie pomiešania stĺpcov znázorňuje obrázok 3.4:

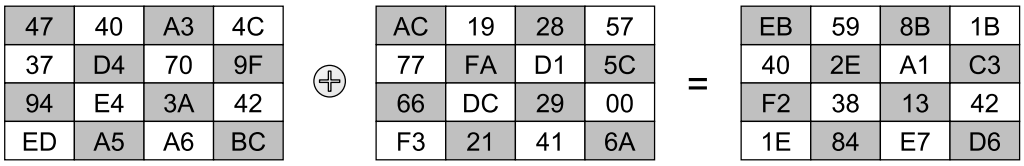
3.1.4 Transformácia pripočítania iteračného kľúča (AddRoundKey)

Transformácia pripočítania iteračného kľúča (AddRoundKey) predstavuje operáciu, kedy sa všetkých 128 bitov stavovej matice „XORuje“ so 128 bitmi iteračného kľúča.



Obr. 3.4: Príklad transformácie pomiešania stĺpcov [2]

Táto transformácia je tak jednoduchá, ako je to len možné. Obrázok 3.5 zobrazuje príklad tejto transformácie. Príklad transformácie pomiešania stĺpcov znázorňuje obrázok 3.4:



Obr. 3.5: Príklad transformácie pripočítania iteračného kľúča [2]

3.2 Expanzia kľúča

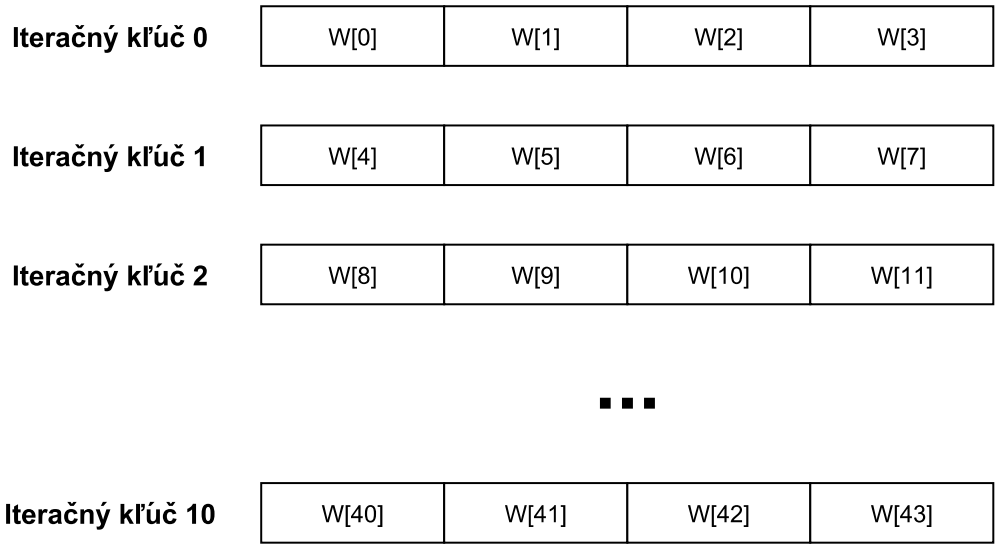
Expanzia kľúča blokovej šifry AES je operácia pracujúca s dátovými typmi *word*, kde 1 word = 32 bitov (4 byty). Iteračné kľúče sú uložené v expanznom poli W, ktoré pozostáva z dátových typov *word*. Pre každú veľkosť AES kľúča (128, 192 a 256 bitov) sa iteračný kľúč expanduje odlišne, aj keď princíp je v podstate veľmi podobný [4].

Nasledujúca podkapitola bude popísaná podľa [2] a [4].

3.2.1 Výpočet expandovaných iteračných kľúčov

Pri tejto veľkosti sa používa 11 iteračných kľúčov (nazývaných tiež *podkľúče (angl. subkeys)*). Tieto sú uložené v expanznom poli, ktoré obsahuje prvky $W[0], ..., W[43]$. Každý jeden prvok $W[x]$ má veľkosť jedného *wordu*, teda 32 bitov (4 byty). Ďalej platí, že veľkosť každého z jedenástich iteračných kľúčov má veľkosť 16 bytov (iteračný kľúč 0 pozostáva z prvkov $W[0], W[1], W[2], W[3]$, iteračný kľúč 1 z prvkov $W[4], W[5], W[6], W[7], atď$) (obr. 3.6).

V prvom kroku sa skopíruje originálny šifrovací kľúč do prvých štyroch prvkov iteračného poľa, z čoho vyplýva, že iteračný kľúč 0 sa rovná originálnemu šifrovaciemu kľúču. Ďalšie iteračné kľúče 1 až 10 sú počítané nasledovným spôsobom.



Obr. 3.6: Označenie prvkov iteračných kľúčov pre šifrovací kľúč 128 bitov

Každý ľavý byte expanzného poľa $W[4i]$, kde $i = 1$ až 10 (tj. $W[4], W[8], ..., W[40]$) sa počíta ako

$$W[4i] = W[4(i-1] + g(W[4i-1]),$$

kde $g()$ je nelineárna funkcia so 4-bytovým vstupom a výstupom. Zvyšné tri prvky iteračných kľúčov sa počítajú rekurzívne ako

$$W[4i+j] = W[4i+j-1] + W[4(i-1)+j],$$

kde $i = 1, ..., 10$ a $j = 1, 2, 3$. Funkcia g() pozostáva z:

1. operácie RotWord, ktorá vykonáva posun vľavo o jednu pozíciu. To znamená, že vstupné slovo $[B_0, B_1, B_2, B_3]$ je transformované do podoby $[B_1, B_2, B_3, B_0]$,
2. operácie SubWord, kde sa vykonáva bytovo orientovaná substitúcia pomocou tabuľky S-box,
3. operácie XOR, kde výsledok kroku 1 a 2 je „XORovaný“ s iteračným koeficientom RC[j].

Iteračný koeficient je prvok Galoisovho poľa $GF(2^8)$. Tento koeficient sa mení s každou ďalšiou iteráciou podľa tabuľky 3.2. Pre rôzne veľkosti AES šifrovacieho kľúča

Tab. 3.2: Hodnoty iteračného koeficientu RC[j]

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

sa používa rôzny počet iteračných koeficientov:

- Pri veľkosti kľúča 128 bitov sa používa všetkých 10 iteračných koeficientov.
- Pri veľkosti kľúča 192 bitov sa používa prvých 8 iteračných koeficientov.
- Pri veľkosti kľúča 256 bitov sa používa prvých 7 iteračných koeficientov.

Pri expanzii kľúča veľkosti 256 bitov sa okrem vyššie zmienených operácií vykonáva navyše operácia SubWord (tj. substitúcia pomocou tabuľky S-box) pre dvojslova (doubleword), ktoré sú násobkom čísla 4 a nie sú násobkom čísla K, kde K je počet dvojslov šifrovacieho kľúča. Pri šifrovacom kľúči veľkosti 256 bitov platí, že $K = 8$.

3.3 Inverzná šifra

V predchádzajúcich kapitolách boli popísané využívané transformácie blokovej šifry AES. Ich poradie pri šifrovaní je nasledovné:

1. Substitúcia bytov (SubBytes).
2. Posun riadkov (ShiftRows).
3. Pomiešanie stĺpcov (MixColumns).
4. Pripočítanie iteračného kľúča (AddRoundKey).

Pri dešifrovaní sa používajú inverzné transformácie, okrem tranformácie AddRoundKey, ktorá ostáva rovnaká. Poradie transformácií sa pri dešifrovaní mení a má nasledovnú sekvenciu [2]:

1. Inverzný posun riadkov (InvShiftRows).
2. Inverzná substitúcia bytov (InvSubBytes).
3. Pripočítanie iteračného kľúča (AddRoundKey).
4. Inverzné pomiešanie stĺpcov (InvMixColumns).

Podľa [2] transformácia InvShiftRows ovplyvňuje sekvenciu bytov stavovej matice, ale nemení ich sekvenciu a vykonanie tranformácie nie je závislé na obsahu bytov. Transformácia InvSubBytes naopak mení obsah bytov stavovej matice podľa inverznej S-box tabuľky (tab. 3.3), ale nemení ich sekvenciu a transformácia nie je na ich sekvencii závislá. Preto je možné tieto transformácie prehodiť. Platí

$$InvShiftRows\left[InvSubBytes\left(S_i\right)\right] = InvSubBytes\left[InvShiftRows\left(S_i\right)\right].$$

Takisto je možné prehodiť aj transformácie AddRoundKey a InvMixColumns. Obe transformácie ovplyvňujú sekvenciu bytov v stavovej matici, ale nemenia obsah bytov a nie sú na ich obsahu závislé. V prípade iteračného kľúča $w_j$stavu matice S_i platí:

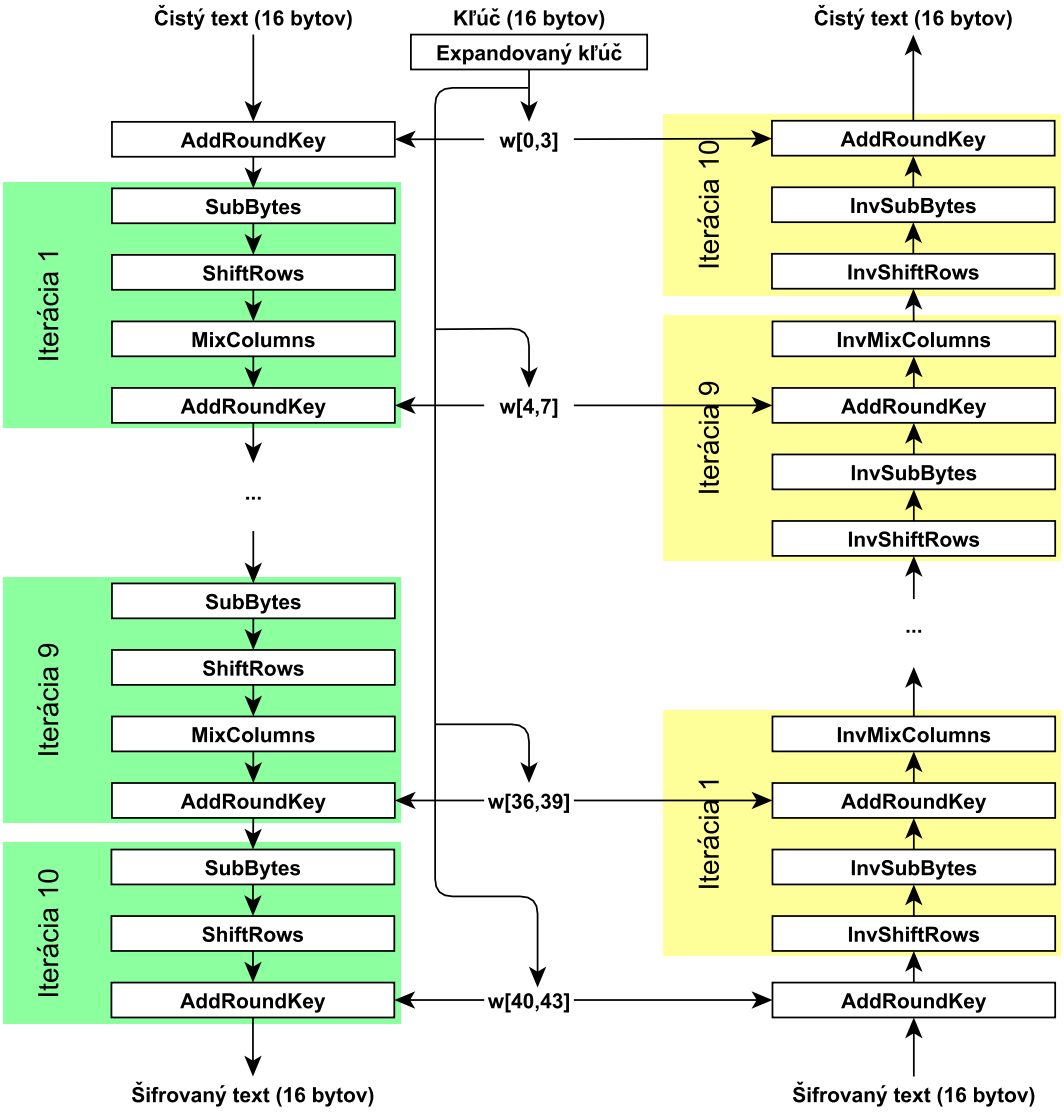
$$InvMixColumns\left(S_i \oplus w_j\right) = \left[InvMixColumns\left(S_i\right)\right] \oplus \left[InvMixColumns\left(w_j\right)\right].$$

Hodnota $(S_i \oplus w_j)$ je výstupom transformácie AddRoundKey, ktorá je vstupom transformácie InvMixColumns. Ako ukazuje rovnica, poradie týchto dvoch transformácií je možné prehodiť v prípade, že sa ako prvá nad iteračným kľúčom w_j

Tab. 3.3: Inverzná tabuľka S-box [2]

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
1	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
2	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
3	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
4	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
5	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
6	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
7	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
8	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
9	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

aplikuje transformácia InvMixColumns. Vyššie uvedeným postupom dostávame inverznú AES šifru. Šifrovací a dešifrovací proces s uvedenými transformáciami je znázornený na obr. 3.7.



Obr. 3.7: Šifrovací a dešifrovací proces AES [2]

4 INŠTRUKČNÉ SADY PRE PODPORU KRYPTOGRAFIE

4.1 Inštrukčná sada pre podporu štandardu pokročilého šifrovania (AES-NI)

Firma Intel predstavila inštrukčnú sadu AES-NI (Advanced Encryption Standard - New Instructions) s príchodom rodiny procesorov Intel Core, ktoré sú založené na 32 nm mikroarchitektúre s kódovým označením Westmere. Táto architektúra má 6 inštrukcií pre podporu šifrovania pomocou blokovej šifry AES:

- 4 inštrukcie, ktoré zvyšujú výkon AES šifrovania a dešifrovania (AESENC, AESENCLAST, AESDEC, AESDECLAST).
- 2 inštrukcie, ktoré slúžia ako podpora pri expanzii kľúča (AESKEYGENASSIST, AESIMC).

Spoločne tieto inštrukcie poskytujú plnú hardvérovú podporu AES šifrovania a dešifrovania pre všetky veľkosti kľúča (128, 192 a 256 bitov) a štandardnej veľkosti blokovej šifry 128 bitov. Sú vhodné pre všeobecné použitie šifry AES, vrátane ich využitia pri šifrovaní/dešifrovaní v rôznych operačných módoch. Tieto inštrukcie sú tiež vhodné pri použití autentizovaného šifrovania, akým je mód GCM [5].

Tabuľka C.1 poskytuje zápis inštrukcie spolu s vysvetlením vykonanej operácie. Formát zápisu inštrukcie s operandami je [1]:

```
INŠTRUKCIA CIEĽ,ZDROJ,MASKA
```

4.2 Inštrukčná sada pre podporu násobenia bez prenosu (CLMUL)

Podobne ako pri inštrukciách AES-NI sa s príchodom mikroarchitektúry Intel Westmere objavila nová inštrukcia PCLMULQDQ, ktorá vynásobí dva 64 bitové operandy bez prenosu. Násobenie bez prenosu je relatívne časovo náročná operácia, ktorá sa používa v niekoľkých kryptografických systémoch a štandardoch. Preto urýchľenie tejto operácie prináša výrazné zvýšenie rýchlosti výpočtu. Násobenie bez prenosu je kľúčové pri implementácii operačného módu blokovej šifry Galois Counter Mode (GCM). GCM používa násobenie bez prenosu dvoch 128 bitových operandov. Výsledkom je 255 bitová hodnota. Táto operácia sa používa v prvej fáze výpočtu autentizačného tagu v GCM móde. PCLQMULQDQ inštrukcia vypočíta

[1] Podrobné informácie o zápise rôznych inštrukcií a ich operandoch poskytuje [1].

Tab. 4.1: Prehľad AES-NI inštrukcií

Inštrukcia	Popis
AESENC xmm1,xmm2/m128	Vykoná jednu iteráciu (round) šifrovania 128 bitov dát v xmm1 so 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESENCLAST xmm1,xmm2/m128	Vykoná poslednú iteráciu (round) šifrovania 128 bitov dát v xmm1 so 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESDEC xmm1,xmm2/m128	Vykoná jednu iteráciu (round) dešifrovania 128 bitov dát v xmm1 so 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESDECLAST xmm1,xmm2/m128	Vykoná poslednú iteráciu (round) dešifrovania 128 bitov dát v xmm1 so 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESIMC xmm1,xmm2/m128	Vykoná transformáciu InvMixColumn nad 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESKEYGENASSIST xmm1, xmm2/m128,imm8	Asistuje pri generovaní AES iteračného kľúča (round key) s využitím 8 bitovej iteračnej konštanty (round constant) špecifikovanej v maske imm8. Inštrukcia pracuje nad dátami v xmm2/m128 a výsledok uloží do xmm1.

127 bitový výsledok z dvoch 64 bitových operandov. Preto ju je možné využiť pri výpočte výsledku potrebného pri GCM móde [6].

Tabuľka C.2 poskytuje zápis inštrukcie spolu s jej popisom. Formát zápisu inštrukcie PCLMULQDQ je:

```
INŠTRUKCIA CIEĽ,ZDROJ,MASKA
```

4.3 Inštrukčná sada pre podporu rozšírenej hešovacej funkcie (SHA Extensions)

Secure Hash Algorithm (SHA) je kryptografický hešovací algoritmus, ktorý sa v súčasnosti používa v mnohých všeobecných kryptografických aplikáciách. Primárne

Tab. 4.2: Zápis a popis inštrukcie násobenia bez prenosu

Inštrukcia	**Popis**
PCLMULQDQ xmm1, xmm2/m128,imm8	Inštrukcia vynásobí jeden dátový typ quadword (8 bytov) v xmm1 s jedným quadwordom v xmm2/m128 bez prenosu. Výsledkom je dátový typ double quadword (16 bytov) uložený v xmm1. Operand imm8 je maska, ktorá definuje, ktorý quadword xmm1 a xmm2/m128 sa má použiť.

použitie SHA zahŕňa:

- integritu,
- autentizáciu správ,
- digitálne podpisy.

Inštrukčná sada SHA Extensions bola navrhnutá tak, aby poskytovala podporu pre dva algoritmy: SHA-1 a SHA-256 [2].

Inštrukčná sada SHA Extensions je súbor siedmich SIMD inštrukcií, ktoré spolu poskytujú zvýšenie výkonu algoritmov SHA-1 a SHA-256 v IA (Intel Architecture) procesoroch. Pre algoritmus SHA-1 sú to štyri inštrukcie (SHA1RNDS4, SHA1-NEXTE, SHA1MSG1, SHA1MSG2) a pre SHA-256 tri inštrukcie (SHA256RNDS2, SHA256MSG1, SHA256MSG2) [7]. Tabuľka 4.3 poskytuje prehľad týchto inštrukcií spolu s vysvetlením vykonanej operácie. Formát zápisu je:

INŠTRUKCIA CIEĽ,ZDROJ,MASKA

[2] Algoritmus SHA-224 je implicitne podporovaný inštrukciami pre SHA-256.

Tab. 4.3: Prehľad SHA Extensions inštrukcií

Inštrukcia	**Popis**
SHA1RNDS4 xmm1, xmm2/m128,imm8	Inštrukcia vykoná štyri SHA-1 operácie nad stavmi (A,B,C,D) s predvypočítaným súčtom nasledujúcich štyroch dvojslov SHA-1 správ a stavovou premennou E z xmm2/m128. Operand imm8 je kontrolná logická funkcia a konštanta cyklu.
SHA1NEXTE xmm1, xmm2/m128	Inštrukcia počíta SHA-1 stavovú premennú E po štyroch operáciách nad aktuálnou stavovou premennou A v xmm1. Hodnota E je pripočítaná k preddefinovanému dvojslovu v xmm2/m128 a uložená do preddefinovaného dvojslova xmm1.
SHA1MSG1 xmm1, xmm2/m128	Inštrukcia počíta medzivýsledok nasledujúceho dvojslova správy SHA-1 s využitím predchádzajúceho dvojslova správy z xmm1 a xmm2/m128. Výsledok uloží do xmm1.
SHA1MSG2 xmm1, xmm2/m128	Inštrukcia vykoná záverečný výpočet štyroch ďalších dvojslov SHA-1 správy s využitím medzivýsledku v xmm1 a predchádzajúceho dvojslova správy v xmm2/m128. Výsledok uloží do xmm1.
SHA256RNDS2 xmm1,xmm2/m128, <xmm0>	Inštrukcia vykoná 2 cykly SHA-256 operácie s využitím počiatočného stavu SHA-256 (C,D,G,H) z xmm1, počiatočného stavu SHA-256 (A,B,E,F) z xmm2/m128, predvypočítaného súčtu nasledujúcich dvoch dvojslov správy a príslušnej konštanty z implicitného operandu xmm0. Aktualizovaný SHA-256 stav (A,C,E,F) uloží do xmm1.
SHA256MSG1 xmm1,xmm2/m128	Inštrukcia vykoná medzivýpočet pre nasledujúce štyri dvojslová SHA-256 správy s využitím predchádzajúcich dvojslov správy z xmm1 a xmm2/m128. Výsledok uloží do xmm1.
SHA256MSG2 xmm1,xmm2/m128	Inštrukcia vykoná záverečný výpočet štyroch ďalších dvojslov SHA-256 správy s využitím predchádzajúcich dvojslov správy z xmm1 a xmm2/m128. Výsledok uloží do xmm1.

5 PROGRAMOVANIE NA PLATFORME WINDOWS X64

Programovanie pod 64 bitovým WinAPI je v niektorých aspektoch výrazne odlišné od programovania pod 32 bitovou verziou. Táto kapitola je zameraná práve na výhody, nevýhody a rozdiely programovania v 32 a 64 bitovom prostredí.

Jednou z výhod 64 bitového systému je podpora omnoho väčšieho množstva fyzickej pamäte. Väčšina 32 bitových operačných systémov Windows podporuje maximálne 4 GB fyzickej pamäte spolu s 3 GB adresného priestoru pre každý proces, zatiaľ čo 64 bitové systémy podporujú až do 2 TB fyzickej pamäte spolu s 8 TB adresného priestoru pre každý proces. Zvýšenie objemu fyzickej pamäte zahŕňa nasledujúce výhody pre aplikácie [8]:

- Každá aplikácia dokáže podporovať viac používateľov. Každá aplikácia (alebo jej časť) musí byť pre každého užívateľa replikovaná, čo vyžaduje dodatočnú pamäť.
- Každá aplikácia má vyššiu výkonnosť. Zvýšením fyzickej pamäte sa dosiahlo to, že viac aplikácií môže bežať súčasne a všetky tieto aplikácie zostávajú v hlavnej pamäti systému rezidentné. Tým sa redukuje, či dokonca eliminuje, výkonnostná penalizácia pri prístupe z a do disku.
- Každá aplikácia má väčšiu pamäť pre manipuláciu a ukladanie dát. Databázy dokážu uložiť viac dát do fyzickej pamäte systému. Prístup k dátam je rýchlejší, pretože čítanie z disku nie je potrebné.
- Aplikácie môžu manipulovať s vyšším objemom dát jednoduchšie a spolahlivejšie. Z tohto dôvodu kompozícia a práca s videom vyžaduje 64 bitový systém. Modelovanie vedeckých a finančných aplikácií výrazne benefituje z pamäťovo-rezidentných dátových štruktúr, ktoré v 32 bitovej verzii nie sú.

Z hľadiska výpočtových jednotiek je asi najvýraznejšou zmenou počet ako všeobecných, tak aj dátových XMM/YMM registrov. Oproti programovaniu v 32 bitovom režime je v 64 bitovom režime dostupný dvojnásobný počet registrov:

- Všeobecné registre majú v 64 režime prefix R a označujú sa nasledovne: RAX, RCX, RDX, RDI, RSI, RBX, RBP, RSP a R8 až R15. Okrem ich vyššieho počtu sa zväčšila aj ich dĺžka z 32 bitov na 64 bitov.
- Dátové registre výpočtovej jednotky SSE sú v 64 bitovom režime bez zmeny, čo sa názvov týka. Ich celkový počet je 16 a označujú sa XMM0 až XMM15. Dĺžka jedného registru je 128 bitov.
- Pre dátové registre výpočtových jednotiek AVX a FMA platí to isté, čo pre výpočtovú jednotku SSE. Názov ostal bez zmeny a ich počet sa zvýšil na 16. Označujú sa ako registre YMM0 až YMM15. Dĺžka jedného registru je 256 bitov.

5.1 Konvencia postupnosti volania _fastcall a predávanie parametrov

Ďalšou výraznou zmenou programovania pod 64 bitovým režimom je zjednotenie konvencí postupnosti volania (angl. calling convention) pod jedinú konvenciu s názvom _fastcall, zatiaľ čo v 32 bitovom režime existovalo niekoľko rôznych konvencií. Táto zmena je výhodou, keďže nie je nutné zisťovať, pod akou konvenciou volania bol daný zdrojový kód napísaný.

Táto konvencia volania využíva pod Windows ABI (Application Binary Interface) k predávanie parametrov ako reigstre, tak zásobník. Prvé štyri parametre sú volanej funkcii predané registrami, zvyšné parametre pomocou zásobníku. Akýkoľvek argument, ktorý nemá veľkosť 1, 2, 4 alebo 8 byteov musí byť predaný cez zásobník. Neexistuje žiadna možnosť, ako predať jeden argument pomocou viacerých registrov. Registrový zásobník x87 je nevyužitý. Prvé štyri celočíselné argumenty sú predané pomocou registrov RCX, RDX, R8 a R9 (presne v tomto poradí). Prvé štyri argumenty s plávajúcou desatinnou čiarkou sú predávané pomocou registrov XMM0, XMM1, XMM2 a XMM3 (presne v tomto poradí). Ako návratový register sa využíva register RAX. Za vyhradenie miesta pre parametre je zodpovedný volajúci, ktorý tiež zodpovedá za vyhradenie miesta prvých štyroch parametrov predávaných registrami, a to aj v prípade, že sa volanej funkcii predávajú menej ako 4 parametre [9].

5.2 Využitie registrov

Konvencia volania _fastcall definuje tzv. volatilné a nevolatiné registre. Niektoré hodnoty registrov môžu byť vo volanej funkcii prepísané, resp. zničené, ale pre niektoré registre platí, že ich obsah pri vstupe do funkcie musí byť zachovaný, prípadne obnovený pred návratom z volanej funkcie. Tab. 5.1 zobrazuje stav a využitie registrov [10].

Tab. 5.1: Využitie registrov v 64 bitovom režime

Register	**Stav**	**Použitie**
RAX	Volatilný	Register návratovej hodnoty
RCX	Volatilný	Prvý celočíselný argument
RDX	Volatilný	Druhý celočíselný argument
R8	Volatilný	Tretí celočíselný argument
R9	Volatilný	Štvrtý celočíselný argument
R10:R11	Volatilný	Musí byť zachovaný volajúcim, využitie pri syscall/sysret inštrukciách
R12:R15	Nevolatilný	Musí byť zachovaný volaným
RDI	Nevolatilný	Musí byť zachovaný volaným
RSI	Nevolatilný	Musí byť zachovaný volaným
RBX	Nevolatilný	Musí byť zachovaný volaným
RBP	Nevolatilný	Musí byť zachovaný volaným, používa sa ako rámcový ukazateľ
RSP	Nevolatilný	Ukazateľ na zásobník
XMM0, YMM0	Volatilný	Prvý argument v plávajúcej rádovej čiarke
XMM1, YMM1	Volatilný	Druhý argument v plávajúcej rádovej čiarke
XMM2, YMM2	Volatilný	Tretí argument v plávajúcej rádovej čiarke
XMM3, YMM3	Volatilný	Štvrtý argument v plávajúcej rádovej čiarke
XMM4, YMM4	Volatilný	Musí byť zachovaný volajúcim
XMM5, YMM5	Volatilný	Musí byť zachovaný volajúcim
XMM6:XMM15, YMM6:YMM15	Nevolatilný (XMM), Volatilný (horná polovica YMM)	Musí byť zachovaný volaným, YMM musia byť zachované volajúcim

6 PRAKTICKÁ REALIZÁCIA VÝPOČTOV

Táto kapitola poskytuje informácie o realizácií ako algebraických, tak kryptografických operácií testovaných v práci. V nasledujúcich podkapitolách sú vysvetlené postupy riešenia jednotlivých problematík, spolu s popisom a ukážkami vytvorených zdrojových kódov, ktoré boli následne použité pri testovaní, a teda aj samotné inštrukcie pre realizáciu výpočtov. Tieto inštrukcie je možné nájsť v samostatných prílohách k práci.

6.1 Algebraická časť

Testy algrebraickej časti v 32 bitovom režime v [1] boli v tejto práci doplnené o testy v 64 bitovom režime, ktorých je spolu 12. Jedná sa o základné výpočty s maticami a vektormi, konkrétne násobenie matice vektorom, resp. vektora maticou. Zdrojové kódy pracujú s číslami ako v jednoduchej presnosti, tak v dvojitej presnosti.

6.2 Zdrojové kódy

Všetky zdrojové kódy sú realizované pomocou cyklov tak, aby bolo možné pracovať s rôznymi rozmermi matice a príslušného vektoru. Cykly sú použité dva, jeden vnorený do druhého a tieto zaručujú korektný pohyb ukazateľa po matici. V každom kroku výpočet pracuje s rozmermi submatice, ktoré sú závislé na použitej výpočtovej jednotke a presnosti:

- 4x4 - výpočtová jednotka SSE, jednoduchá presnosť,
- 8x8 - výpočtové jednotky AVX a FMA, jednoduchá presnosť,
- 2x2 - výpočtová jednotka SSE, dvojitá presnosť,
- 4x4 - výpočtové jednotky AVX a FMA, dvojitá presnosť.

Samotný výpočet vo vnútri vnoreného cyklu pre danú výpočtovú jednotku a presnosť je popísaný v rámci tejto kapitoly.

6.2.1 Transpozícia vektora

Zdrojový kód transpozície vektora sa opiera o kapitolu 1.1. Pre jednotku SSE bola transpozícia realizovaná odlišným spôsobom, ako u jednotiek AVX a FMA. Pri násobení sprava nebola nutná transpozícia, pretože samotný presun hodnôt do registra XMM, resp. YMM zabezpečil jeho okamžitú transpozíciu.

Výpočtová jednotka SSE

Nižšie sú uvedené a vysvetlené dva odlišné postupy zdrojových kódov transpozície vektora pre SSE. V jednej verzii boli použité inštrukcie pomiešania (shuffle), v druhej inštrukcie rozbaliena (unpack). Nasledujú zdrojové kódy transpozície vektora v jednoduchej presnosti:

Verzia Shuffle:

```
;vstupný vektor v XMM0
movups xmm1,xmm0      ; XMM1 = v4,v3,v2,v1
shufps xmm1,xmm1,0x00      ; XMM1 = v1,v1,v1,v1
movups xmm2,xmm0      ; XMM2 = v4,v3,v2,v1
shufps xmm2,xmm2,0x55      ; XMM2 = v2,v2,v2,v2
movups xmm3,xmm0      ; XMM3 = v4,v3,v2,v1
shufps xmm3,xmm3,0xAA      ; XMM3 = v3,v3,v3,v3
movups xmm4,xmm0      ; XMM4 = v4,v3,v2,v1
shufps xmm4,xmm4,0xFF      ; XMM4 = v4,v4,v4,v4
;transponovaný vektor v XMM1 až XMM4
```

Verzia Unpack:

```
;vstupný vektor v xmm0
movups xmm1,xmm0      ; XMM1 = v4,v3,v2,v1
unpcklps xmm1,xmm1      ; XMM1 = v2,v2,v1,v1
movups xmm2,xmm1      ; XMM2 = v2,v2,v1,v1
unpcklps xmm1,xmm1      ; XMM1 = v1,v1,v1,v1
unpckhps xmm2,xmm2      ; XMM2 = v2,v2,v2,v2
movups xmm3,xmm0      ; XMM3 = v4,v3,v2,v1
unpckhps xmm3,xmm3      ; XMM3 = v4,v4,v3,v3
movups xmm4,xmm3      ; XMM4 = v4,v4,v3,v3
unpcklps xmm3,xmm3      ; XMM3 = v3,v3,v3,v3
unpckhps xmm4,xmm4      ; XMM4 = v4,v4,v4,v4
;transponovaný vektor v XMM1 až XMM4
```

Nasledujú zdrojové kódy transpozície vektora v dvojitej prensosti:

Verzia Shuffle:

```
;vstupný vektor v XMM0
```

```
movupd xmm1,xmm0      ; XMM1 = v2,v1
movupd xmm2,xmm0      ; XMM2 = v2,v1
shufpd xmm1,xmm1,0x00      ; XMM1 = v1,v1
shufpd xmm2,xmm2,0xFF      ; XMM2 = v2,v2
;transponovaný vektor v XMM1 a XMM2
```

Verzia Unpack:

```
;vstupný vektor v xmm0
movupd xmm1,xmm0      ; XMM1 = v2,v1
movupd xmm2,xmm0      ; XMM2 = v2,v1
unpcklpd xmm1,xmm1      ; XMM1 = v1,v1
unpckhpd xmm2,xmm2      ; XMM2 = v2,v2
;transponovaný vektor v XMM1 a XMM2
```

Výpočtové jednotky AVX a FMA

Nové inštrukčné sady pre jednotky AVX a FMA poskytujú inštrukcie, ktoré, z hľadiska kódu, prinašajú skrátenie. Konkrétne sú to inštrukcie **broadcast**, ktoré jednu vybranú hodnotu roztiahnu do celého registra XMM, resp. YMM. Nasleduje zdrojový kód transpozície vektora v jednoduchej presnosti:

```
;ukazateľ na vstupný vektor v RSI
vbroadcastss ymm0,[esi]      ; YMM0 = v1,v1,v1,v1,v1,v1,v1,v1
add esi,4      ; posun ukazateľa na ďalší prvok
vbroadcastss ymm1,[esi]      ; YMM1 = v2,v2,v2,v2,v2,v2,v2,v2
add esi,4      ; posun ukazateľa na ďalší prvok
vbroadcastss ymm2,[esi]      ; YMM2 = v3,v3,v3,v3,v3,v3,v3,v3
add esi,4      ; posun ukazateľa na ďalší prvok
vbroadcastss ymm3,[esi]      ; YMM3 = v4,v4,v4,v4,v4,v4,v4,v4
add esi,4      ; posun ukazateľa na ďalší prvok
vbroadcastss ymm4,[esi]      ; YMM4 = v5,v5,v5,v5,v5,v5,v5,v5
add esi,4      ; posun ukazateľa na ďalší prvok
vbroadcastss ymm5,[esi]      ; YMM5 = v6,v6,v6,v6,v6,v6,v6,v6
add esi,4      ; posun ukazateľa na ďalší prvok
vbroadcastss ymm6,[esi]      ; YMM6 = v7,v7,v7,v7,v7,v7,v7,v7
add esi,4      ; posun ukazateľa na ďalší prvok
```

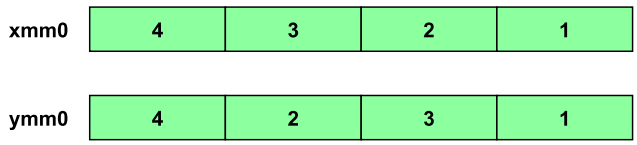
```
vbroadcastss ymm7,[esi]        ; YMM7 = v8,v8,v8,v8,v8,v8,v8,v8
;transponovaný vektor v YMM0 až YMM7
```

Nasleduje zdrojový kód transpozície vektora v dvojitej presnosti:

```
;ukazateľ na vstupný vektor v RSI
vbroadcastsd ymm0,[rsi]        ; YMM0 = v1,v1,v1,v1
add rsi,8       ; posun ukazateľa na ďalší prvok
vbroadcastsd ymm1,[rsi]        ; YMM1 = v2,v2,v2,v2
add rsi,8       ; posun ukazateľa na ďalší prvok
vbroadcastsd ymm2,[rsi]        ; YMM2 = v3,v3,v3,v3
add rsi,8       ; posun ukazateľa na ďalší prvok
vbroadcastsd ymm3,[rsi]        ; YMM3 = v4,v4,v4,v4
;transponovaný vektor v YMM0 až YMM3
```

6.2.2 Potrebné permutačné operácie

Pri práci s YMM registrami sa väčšina inštrukcií správa odlišne, ako je to pri XMM registroch. Výsledné hodnoty po základných matematických operáciách sú uložené v prehádzanom poradí. Napríklad pri použití rovnakej operácie, ako je horizontálne sčítanie, sa pri výpočtovej jednotke SSE uložia hodnoty do XMM registra v očakávanom poradí, zatiaľ čo pri jednotkách AVX a FMA sú niektoré hodnoty prehodené, čo ilustruje obrázok 6.1. Preto bolo nevyhnutné využit pri výpočtových jednotkách AVX a FMA permutačné inštrukcie, ktoré usporiadali hodnoty v YMM registri do požadovaného poradia.



Obr. 6.1: Rozdielne uloženie hodnôt v registri XMM a YMM

6.2.3 Násobenie zľava v jednoduchej presnosti

Výpočty násobenia vektora maticou nadväzujú na teoretický základ z kapitoly 1.2.

Výpočtová jednotka SSE

Pre výpočtovú jednotku boli použité dve verzie výpočtu, ktoré sa odlišovali v zmienenej transpozícii vektora.

Verzia Shuffle:

```
movups xmm0,[rsi]       ; vstupný vektor v XMM0
movups xmm4,[rcx]       ; XMM4 = m4,m3,m2,m1
add rcx,r12       ; posun ukazateľa na prvky matice
movups xmm5,[rcx]       ; XMM5 = m8,m7,m6,m5
add rcx,r12       ; posun ukazateľa na prvky matice
movups xmm6,[rcx]       ; XMM6 = m12,m11,m10,m9
add rcx,r12       ; posun ukazateľa na prvky matice
movups xmm7,[rcx]       ; XMM7 = m16,m15,m14,m13
add rcx,r12       ; posun ukazateľa na prvky matice
movups xmm2,[rax]       ; výstupný vektor v XMM2

movups xmm1,xmm0       ; výpočet submatice 4x4
shufps xmm1,xmm1,0x00
mulps xmm1,xmm4       ; 1. riadok matice krát 1. prvok vektora
addps xmm2,xmm1       ; medzivýsledok pripočítaný do XMM2
movups xmm1,xmm0
shufps xmm1,xmm1,0x55
mulps xmm1,xmm5       ; 2. riadok matice krát 2. prvok vektora
addps xmm2,xmm1       ; medzivýsledok pripočítaný do XMM2
movups xmm1,xmm0
shufps xmm1,xmm1,0xAA
mulps xmm1,xmm6       ; 3. riadok matice krát 3. prvok vektora
addps xmm2,xmm1       ; medzivýsledok pripočítaný do XMM2
movups xmm1,xmm0
shufps xmm1,xmm1,0xFF
mulps xmm1,xmm7       ; 4. riadok matice krát 4. prvok vektora
addps xmm2,xmm1       ; medzivýsledok pripočítaný do XMM2
movups [rax],xmm2       ; výsledný subvektor do RAX
```

Verzia Unpack:

```
movups xmm0,[rsi]       ; vstupný vektor v XMM0
movups xmm4,[rcx]       ; XMM4 = m4,m3,m2,m1
add rcx,r12       ; posun ukazateľa na prvky matice
movups xmm5,[rcx]       ; XMM5 = m8,m7,m6,m5
add rcx,r12       ; posun ukazateľa na prvky matice
```

```
movups xmm6,[rcx]       ; XMM6 = m12,m11,m10,m9
add rcx,r12       ; posun ukazateľa na prvky matice
movups xmm7,[rcx]       ; XMM7 = m16,m15,m14,m13
add rcx,r12       ; posun ukazateľa na prvky matice
movups xmm3,[rax]       ; výstupný vektor v XMM3

movups xmm1,xmm0       ; výpočet submatice 4x4
unpcklps xmm1,xmm1
movups xmm2,xmm1
unpcklps xmm1,xmm1
unpckhps xmm2,xmm2
mulps xmm1,xmm4       ; 1. riadok matice krát 1. prvok vektora
addps xmm3,xmm1       ; medzivýsledok pripočítaný do XMM3
mulps xmm2,xmm5       ; 2. riadok matice krát 2. prvok vektora
addps xmm3,xmm2       ; medzivýsledok pripočítaný do XMM3

movups xmm1,xmm0
unpckhps xmm1,xmm0
movups xmm2,xmm1
unpcklps xmm1,xmm1
unpckhps xmm2,xmm2
mulps xmm1,xmm6       ; 3. riadok matice krát 3. prvok vektora
addps xmm3,xmm1       ; medzivýsledok pripočítaný do XMM3
mulps xmm2,xmm7       ; 4. riadok matice krát 4. prvok vektora
addps xmm3,xmm2       ; medzivýsledok pripočítaný do XMM3
movups [rax],xmm3       ; výsledný subvektor do RAX
```

Výpočtové jednotky AVX a FMA

Pre výpočtové jednotku bol použitý jeden výpočet pre každú jednotku, a to vyššie zmienená verzia transpozície vektora pomocou broadcast inštrukcií.

Výpočet pomocou AVX:

```
vmovups ymm4,[rcx]       ; submatica 8x8 do YMM4 až YMM11
add rcx,r12       ; posun ukazateľa na prvky matice
vmovups ymm5,[rcx]
add rcx,r12
vmovups ymm6,[rcx]
```

```
add rcx,r12
vmovups ymm7,[rcx]
add rcx,r12
vmovups ymm8,[rcx]
add rcx,r12
vmovups ymm9,[rcx]
add rcx,r12
vmovups ymm10,[rcx]
add rcx,r12
vmovups ymm11,[rcx]

vbroadcastss ymm0,[esi]      ; 1. prvok vektora v celom YMM0
vmulps ymm4,ymm0,ymm4        ; 1. riadok matice krát 1. prvok vektora
add esi,4      ; posun ukazateľa na vst. vektor o 1 prvok

vbroadcastss ymm0,[esi]      ; 2. prvok vektora v celom YMM0
vmulps ymm5,ymm0,ymm5        ; 2. riadok matice krát 2. prvok vektora
add esi,4      ; posun ukazateľa na vst. vektor o 1 prvok

vbroadcastss ymm0,[esi]      ; 3. prvok vektora v celom YMM0
vmulps ymm6,ymm0,ymm6        ; 3. riadok matice krát 3. prvok vektora
add esi,4      ; posun ukazateľa na vst. vektor o 1 prvok

vbroadcastss ymm0,[esi]
vmulps ymm7,ymm0,ymm7
add esi,4

vbroadcastss ymm0,[esi]
vmulps ymm8,ymm0,ymm8
add esi,4

vbroadcastss ymm0,[esi]
vmulps ymm9,ymm0,ymm9
add esi,4

vbroadcastss ymm0,[esi]
vmulps ymm10,ymm0,ymm10
add esi,4
```

```
vbroadcastss ymm0,[esi]
vmulps ymm11,ymm0,ymm11
sub esi,28      ; posun ukazateľa späť na pôvodnú hodnotu

vaddps ymm1,ymm1,ymm4      ; sčítanie medzivýsledkov po násobení
vaddps ymm1,ymm1,ymm5      ; do výsledného subvektora v YMM1
vaddps ymm1,ymm1,ymm6
vaddps ymm1,ymm1,ymm7
vaddps ymm1,ymm1,ymm8
vaddps ymm1,ymm1,ymm9
vaddps ymm1,ymm1,ymm10
vaddps ymm1,ymm1,ymm11

vmovups [rax],ymm1      ; výsledný subvektor do RAX
```

Výpočet pomocou FMA:

```
vmovups ymm4,[rcx]      ; submatica 8x8 do YMM4 až YMM11
add rcx,r12      ; posun ukazateľa na prvky matice
vmovups ymm5,[rcx]
add rcx,r12
vmovups ymm6,[rcx]
add rcx,r12
vmovups ymm7,[rcx]
add rcx,r12
vmovups ymm8,[rcx]
add rcx,r12
vmovups ymm9,[rcx]
add rcx,r12
vmovups ymm10,[rcx]
add rcx,r12
vmovups ymm11,[rcx]

vbroadcastss ymm0,[esi]      ; 1. prvok vektora v celom YMM0
vfmadd231ps ymm1,ymm0,ymm4      ; 1. riadok matice krát 1. prvok vektora a následné sčítanie výsledku po násobení do YMM1 v jednom kroku
add esi,4      ; posun ukazateľa na vst. vektor o 1 prvok
```

```
vbroadcastss ymm0,[esi]       ; 2. prvok vektora v celom YMM0
vfmadd231ps ymm1,ymm0,ymm5    ; 2. riadok matice krát 2. prvok vektora a následné sčítanie výsledku po násobení do YMM1 v jednom kroku
add esi,4       ; posun ukazateľa na vst. vektor o 1 prvok

vbroadcastss ymm0,[esi]       ; 3. prvok vektora v celom YMM0
vfmadd231ps ymm1,ymm0,ymm6    ; 3. riadok matice krát 3. prvok vektora a následné sčítanie výsledku po násobení do YMM1 v jednom kroku
add esi,4       ; posun ukazateľa na vst. vektor o 1 prvok

vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm7
add esi,4

vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm8
add esi,4

vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm9
add esi,4

vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm10
add esi,4

vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm11
sub esi,28      ; posun ukazateľa späť na pôvodnú hodnotu

vmovups [rax],ymm1      ; výsledný subvektor do RAX
```

6.2.4 Násobenie zľava v dvojitej presnosti

Výpočty v dvojitej presnosti sa od jednoduchej presnosti odlišujú vo veľkosti týchto dátových typov. Zatiaľ čo číslo v jednoduchej presnosti má 32 bitov, v dvojitej presnosti to je 64 bitov. Preto je veľkosť matice a vektoru v jednom kroku výpočtu dvojnásobne menšia. Principiálne sú však výpočty rovnaké.

Výpočtová jednotka SSE

Verzia Shuffle:

```
movupd xmm0,[rsi]       ; vstupný vektor v XMM0
movupd xmm4,[rcx]       ; XMM4 = m4,m3,m2,m1
add rcx,r12       ; posun ukazateľa na prvky matice
movups xmm5,[rcx]       ; XMM5 = m8,m7,m6,m5
movups xmm3,[rax]       ; výstupný vektor v XMM3

movupd xmm1,xmm0       ; výpočet submatice 2x2
movupd xmm2,xmm0       ; XMM1 = XMM2 = XMM0

shufpd xmm1,xmm1,0x00       ; XMM1 = v1,v1
shufpd xmm2,xmm2,0xFF       ; XMM2 = v2,v2

mulpd xmm1,xmm4       ; 1. riadok matice krát 1. prvok vektora
mulpd xmm2,xmm5       ; 2. riadok matice krát 2. prvok vektora

addpd xmm1,xmm2       ; sčítanie medzivýsledku do výstupného
addpd xmm3,xmm1       ; subvektora v XMM3

movupd [rax],xmm3       ; výsledný subvektor do RAX
```

Verzia Shuffle:

```
movupd xmm0,[rsi]       ; vstupný vektor v XMM0
movupd xmm4,[rcx]       ; XMM4 = m4,m3,m2,m1
add rcx,r12       ; posun ukazateľa na prvky matice
movups xmm5,[rcx]       ; XMM5 = m8,m7,m6,m5
movups xmm3,[rax]       ; výstupný vektor v XMM3

movupd xmm1,xmm0       ; výpočet submatice 2x2
movupd xmm2,xmm0       ; XMM1 = XMM2 = XMM0

unpcklpd xmm1,xmm1       ; XMM1 = v1,v1
unpckhpd xmm2,xmm2       ; XMM2 = v2,v2

mulpd xmm1,xmm4       ; 1. riadok matice krát 1. prvok vektora
```

```
mulpd xmm2,xmm5      ; 2. riadok matice krát 2. prvok vektora

addpd xmm1,xmm2      ; sčítanie medzivýsledku do výstupného
addpd xmm3,xmm1      ; subvektora v XMM3

movupd [rax],xmm3    ; výsledný subvektor do RAX
```

Výpočtové jednotky AVX a FMA

Výpočet pomocou AVX:

```
vmovupd ymm4,[rcx]      ; YMM4 = m4,m3,m2,m1
add rcx,r12      ; posun ukazateľa na prvky matice
vmovupd ymm5,[rcx]      ; YMM5 = m8,m7,m6,m5
add rcx,r12      ; posun ukazateľa na prvky matice
vmovupd ymm6,[rcx]      ; YMM6 = m12,m11,m10,m9
add rcx,r12      ; posun ukazateľa na prvky matice
vmovupd ymm7,[rcx]      ; YMM7 = m16,m15,m14,m13
vmovupd ymm3,[rax]      ; výstupný vektor v YMM3

vbroadcastsd ymm1,[rsi]      ; 1. prvok vektora v celom YMM1
add rsi,8      ; posun ukazateľa na vst. vektor o 1 prvok
vmulpd ymm4,ymm4,ymm1      ; 1. riadok matice krát 1. prvok vektora

vbroadcastsd ymm1,[rsi]      ; 2. prvok vektora v celom YMM1
add rsi,8      ; posun ukazateľa na vst. vektor o 1 prvok
vmulpd ymm5,ymm5,ymm1      ; 2. riadok matice krát 2. prvok vektora

vbroadcastsd ymm1,[rsi]      ; 3. prvok vektora v celom YMM1
add rsi,8      ; posun ukazateľa na vst. vektor o 1 prvok
vmulpd ymm6,ymm6,ymm1      ; 3. riadok matice krát 3. prvok vektora

vbroadcastsd ymm1,[rsi]      ; 4. prvok vektora v celom YMM1
sub rsi,24      ; posun ukazateľa späť na pôvodnú hodnotu
vmulpd ymm7,ymm7,ymm1      ; 4. riadok matice krát 4. prvok vektora

vaddpd ymm4,ymm4,ymm5      ; sčítanie medzivýsledkov po násobení
vaddpd ymm6,ymm6,ymm7      ; do výsledného subvektora, ktorý je
vaddpd ymm4,ymm4,ymm6      ; v YMM3
```

```
vaddpd ymm3,ymm3,ymm4
vmovupd [rax],ymm3      ; výsledný subvektor do RAX
```

Výpočet pomocou FMA:

```
vmovupd ymm4,[rcx]      ; YMM4 = m4,m3,m2,m1
add rcx,r12      ; posun ukazateľa na prvky matice
vmovupd ymm5,[rcx]      ; YMM5 = m8,m7,m6,m5
add rcx,r12      ; posun ukazateľa na prvky matice
vmovupd ymm6,[rcx]      ; YMM6 = m12,m11,m10,m9
add rcx,r12      ; posun ukazateľa na prvky matice
vmovupd ymm7,[rcx]      ; YMM7 = m16,m15,m14,m13
vmovupd ymm2,[rax]      ; výstupný vektor v YMM2

vbroadcastsd ymm1,[rsi]      ; 1. prvok vektora v celom YMM1
add rsi,8      ; posun ukazateľa na vst. vektor o 1 prvok
vfmadd231pd ymm2,ymm4,ymm1      ; 1. riadok matice krát 1. prvok vektora a následné sčítanie výsledku po násobení do YMM2 v jednom kroku

vbroadcastsd ymm1,[rsi]      ; 2. prvok vektora v celom YMM1
add rsi,8      ; posun ukazateľa na vst. vektor o 1 prvok
vfmadd231pd ymm2,ymm5,ymm1      ; 2. riadok matice krát 2. prvok vektora a následné sčítanie výsledku po násobení do YMM2 v jednom kroku

vbroadcastsd ymm1,[rsi]      ; 3. prvok vektora v celom YMM1
add rsi,8      ; posun ukazateľa na vst. vektor o 1 prvok
vfmadd231pd ymm2,ymm6,ymm1      ; 3. riadok matice krát 3. prvok vektora a následné sčítanie výsledku po násobení do YMM2 v jednom kroku

vbroadcastsd ymm1,[rsi]      ; 4. prvok vektora v celom YMM1
sub rsi,24      ; posun ukazateľa späť na pôvodnú hodnotu
vfmadd231pd ymm2,ymm7,ymm1      ; 4. riadok matice krát 4. prvok vektora a následné sčítanie výsledku po násobení do YMM2 v jednom kroku

vmovupd [rax],ymm2      ; výsledný subvektor do RAX
```

6.2.5 Násobenie sprava v jednoduchej presnosti

Výpočty násobenia matice vektorom nadväzujú na teoretický základ z kapitoly 1.3. Využité boli výpočtové jednotky SSE a AVX, pretože podmienkou využitia jednotky FMA je, že násobenie a sčítanie musí nasledovať bezprostredne po sebe, čo v tomto type výpočtu nie je možné.

Pri násobení sprava boli u výpočtovej jednotky AVX využité permutačné inštrukcie pre prehádzanie poradia hodnôt v registroch YMM do požadovaného poradia, ako bolo zmienené v kapitole 6.2.2.

Výpočtová jednotka SSE

```
movups xmm0,[r15]      ; transponovaný vstupný vektor v XMM0
movups xmm4,[rcx]      ; XMM4 = m4,m3,m2,m1
add rcx,r12      ; posun ukazateľa na prvky matice
movups xmm5,[rcx]      ; XMM5 = m8,m7,m6,m5
add rcx,r12      ; posun ukazateľa na prvky matice
movups xmm6,[rcx]      ; XMM6 = m12,m11,m10,m9
add rcx,r12      ; posun ukazateľa na prvky matice
movups xmm7,[rcx]      ; XMM7 = m16,m15,m14,m13
add rcx,r12      ; posun ukazateľa na prvky matice
movups xmm2,[r15]      ; výstupný vektor do XMM2

mulps xmm4,xmm0      ; 1. riadok matice krát 1. prvok vektora
mulps xmm5,xmm0      ; 2. riadok matice krát 2. prvok vektora
mulps xmm6,xmm0      ; 3. riadok matice krát 3. prvok vektora
mulps xmm7,xmm0      ; 4. riadok matice krát 4. prvok vektora

haddps xmm4,xmm5      ; horizontálne sčítanie medzivýsledkov
haddps xmm6,xmm7      ; po násobení do výsledného subvektora
haddps xmm4,xmm6

addps xmm2,xmm4      ; výsledný subvektor v XMM2
movups [rax],xmm2      ; výsledný subvektor do RAX
```

Výpočtová jednotka AVX

```
vmovups ymm0,[r15]      ; transponovaný vstupný vektor v YMM0
vmovups ymm4,[rcx]      ; nahratie submatice 8x8 do registrov YMM4-11
```

```
add rcx,r12       ; posun ukazateľa na prvky matice
vmovups ymm5,[rcx]
add rcx,r12
vmovups ymm6,[rcx]
add rcx,r12
vmovups ymm7,[rcx]
add rcx,r12
vmovups ymm8,[rcx]
add rcx,r12
vmovups ymm9,[rcx]
add rcx,r12
vmovups ymm10,[rcx]
add rcx,r12
vmovups ymm11,[rcx]
vmovups ymm2,[r15]       ; výstupný vektor do YMM2

vmulps ymm4,ymm4,ymm0       ; 1. riadok matice krát 1. prvok vektora
vmulps ymm5,ymm5,ymm0       ; 2. riadok matice krát 2. prvok vektora
vmulps ymm6,ymm6,ymm0       ; 3. riadok matice krát 3. prvok vektora
vmulps ymm7,ymm7,ymm0       ; ...
vmulps ymm8,ymm8,ymm0
vmulps ymm9,ymm9,ymm0
vmulps ymm10,ymm10,ymm0
vmulps ymm11,ymm11,ymm0

vmovdqa ymm1,[permute_mask]       ; permutačná maska do YMM1

vhaddps ymm4,ymm4,ymm5       ; horizontálne sčítanie medzivýsledkov
vpermps ymm4,ymm1,ymm4       ; prehodenie hodnôt do správneho poradia
vhaddps ymm6,ymm6,ymm7       ; horizontálne sčítanie medzivýsledkov
vpermps ymm6,ymm1,ymm6       ; prehodenie hodnôt do správneho poradia
vhaddps ymm8,ymm8,ymm9       ; horizontálne sčítanie medzivýsledkov
vpermps ymm8,ymm1,ymm8       ; prehodenie hodnôt do správneho poradia
vhaddps ymm10,ymm10,ymm11
vpermps ymm10,ymm1,ymm10
```

```
vhaddps ymm4,ymm4,ymm6
vpermps ymm4,ymm1,ymm4
vhaddps ymm8,ymm8,ymm10
vpermps ymm8,ymm1,ymm8
vhaddps ymm4,ymm4,ymm8
vpermps ymm4,ymm1,ymm4

vaddps ymm2,ymm2,ymm4       ; výsledný subvektor v YMM2
vmovups [rax],ymm2          ; výsledný subvektor do RAX
```

6.2.6 Násobenie sprava v dvojitej presnosti

Výpočty v dvojitej presnosti sú zhodné s výpočtami v jednoduchej presnosti, ale pracujú s väčším dátovým typom, a teda aj s dvojnásobne menšou submaticou v jednom kroku.

Výpočtová jednotka SSE

```
movupd xmm0,[r15]       ; transponovaný vstupný vektor v XMM0
movupd xmm4,[rcx]       ; XMM4 = m4,m3,m2,m1
add rcx,r12         ; posun ukazateľa na prvky matice
movupd xmm5,[rcx]       ; XMM5 = m8,m7,m6,m5
movupd xmm3,[rax]       ; výstupný vektor do XMM3

mulpd xmm4,xmm0         ; 1. riadok matice krát 1. prvok vektora
mulpd xmm5,xmm0         ; 2. riadok matice krát 2. prvok vektora

haddpd xmm4,xmm5        ; horizontálne sčítanie medzivýsledkov

addpd xmm3,xmm4         ; výsledný subvektor v XMM3
movupd [rax],xmm3       ; výsledný subvektor do RAX
```

Výpočtová jednotka AVX

```
vmovupd ymm0,[r15]      ; transponovaný vstupný vektor v YMM0
vmovupd ymm4,[rcx]      ; YMM4 = m4,m3,m2,m1
add rcx,r12         ; posun ukazateľa na prvky matice
vmovupd ymm5,[rcx]      ; YMM5 = m8,m7,m6,m5
add rcx,r12         ; posun ukazateľa na prvky matice
```

```
vmovupd ymm6,[rcx]      ; YMM6 = m12,m11,m10,m9
add rcx,r12      ; posun ukazateľa na prvky matice
vmovupd ymm7,[rcx]      ; YMM7 = m16,m15,m14,m13
add rcx,r12      ; posun ukazateľa na prvky matice
vmovupd ymm3,[rax]      ; výstupný vektor do YMM3

vmulpd ymm4,ymm4,ymm0      ; 1. riadok matice krát 1. prvok vektora
vmulpd ymm5,ymm5,ymm0      ; 2. riadok matice krát 2. prvok vektora
vmulpd ymm6,ymm6,ymm0      ; 3. riadok matice krát 3. prvok vektora
vmulpd ymm7,ymm7,ymm0      ; 4. riadok matice krát 4. prvok vektora

vhaddpd ymm4,ymm4,ymm5      ; horizontálne sčítanie medzivýsledkov
vpermpd ymm4,ymm4,0xD8      ; prehodenie hodnôt do správneho poradia
vhaddpd ymm6,ymm6,ymm7      ; horizontálne sčítanie medzivýsledkov
vpermpd ymm6,ymm6,0xD8      ; prehodenie hodnôt do správneho poradia
vhaddpd ymm4,ymm4,ymm6      ; horizontálne sčítanie medzivýsledkov
vpermpd ymm4,ymm4,0xD8      ; prehodenie hodnôt do správneho poradia

vaddpd ymm3,ymm3,ymm4      ; výsledný subvektor v YMM3
vmovupd [rax],ymm3      ; výsledný subvektor do RAX
```

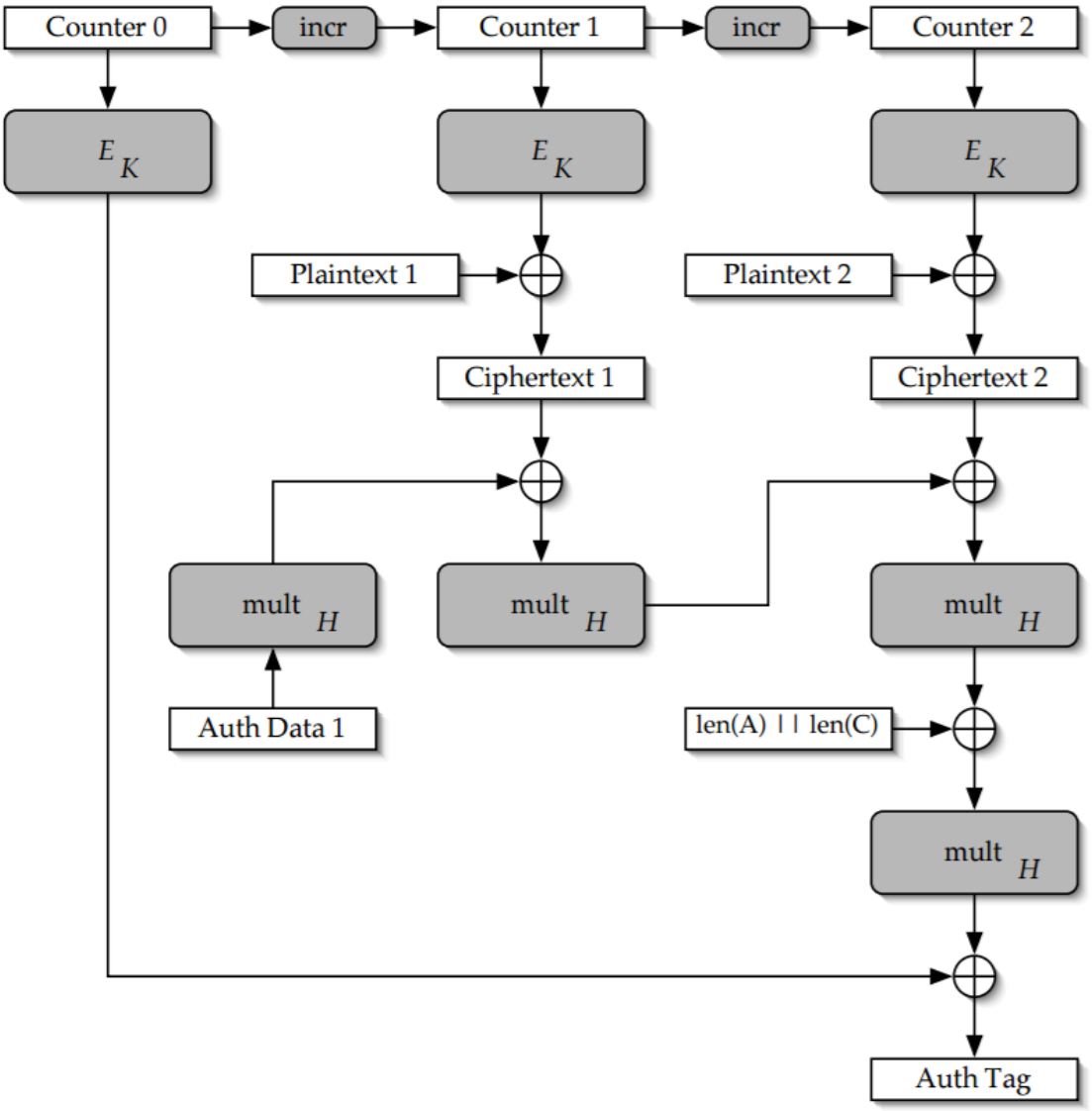
6.3 Kryptografická časť

Výsledkom kryptografickej časti mal byť zdrojový kód implementácie blokovej šifry AES v operačnom móde GCM. Pri riešení zadania bol celkový výpočet rozdelený na 3 časti:

1. šifrovanie,
2. dešifrovanie,
3. autentizácia.

Aby bolo možné overiť správnom výpočtu, boli použité testovacie vektory definované podľa NIST Special Publication 800-38A. Celkový návrh výpočtu sa opieral o blokovú schému, ktorá je na obr. 6.2. V hornej časti schémy je šifrovanie v operačnom móde GCTR, ktorý je prvou časťou algoritmu GCM. Výstup po šifrovaní, teda šifrovaný text, je využitý v druhej časti módu GCM, ktorou je výpočet autentizačného tagu. Nasleduje vysvetlenie jednotlivých blokov schémy:

- Bloky ***Counter*** predstavujú čítač.
- Bloky ***incr*** predstavujú funkciu inkrementácie o 1.



Obr. 6.2: Bloková schéma operačného módu GCM

- Bloky $\boldsymbol{E_K}$ predstavujú proces šifrovania s kľúčom.
- Bloky ***Plaintext*** predstavujú otvorený text.
- Bloky ***Ciphertext*** predstavujú šifrovaný text.
- Bloky $\boldsymbol{mult_H}$ predstavujú násobenie v Galoisovom poli $GF(2^{128})$ s hashovacím kľúčom H. H je 128 bitov núl zašifrovaných blokovou šifrou.
- Blok ***Auth Data*** predstavuje dodatočné autentizačné dáta, čo môže byť napr. hlavička IP protokolu.
- Blok $\boldsymbol{len(A) \,||\, len(C)}$ predstavuje funkciu zlúčenia dvoch 64 bitových stringov do jedného 128 bitového stringu. $\boldsymbol{A}$ je dĺžka dodatočných autentizačných dát, $\boldsymbol{C}$ je dĺžka šifrovaného textu.
- Blok ***Auth Tag*** je výstupný tag použitý pri autentizácií.

6.4 Zdrojové kódy

Všetky vytvorené zdrojové kódy kryptografickej časti sú detailne popísané nižšie. Je nutné uviesť, že funkcie boli vytvorené pre výpočet AES-128, tj. veľkosť kľúča je 128 bitov.

6.4.1 Používané makrá

Keďže mnoho výpočtov sa v jednotlivých funkciách opakuje, boli vytvorené makrá pre zjednodušenie a väčšiu prehľadnosť zdrojových kódov. Nižšie sú uvedené zdrojové kódy použitých makier.

Makro expanzie kľúča

Detailný popis expanzie kľúča poskytuje kapitola 3.2. Nasleduje zdrojový kód makra pre expanziu 128 bitového kľúča.

```
%macro m_key_expansion 1        ; počet vstupných par. makra je 1
AESKEYGENASSIST xmm2,xmm1,%1
pshufd xmm2,xmm2,0xff           ; pomiešanie hodnôt v XMM2 na základe masky
movdqu xmm3,xmm1                ; XMM3 = XMM1
pslldq xmm3,0x4                 ; bytový posun vľavo na základe masky
pxor xmm1,xmm3                  ; XMM3 XOR XMM1
movdqu xmm3,xmm1                ; XMM3 = XMM1
pslldq xmm3,0x4                 ; bytový posun vľavo na základe masky
pxor xmm1,xmm3                  ; XMM3 XOR XMM1
movdqu xmm3,xmm1                ; XMM3 = XMM1
pslldq xmm3,0x4                 ; bytový posun vľavo na základe masky
pxor xmm1,xmm3                  ; XMM3 XOR XMM1
pxor xmm1,xmm2                  ; XMM2 XOR XMM1, XMM1 = výsledok
%endmacro                       ; koniec makra
```

Makro násobenia v Galoisovom poli

Výpočet potrebnej hodnoty je možné rozdeliť na dve časti:

1. Násobenie 128 bitových operandov, kde výsledkom je 255 bitová hodnota. Toto je prvý krok pri výpočte GHASH. Práve v tomto kroku je možné využiť inštrukciu PCLMULQDQ (kap. 4.2), ktorá dva 64 bitové operandy a výsledkom je 127 bitový produkt, ktorý je polovicou potrebnej 255 bitovej hodnoty.

Správne navrhnutý algoritmus dokáže potrebnú 255 bitovú hodnotu korektne vypočítať.

2. Ďalší krok je redukcia modulo polynómu $x^{128} + x^7 + x^2 + x + 1$. Pre úrychlenie tejto operácie sa používajú inštrukcie logického posunu vľavo, resp. vpravo (PSLLD, PSRLD) a inštrukcia pomiešania (PSHUFD).

Nasleduje zdrojový kód makra výpočtu:

```
%macro m_gfmul 0        ; počet vstupných par. makra je 0
movdqu xmm3,xmm0        ; XMM3 = XMM0
pclmulqdq xmm3,xmm1,0       ; násobenie bez prenosu
movdqu xmm4,xmm0        ; XMM4 = XMM0
pclmulqdq xmm4,xmm1,16      ; násobenie bez prenosu
movdqu xmm5,xmm0        ; XMM5 = XMM0
pclmulqdq xmm5,xmm1,1       ; násobenie bez prenosu
movdqu xmm6,xmm0        ; XMM6 = XMM0
pclmulqdq xmm6,xmm1,17      ; násobenie bez prenosu
pxor xmm4,xmm5      ; XMM5 XOR XMM4
movdqu xmm5,xmm4        ; XMM5 = XMM4
psrldq xmm4,8       ; logický bytový posun vpravo
pslldq xmm5,8       ; logický bytový posun vľavo
pxor xmm3,xmm5      ; XMM3 XOR XMM5
pxor xmm6,xmm4      ; XMM6 XOR XMM4
movdqu xmm7,xmm3        ; XMM7 = XMM3
movdqu xmm8,xmm6        ; XMM8 = XMM6
pslld xmm3,1        ; logický bytový posun vľavo
pslld xmm6,1        ; logický bytový posun vľavo
pslld xmm7,31       ; logický bytový posun vľavo
pslld xmm8,31       ; logický bytový posun vľavo
movdqu xmm9,xmm7        ; XMM9 = XMM7
pslldq xmm8,4       ; logický bytový posun vľavo
pslldq xmm7,4       ; logický bytový posun vľavo
psrldq xmm9,12      ; logický bytový posun vpravo
por xmm3,xmm7       ; XMM7 OR XMM3
por xmm6,xmm8       ; XMM8 OR XMM6
por xmm6,xmm9       ; XMM9 OR XMM6
movdqu xmm7,xmm3        ; XMM7 = XMM3
movdqu xmm8,xmm3        ; XMM8 = XMM3
movdqu xmm9,xmm3        ; XMM9 = XMM3
pslld xmm7,31       ; logický bytový posun vľavo
```

```
pslld xmm8,30       ; logický bytový posun vľavo
pslld xmm9,25       ; logický bytový posun vľavo
pxor xmm7,xmm8      ; XMM5 XOR XMM4
pxor xmm7,xmm9      ; XMM5 XOR XMM4
movdqu xmm8,xmm7    ; XMM8 = XMM7
pslldq xmm7,12      ; logický bytový posun vľavo
psrldq xmm8,4       ; logický bytový posun vpravo
pxor xmm3,xmm7      ; XMM7 XOR XMM3
movdqu xmm2,xmm3    ; XMM2 = XMM3
movdqu xmm4,xmm3    ; XMM4 = XMM3
movdqu xmm5,xmm3    ; XMM5 = XMM3
psrld xmm2,1        ; logický bytový posun vpravo
psrld xmm4,2        ; logický bytový posun vpravo
psrld xmm5,7        ; logický bytový posun vpravo
pxor xmm2,xmm4      ; XMM4 XOR XMM2
pxor xmm2,xmm5      ; XMM5 XOR XMM2
pxor xmm2,xmm8      ; XMM8 XOR XMM2
pxor xmm3,xmm2      ; XMM2 XOR XMM3
pxor xmm6,xmm3      ; XMM3 XOR XMM6
; výsledok v registri XMM6
%endmacro       ; koniec makra
```

6.4.2 Šifrovanie v móde AES-GCM

Zdrojový kód procesu šifrovania je pre šifrovanie jedného bloku dát. Pri šifrovaní väčšieho počtu blokov sa funkcia volala viackrát (napr. pre 4 bloky 4-krát).

```
movdqu xmm0,[rcx]       ; čitač do XMM0
movdqu xmm8,[rdx]       ; otvorený text do XMM8
movdqu xmm1,[r8]        ; kľúč do XMM1

pxor xmm0,xmm1      ; nultá iterácia AddRoundKey
m_key_expansion 0x1     ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 1. iterácia šifrovania
m_key_expansion 0x2     ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 2. iterácia šifrovania
m_key_expansion 0x4     ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 3. iterácia šifrovania
m_key_expansion 0x8     ; volanie makra expanzie kľúča
```

```
AESENC xmm0,xmm1        ; 4. iterácia šifrovania
m_key_expansion 0x10
AESENC xmm0,xmm1
m_key_expansion 0x20
AESENC xmm0,xmm1
m_key_expansion 0x40
AESENC xmm0,xmm1
m_key_expansion 0x80
AESENC xmm0,xmm1
m_key_expansion 0x1b
AESENC xmm0,xmm1
m_key_expansion 0x36
AESENCLAST xmm0,xmm1        ; 10. iterácia šifrovania

; Šifrovaný text = Otvorený text XOR Zašifrovaný čítač
pxor xmm0,xmm8
movdqu [rax],xmm0        ; uloženie výsledku do RAX
```

6.4.3 Dešifrovanie v móde AES-GCM

Zdrojový kód procesu dešifrovania je pre dešifrovanie jedného bloku dát. Pri dešifrovaní väčšieho počtu blokov sa funkcia volala viackrát (napr. pre 4 bloky 4-krát).

```
movdqu xmm0,[rcx]        ; čitač do XMM0
movdqu xmm8,[rdx]        ; šifrovaný text do XMM8
movdqu xmm1,[r8]        ; kľúč do XMM1

pxor xmm0,xmm1        ; nultá iterácia AddRoundKey
m_key_expansion 0x1        ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 1. iterácia šifrovania
m_key_expansion 0x2        ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 2. iterácia šifrovania
m_key_expansion 0x4        ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 3. iterácia šifrovania
m_key_expansion 0x8        ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 4. iterácia šifrovania
m_key_expansion 0x10
AESENC xmm0,xmm1
m_key_expansion 0x20
```

```
AESENC xmm0,xmm1
m_key_expansion 0x40
AESENC xmm0,xmm1
m_key_expansion 0x80
AESENC xmm0,xmm1
m_key_expansion 0x1b
AESENC xmm0,xmm1
m_key_expansion 0x36
AESENCLAST xmm0,xmm1        ; 10. iterácia šifrovania

; Dešifrovaný text = Šifrovaný text XOR Zašifrovaný čítač
pxor xmm0,xmm8
movdqu [rax],xmm0        ; uloženie výsledku do RAX
```

6.4.4 Autentizácia v móde AES-GCM

Zdrojový kód procesu autentizácie je realizovaný v cykloch. Cyklus sa opakuje toľkokrát, koľko je šifrovaných blokov. Výstupom funkcie je autentizačný tag.

```
movdqu xmm1,[r8]        ; kľúč do XMM1
movdqu xmm14,[rcx]        ; čítač 0 do XMM14
movdqu xmm0,[HashKey]        ; Hashovací kľúč do XMM0

pxor xmm0,xmm1        ; nultá iterácia AddRoundKey
pxor xmm14,xmm1        ; nultá iterácia AddRoundKey
m_key_expansion 0x1        ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 1. iterácia šifrovania čítača
AESENC xmm14,xmm1        ; 1. iterácia šifrovania hash kľúča
m_key_expansion 0x2        ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 2. iterácia šifrovania čítača
AESENC xmm14,xmm1        ; 2. iterácia šifrovania hash kľúča
m_key_expansion 0x4        ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 3. iterácia šifrovania čítača
AESENC xmm14,xmm1        ; 3. iterácia šifrovania hash kľúča
m_key_expansion 0x8        ; volanie makra expanzie kľúča
AESENC xmm0,xmm1        ; 4. iterácia šifrovania čítača
AESENC xmm14,xmm1        ; 4. iterácia šifrovania hash kľúča
m_key_expansion 0x10
AESENC xmm0,xmm1
```

```
AESENC xmm14,xmm1
m_key_expansion 0x20
AESENC xmm0,xmm1
AESENC xmm14,xmm1
m_key_expansion 0x40
AESENC xmm0,xmm1
AESENC xmm14,xmm1
m_key_expansion 0x80
AESENC xmm0,xmm1
AESENC xmm14,xmm1
m_key_expansion 0x1b
AESENC xmm0,xmm1
AESENC xmm14,xmm1
m_key_expansion 0x36
AESENCLAST xmm0,xmm1        ; 10. iterácia šifrovania čítača
AESENCLAST xmm14,xmm1       ; 10. iterácia šifrovania hash kľúča

movdqu xmm1,[AuthData]      ; Dodatočné auten. dáta do XMM1
mov qword rbx,[NumberOfBlocks]      ; RBX = počet blokov

.gfmul:      ; cyklus násobenia v GF
movdqu xmm15,[rdx]      ; XMM15 = blok šifrovaného textu
m_gfmul      ; volanie makra násobenia v GF
pxor xmm6,xmm15      ; výsledok GF XOR šifrovaný text
movdqu xmm1,xmm6      ; subhash do xmm1
add rdx,16      ; posun ukazateľa na šifrovaný text
dec rbx      ; dekrementácia RBX o 1
jnz .gfmul      ak je RBX>0, skok na .gfmul

movdqu xmm15,[lenAC]      ; XMM15 = len(A)||len(C)
m_gfmul      ; volanie makra násobenia v GF
pxor xmm6,xmm15      ; výsledok GF XOR len(A)||len(C)
movdqu xmm1,xmm6      ; subhash do xmm1
m_gfmul      ; volanie makra násobenia v GF
pxor xmm6,xmm15      ; výsledok GF XOR zašifrovaný čítač 0 - v XMM6
je finálny autentizačný hash

movdqu [rax],xmm6      ; výsledok do RAX
```

7 TESTOVANIE A VYHODNOTENIE VÝPOČTOV

7.1 Algebraická časť

Testovanie elementárnych operácií s maticami a vektormi boli zamerané na porovnanie rýchlosti výpočtu vektorových výpočtových jednotiek SSE, AVX a FMA na platforme x86-64, a následne tiež s výpočtami z [1], ktoré boli realizované na platforme x86-32. V rámci tejto práce bolo vytvorených 12 funkcií v jazyku symbolických inštrukcií. Tieto funkcie boli testované na príklade diskrétnej kosínovej transformácie ako maticového počtu v programe Matlab. Pri testovaní boli použité nasledujúce rozmery matice:

- Rád matice 14 000 pre 32 bitové verzie kódov v jednoduchej presnosti.
- Rád matice 11 000 pre 32 bitové verzie kódov v dvojitej presnosti.
- Rád matice 14 000 pre 64 bitové verzie kódov v jednoduchej presnosti.
- Rád matice 11 000 pre 64 bitové verzie kódov v dvojitej presnosti.
- Rád matice 33 000 pre 64 bitové verzie kódov v jednoduchej presnosti.
- Rád matice 33 000 pre 64 bitové verzie kódov v dvojitej presnosti.

Rád matice bol závislý na maximálnej možnej veľkosti, ktorú program Matlab dokázal spracovať. V 64 bitovej verzii Matlabu bolo možné použiť viac ako dvojnásobný rád matice, čo je jeden z prínosov programov vyvinutých na platforme x86-64. Pre obmedzenie veľkosti matice v 32 bitovom Matlabe boli testy v 64 bitovej verzii opakované dvakrát, a to ako pre najväčšiu možnú maticu v 64 bitovej verzii, tak pre maticu rovnakých rozmerov, aká vola použitá v 32 bitovej verzii Matlabu.

Výpočty pracovali s číslami v jednoduchej a dvojitej presnosti. Prínosom výpočtových jednotiek AVX a FMA je dvojnásobná veľkosť registrov YMM, ktoré majú šírku 256 bitov, zatiaľ čo výpčotová jednotka disponuje registrami XMM o veľkosti 128 bitov. Preto bolo možné u vektorových jednotiek AVX a FMA v každej funkcii pracovať s dvojnásobným počtom operandov. Je nutné uviesť, že mnoho inštrukcií sa správa inak pri práci s XMM registrami, a inak pri práci s YMM registrami. YMM registre sa v podstate správajú ako dva spojené XMM registre, nie ako jeden celok. Preto bolo nutné využiť inštrukčné súbory pre výpočtové jednotky pracujúce s YMM registrami (AVX/FMA), a to najmä permutačné a naplňovacie inštrukcie. Ďalší významným faktom je, že v 64 bitovom programovom prostredí je k dispozícií pre všetky výpočtové jednotky dvakrát viac registrov, ako v 32 bitovom prostredí. Tým pádom bolo možné vytvoriť a testovať funkcie, ktoré 32 bitové programovanie neumožňuje.

Doby trvania jednotlivých výpočtov sú uvedené v tabuľke 7.1. Tabuľka posky-

Tab. 7.1: Výsledky testov algebraickej časti

	x86-32	**x86-64**	**x86-64**
Rád Matice SP	14000	14000	33000
Rád Matice DP	11000	11000	33000
	ZĽAVA SP [ms]		
SSE Shuffle	39,9	40,4	201,2
SSE Unpack	39,7	39,9	201,7
AVX (xmm)	38,9		
FMA (xmm)	37,5		
AVX (ymm)		35,4	183,6
FMA (ymm)		35,9	182,3
	SPRAVA SP [ms]		
SSE	39,7	42,2	205,2
AVX (xmm)	38,3		
AVX (ymm)		36,7	184,7
	ZĽAVA DP [ms]		
SSE Shuffle		51,7	504,6
SSE Unpack		51,2	468,6
AVX (ymm)	44,4	45,1	391,4
FMA (ymm)	44,5	43,2	382,6
	SPRAVA DP [ms]		
SSE		79,8	812,9
AVX (ymm)	43,9	42,7	386,2

tuje prehľad všetkých testovaných funkcií, rovnako ako rád matice, platformu, typ násobenia a danú presnosť.

Prvý stĺpec označuje názov funkcie, prípadne použité registre. Druhý a tretí stĺpec porovnáva doby výpočtov funkcií v 32 bitovom a 64 bitovom prostredí pre rovnaký rád matice. Posledný stĺpec zobrazuje doby výpočtov funkcií v 64 bitovom prostredí pre maximálny možný rád matice.

Z výsledkov porovnania 32 bitového a 64 bitového programovacieho prostredia je možné vidieť, že výpočtové jednotky novej generácie AVX a FMA neprinášajú pri týchto výpočtoch výrazné zrýchlenie doby výpočtu, ktoré by teoreticky malo byť dvojnásobné. Jediným výrazným rozdielom je násobenie sprava v dvojitej presnosti, kde je v 64 bitovom prostredí možné sledovať približne 46,5% zrýchlenie výpočtu pre menší rád matice a približne 52,5% zrýchlenie pre väčší rád matice. Všetky ostatné funkcie neprinášajú ako z pohľadu 64 bitového programovania, tak z pohľadu

novej generácie vektorových jednotiek takmer žiadny prínos.

To, že sa teoretické zrýchlenie nepotvrdilo, môže byť spôsobené niekoľkými faktormi. Jedným z nich sú atribúty samotných inštrukcií, a to hlavne priepustnosť a odozva. Tieto dve veličiny sú odlišné pre rôzne inštrukcie. Inými slovami, na vykonanie jednej inštrukcie potrebuje mikroprocesor viac strojových cyklov, a teda viac času, ako pre inú inštrukciu. Taktiež je nutné uviesť, že každá inštrukcia má pevne dané výpočtové porty, na ktorých sa môže vykonávať. Niektoré inštrukcie dokáže procesor spracovať na troch výpočtových portoch, zatiaľ čo iné inštrukcie len na jednom. Z toho vyplýva, že v jednom strojovom cykle je možné spracovať buď jednu, alebo tri inštrukcie rovnakého typu, čo je zásadný rozdiel.

Keďže v tejto práci boli testované relatívne jednoduché operácie násobenia matice a vektora, resp. vektora a matice, zdrojové kódy väčšinou nevyžadovali implementáciu výpočtovo náročných operácií, ako je napr. transpozícia matice. Najnáročnejší výpočet spomedzi vytvorených je násobenie sprava, kde je nutné využiť buď transpozíciu matice, alebo inštrukcie horizontálneho sčítania, ktoré sú veľmi pomalé. Práve v prípade takto výpočtovo náročných operácií je možné vidieť prínos nových výpočtových jednotiek, kde je urýchlenie výpočtu veľmi znateľné. Je možné, že v prípade pokročilejších algebraických výpočtov, ktoré pracujú s vyššou presnosťou a vyžadujú si rôzne transpozície a horizontálne sčítania, bude prínos novej generácie vektorových jednotiek značný, čo v podstate dokazuje aj výsledok násobenia sprava v dvojitej presnosti.

Oproti 32 bitovému programovaniu v jazyku symbolických inštrukcií je 64 bitové programové prostredie odlišné, no prináša niekoľko výhod. Asi najvýraznejšou zmenou v týchto dvoch typoch programových prostredí je konvencia volania funkcií. Zatiaľ čo v 32 bitovom programovaní je možné použiť rôzne konvencie volania, 64 bitové programovanie používa len jednu. Výhodou však je dvojnásobný počet XMM/YMM registrov, rovnako ako všeobecných reigstrov. Vďaka tomu bolo možné v 64 bitovom prostredí naprogramovať také výpočtové funkcie, ktoré v 32 bitovom programovom prostredí nebolo možné vytvoriť. Konkrétne sa v rámci práce jednalo o rozmer matice. Vďaka vyššiemu počtu registrov bolo možné v 64 bitovom prostredí pracovať v jednom kroku s takým rádom matice, ktorý 32 bitové prostredie nezvládne. Preto nie pre všetky funkcie v 64 bitovom prostredí bolo možné vytvoriť ekvivalentné funkcie v 32 bitovom programovom prostredí, čo vyplýva aj z tab.7.1. Ďalším prínosom 64 bitového programovania je, z pohľadu konečného testovania vytvorených funkcií a z pohľadu programu Matlab, možnosť pracovať s vyšším rádom matice, čo taktiež zobrazuje zmienená tabuľka.

7.2 Kryptografická časť

V kryptografickej časti bolo úlohou implementovanie autentizovaného šifrovania v podobe blokovej šifry AES v operačnom móde GCM. Po naštudovaní problematiky boli v jazyku symbolických inštrukcií vytvorené tri funkcie.

Prvá funkcia implementuje šifrovanie v móde GCTR. Pri vytváraní zdrojového kódu boli využité inštrukcie pre podporu AES šifrovania, ktoré poskytuje inštrukčný súbor AES-NI. Vďaka týmto inštrukciám sa všetky používané transformácie blokovej šifry AES vykonajú jedinou inštrukciou, čo by inak vyžadovalo zložitý proces implementácie týchto transformácií.

Druhá funkcia je funkcia dešifrovania. Keďže dešifrovanie v móde GCTR sa neimplementuje inverznou šifrou, ale šifrovacím algoritmom, je funkcia takmer totožná s tou šifrovacou. Rozdielom je predávanie iných parametrov funkcií, s ktorými sa pracuje.

Posledná funkcia je autentizácia. Táto funkcia počíta autentizačný tag, ktorý musí byť rovnaký na oboch stranách prenosu. Tým sa potvrdí, že so správou nebolo v priebehu prenosu nijak manipulované. Autentizačná funkcia využíva inštrukcie násobenia bez prenosu, ktoré celý algoritmus urýchľujú.

Keďže mód GCTR je vo svojom princípe rovnaký, ako mód CTR, boli pre overenie správnosti šifrovania a dešifrovania použité testovacie vektory definované v NIST Special Publication 800-38A. Vytvorené funkcie boli testované v programe Matlab. Výstupom bolo porovnanie očakávaného šifrovaného textu a porovnanie otvoreného textu pred šifrovaní a po dešifrovaní. Výstup programu Matlab vyzeral nasledovne:

```
ExpectedCiphertext4Blocks =

874D6191B620E3261BEF6864990DB6CE
9806F66B7970FDFF8617187BB9FFFDFF
5AE4DF3EDBD5D35E5B4F09020DB03EAB
1E031DDA2FBE03D1792170A0F3009CEE

OutputCiphertext4Blocks =

874D6191B620E3261BEF6864990DB6CE
9806F66B7970FDFF8617187BB9FFFDFF
5AE4DF3EDBD5D35E5B4F09020DB03EAB
1E031DDA2FBE03D1792170A0F3009CEE
```

```
OriginalPlaintext4Blocks =

6BC1BEE22E409F96E93D7E117393172A
AE2D8A571E03AC9C9EB76FAC45AF8E51
30C81C46A35CE411E5FBC1191A0A52EF
F69F2445DF4F9B17AD2B417BE66C3710

DecryptedPlaintext4Blocks =

6BC1BEE22E409F96E93D7E117393172A
AE2D8A571E03AC9C9EB76FAC45AF8E51
30C81C46A35CE411E5FBC1191A0A52EF
F69F2445DF4F9B17AD2B417BE66C3710

Výstupný a očakávaný Ciphertext sa zhoduje - šifrovanie úspešné.
Šifrovaný a dešifrovaný Plaintext sa zhoduje - dešifrovanie úspešné.
```

Po overení funkčnosti algoritmov šifrovania a dešifrovania bol implementovaný finálny algoritmus AES-GCM, ktorý pracoval s veľkosťou kľúča 128 bitov. V Matlabe bolo teda okrem porovnania šifrovania a dešifrovania uvedené, či bola autentizácia úspešná, alebo nie. V prípade, že sa autentizačný tag na strane odosielateľa zhodoval s autentizačným tagom na strane prijímateľa, autentizácia prebehla úspešne. Celkový výstup algortimu AES-GCM vyzeral v programe Matlab nasledovne:

```
Plaintext4Blocks =

6BC1BEE22E409F96E93D7E117393172A
AE2D8A571E03AC9C9EB76FAC45AF8E51
30C81C46A35CE411E5FBC1191A0A52EF
F69F2445DF4F9B17AD2B417BE66C3710

Ciphertext4Blocks =

5DEAC2DE4933CEF5F19D09C68FC36484
C401492F668A9BD32003A7B75215E215
D85425D953AD7CD731F1F0C20F66F911
469263BDCBC50A195D4371EC76279212
```

DecryptedPlaintext4Blocks =

6BC1BEE22E409F96E93D7E117393172A
AE2D8A571E03AC9C9EB76FAC45AF8E51
30C81C46A35CE411E5FBC1191A0A52EF
F69F2445DF4F9B17AD2B417BE66C3710

AuthTagSender =

1FD37A28155D05E0B78454899BC7C36F

AuthTagReceiver =

1FD37A28155D05E0B78454899BC7C36F

Šifrovaný a dešifrovaný Plaintext sa zhoduje.
Tag na strane odosielatela a prijímatela sa zhoduje - autentizácia úspešná.

Všetky výsledky uvedené v tejto práci boli testovaný na tom istom osobnom počítači, ktorého parametre sú nasledovné:

- **Operačný systém:** Miscrosoft Windows 7 Professional, 64 bitová verzia
- **Mikroprocesor:** Intel Core i5-6500 (mikroarchitektúra Skylake)
- **Frekvencia a počet jadier mikroprocesoru:** 3,2 GHz, 4 jadrá
- **Operačná pamäť:** 16 GB

Zdrojové kódy pre 32 bitové programové prostredie boli testované v 32 bitovom programe Matlab (verzia R2011b), zdrojové kódy pre 64 bitové programové prostredie boli testované v 64 bitovom programe Matlab (verzia R2016b).

8 ZÁVER

Cieľom práce bolo naštudovať a spracovať problematiku základných operácií s maticami a vektormi s využitím vektorových výpočtových jednotiek procesorov, rovnako ako kryptografie, konkrétne blokovej šifry AES pracujúcej v operačnom móde Galois Counter Mode (GCM), na platforme x86-64, teda v 64 bitovom programovom prostredí. Preto sa aj obsah práce delí na dve základné časti. Prvá časť sa zaoberá porovnaním vektorových výpočtových jednotiek staršej a novšej generácie a porovnaním 32 bitového a 64 bitového programového prostredia. Druhá časť je venovaná implementácii algoritmu AES-GCM.

V prvej časti sú popísané testované operácie s maticami a vektormi, spolu s ukážkami testovaných zdrojových kódov pre každú testovanú výpočtovú jednotku. Základnými použitými operáciami sú násobenie matice vektorom, násobenie vektora maticou, transpozícia matice a transpozícia vektora. Táto časť poskytuje prehľad toho, akým spôsobom výpočtové jednotky procesoru spracovávajú dáta, ale tiež popis 64 bitového programového prostredia a jeho odlišností od 32 bitového programového prostredia. Tieto fakty boli zohľadnené pri implementácii testovaných výpočtov. Vytvorené zdrojové kódy v jazyku symbolických inštrukcií boli skompilované a linkované do dynamickej knižnice funkcii. Testovaným výpočtom bola diskrétna kosínová transformácia verzie IV, ktorá využívala funkcie definované v dynamickej knižnici. Výstupom testovania je porovnanie vektorových výpočtových jednotiek z hľadiska staršej a novšej generácie, ale aj z hľadiska programového prostredia. Výsledky a zhodnotenie testovania sú zahrnuté v práci.

V druhej časti je rozobraná problematika symetrickej kryptografie. Jadro tejto časti tvorí autentizované šifrovanie, konkrétne symetrická bloková šifra AES pracujúca v operačnom móde GCM. Práca poskytuje základný popis konceptu symetrickej kryptografie s dôrazom na blokové šifry. Následne je vysvetlený operačný mód blokovej šifry CTR, ktorý je v princípe rovnaký, ako mód GCTR, ktorý tvorí prvú časť operačného módu GCM. Následne je popísaný samotný operačný mód GCM. Keďže jedným zo základov práce je štandard pokročilého šifrovania AES, práca poskytuje jeho detailný popis - informácie o blokovej šifre AES vrátane procesu šifrovania a dešifrovania, expanzie kľúča, či využívaných transformačných funkcií a inverznej šifry. Ďalej sú v kryptografickej časti práce popísané inštrukčné sady, ktoré prinášajú súbory inštrukcií pre zrýchlenie kryptografických výpočtov. Sú to inštrukčné sady AES-NI, CLMUL a SHA Extensions. Výstupom tejto časti je implemetovaný algoritmus AES-GCM, ktorý bol taktiež skompilovaný a linkovaný do dynamickej knižnice funkcii. Správnosť funkcií bola overená testovaním v programe Matlab. Overovala sa správnosť šifrovania, dešifrovania a autentizácie. Výstup testovania je zahrnutý v práci.

Súčasťou práce sú tiež dve laboratórne úlohy, ktoré sú v prílohách. Prvá úloha je zameraná na vytvorenie algoritmu AES-GCM, kde je najprv tento operačný mód popísaný, následne sú vysvetlené špeciálne inštrukcie, ktoré urýchľujú výpočet a zjednodušujú zdrojový kód. Praktická časť laboratórnej úlohy ukazuje vytvorenie knižnice funkcií a následne jej implementáciu v programe Matlab. Druhá úloha implementuje výpočet násobenia vektora maticou z pohľadu novej generácie výpočtových jednotiek mikroprocesorov, konkrétne AVX a FMA. Teoretický úvod práce popisuje vlastnosti týchto výpočtových jednotiek a následne je v praktickej časti popísaný postup implementácie operácie násobenia zľava pre obe vektorové jednotky, spolu s vytvorením skriptu pre testovanie doby výpočtov a overenie správnosti vytvoreného zdrojového kódu. V rámci príloh je tiež uvedený zoznam inštrukcií pre podporu kryptografických operácií, ktoré boli v laboratórnych úlohách použité.

LITERATÚRA

[1] ŠLENKER, S. *Výpočetní jednotky procesorů poslední generace a jejich využití.* Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2015.

[2] STALLINGS, W. *Cryptography and Network Security Principles and Practices.* Piate vydanie. Prentice Hall, 2016. ISBN 0136097049.

[3] BURDA, K. *Bezpečnost informačních systémů.* Prvé vydanie. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2013.

[4] PAAR, Ch.; PELZL, J. *Understanding cryptography: a textbook for students and practitioners.* Springer Science & Business Media, 2009. ISBN 3642041019.

[5] AKDEMIR, K.; DIXON, M.;FEGHALI, W.; FAY, P.; GOPAL, V.; GUILFORD,J.; OZTURK, E.; WOLRICH, G.; ZOHAR, R. *Breakthrough AES Performance with Intel® AES New instructions* [online]. 2010, [cit. 6.11.2016]. Dostupné z URL: <https://software.intel.com/sites/default/files/m/d/4/1/d/8/10TB24_Breakthrough_AES_Performance_with_Intel_AES_New_Instructions.final.secure.pdf>.

[6] SHAY, G.; KOUNAVIS, M.E.; *Intel® Carry-Less Multiplication Instruction and its Usage for Computing the GCM Mode* [online]. Apríl 2014, [cit. 8.11.2016]. Dostupné z URL: <https://software.intel.com/sites/default/files/managed/72/cc/clmul-wp-rev-2.02-2014-04-20.pdf>.

[7] GULEY, S.; GOPAL, V.; YAP, K.; FENHALI, W.; GUILFORD, J.; WOLRICH, G.; *Intel® SHA Extensions* [online]. Apríl 2014, [cit. 8.11.2016]. Dostupné z URL: <https://software.intel.com/sites/default/files/article/402097/intel-sha-extensions-white-paper.pdf>.

[8] MICROSOFT CORPORATION *Programming Guide for 64-bit Windows* [online]. 2017, [cit. 11.5.2017]. Dostupné z URL: <https://msdn.microsoft.com/en-us/library/windows/desktop/bb427430(v=vs.85).aspx>.

[9] MICROSOFT CORPORATION *Overview of x64 Calling Conventions* [online]. 2017, [cit. 11.5.2017]. Dostupné z URL: <https://msdn.microsoft.com/en-us/library/ms235286.aspx>.

[10] MICROSOFT CORPORATION *Register Usage* [online]. 2017, [cit. 11.5.2017]. Dostupné z URL: <https://msdn.microsoft.com/en-us/library/9z1stfyw.aspx>.

ZOZNAM SKRATIEK

AAD	Additional Authenticated Data
AES	Advanced Encryption Standard
AES-NI	Advanced Encryption Standard - New Instructions
AVX	Advanced Vector Extensions
CLMUL	Carry-Less Multiplication
CTR	Counter
FMA	Fused Multiply Add
GCM	Galois Counter Mode
GCTR	Galois Counter
GF	Galois Field
GHASH	Galois Hash
MAC	Message Authentication Code
ROL	Rotation on Left
ROR	Rotation on Right
SHA	Secure Hash Algorithm
SIMD	Single-Instruction Multiple Data
SSE	Streaming SIMD Extensions

ZOZNAM PRÍLOH

A IMPLEMENTACE ALGORITMU AES-GCM NA PLATFORMĚ X86-64

A.1 Operační mód Galois Counter (GCM) blokové šifry

Operační mód GCM je navržen tak, aby kromě samotného šifrování poskyytoval i autentizaci. Mluvíme tedy o autentizovaném šifrování. Skládá se ze dvou částí, a to z:

- operačního módu čítače (Counter Mode (CTR)),
- výpočtu autentizačního heše GMAC (Galois Message Authentication Code).

Výstupem funkce GMAC je Galoisův heš GHASH, který slouží na autentizaci zprávy, tedy bylo-li ze zprávou v průběhu přenosu manipulováno, nebo ne, a tak chrání důvěrnost otevřeného textu a zajišťuje integritu zprávy.

Operační mód je složen z blokové šifry (např. AES), která je základem, a z násobitele Galoisova pole. Šifra musí mít velikost 128 bitů. Odesílatel nejprve zašifruje v módu Galois CTR (GCTR) a následně vypočte autentizační GHASH.

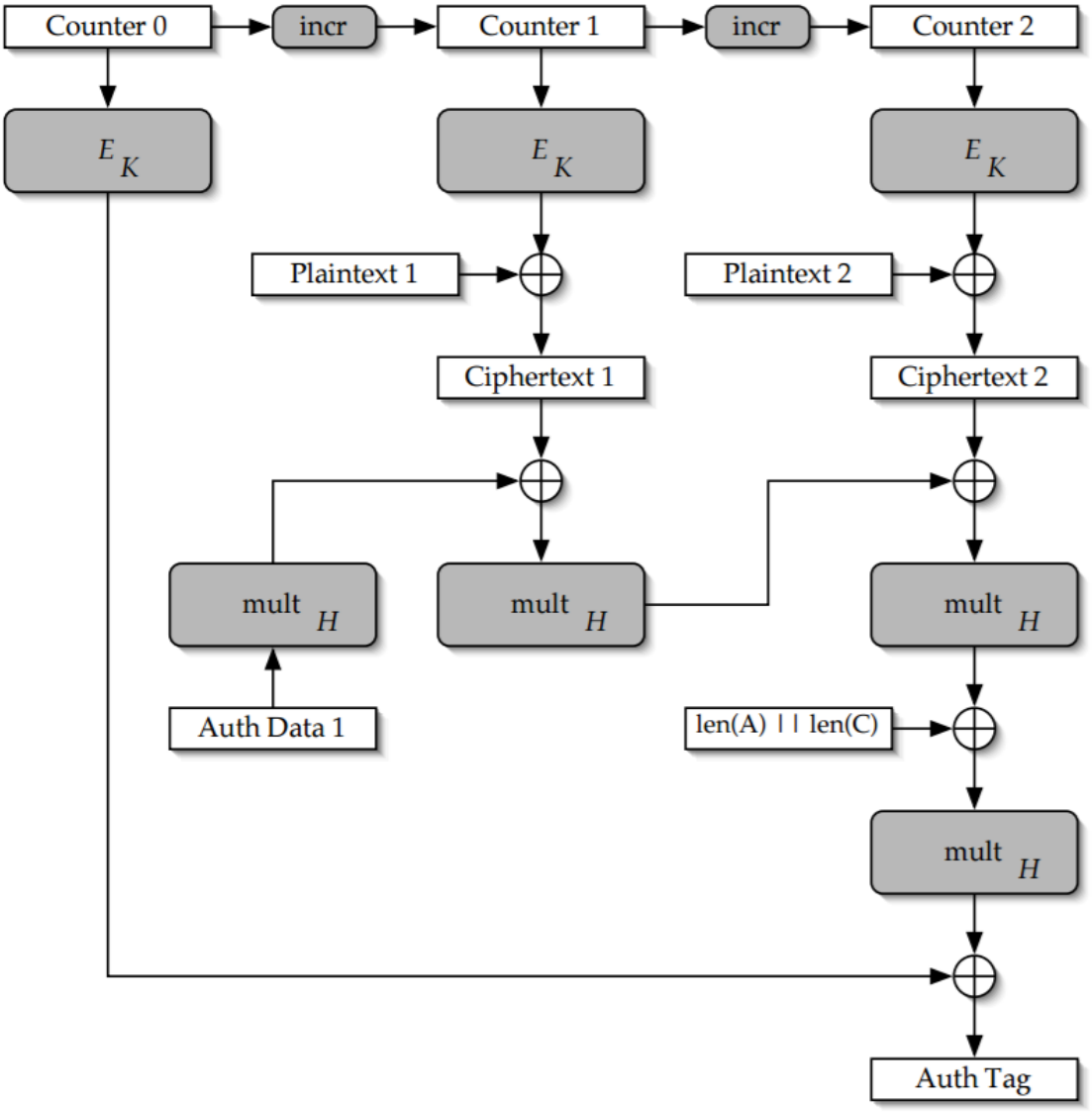
Operační mód čítače je jendoduchý proces, kdy je dán počáteční čítač o nějaké hodnotě. Ten je následně šifrován blokovou šifrou a XORován s otevřeným textem. Výstupem je šifrovaný text. Následný otevřený text je XORován zašifrovaným **inkrementovaným** čítačem o 1. Jinými slovy, k počáteční hodnotě čítače se přičte 1, tato nová hodnota se zašifruje a XORuje s otevřeným textem. Čítač je pro každý nový blok inkrementován. V procesu dešifrování je postup zcela totožný - neprovádí se inverzní šifra. Zašifrovaný čítač se XORuje se šifrovaným textem a výstupem je otevŘený text.

Při autentizaci vykoná GCM zřetězené násobení Galoisova pole. Pro každý otevřený text x_i je odvozen autentizační parametr g_i. Ten je vypočten XORem aktuálního šifrovaného textu y_i a g_i a následně násoben klíčem H. Hodnota klíče je vygenerována šifrováním nulového vstupu blokovou šifrou. Všechny operace násobení jsou prováděny ve 128 bitovém Galoisovém poli $GF(2^{128})$ s neredukovatelným polynomem $P(x) = x^{128} + x^7 + x^2 + x + 1$. Jelikož pro každou kždý blok šifry je nutné jenom jedno násobení, GCM přidává k celkovému procesu jenom malou režii.

Blokové schéma módu GCM je na obr. A.1. V horní části schématu je šifrování v operačním móde CTR, který je první částí algoritmu GCM. Výstup po šifrovaní, tedy šifrovaný text, je použit v druhé části módu GCM, kterou je výpočet autentizačního tagu. Následuje vysvětlení jednotlivých bloků:

- Bloky ***Counter*** představují čítač.
- Bloky ***incr*** představují funkci inkrementace o 1.

- Bloky $\boldsymbol{E_K}$ představují proces šifrování s klíčem.
- Bloky ***Plaintext*** představují otevřený text.
- Bloky ***Ciphertext*** představují šifrovaný text.
- Bloky $\boldsymbol{mult_H}$ představují násobení v poli $GF(2^{128})$ s hashovacím klíčem H. H je 128 bitů nul zašifrovaných blokovou šifrou.
- Blok ***Auth Data*** představuje dodatočná autentizační data, co může být např. hlavička IP protokolu.
- Blok $\boldsymbol{len(A) \,||\, len(C)}$ představuje funkcu sloučení dvou 64 bitových stringů do jednoho 128 bitového stringu. $\boldsymbol{A}$ je délka dodatočných autentizačních dat, $\boldsymbol{C}$ je délka šifrovaného textu.
- Blok ***Auth Tag*** je výstupný tag pro autentizaci.



Obr. A.1: Blokové schéma operačního módu GCM

A.2 Bloková šifra AES

Advanced Encryption Standard je v současné době velmi rozšířená šifra. Podle velikosti použitého klíče se mění i počet iterací jednotlivých rund:

- klíč velikosti 128 bitů - 10 iterací,
- klíč velikosti 192 bitů - 12 iterací,
- klíč velikosti 256 bitů - 14 iterací.

První $N - 1$ iterací je složeno ze čtyř různých transformací - SubBytes, ShiftRows, MixColumns a AddRoundKey. V poslední iteraci se nepoužívá MixColumns, proto je poslední iterace složena jenom ze tří funkcí. Také je nutné poznamenat, že pŘed první iterací se provádí samostatně transformace AddRoundKey.

Transformace SubBytes je jednoduchá transformace substituce bytov, která se realizuje na základě tabulky S-Box, kterou definuje standard AES a má rozměr $16x16$ bytů.

Transformace ShiftRows bytově posouvá řádky. První řádek se nemění, druhý řádek je posunut o 1 byte vlevo, druhý řádek o 2 byte, třetí o 3 byte.

Transformace MixColumns mapuje každý byte sloupce na novou hodnotu. Tato transformace je v podstatě maticové násobení stavové matice.

Transformace AddRoundKey pracuje s expanzním klíčem, který je XORován se vstupem.

V procesu dešifrování, tedy mluvíme-li o inverzní šifre, jsou použity tyto transformace v inverzním režimu, kromě transformace AddRoundKey.

A.2.1 Instrukce pro podporu standardu AES - AES-NI

Jelikož transformace každé rundy blokové šifry AES způsobovali prodloužení a nepřehlednost kódu z pohledu programátora, byl vyvinut instrukční soubor AES-NI (Advanced Encryption Standard - New Instructions), který, kromě jiného, slučuje všechny transformace do jedné instrukce. Kromě toho jsou k dispozici instrukce pro dešifrování, expanzi klíče atd. Jejich výčet je možné najít v příloze. Níže jsou v tab. A.1 uvedeny instrukce, které budou využity v této laboratorní úloze.

A.2.2 Instrukce násobení bez přenosu CLMUL

Instrukce Carry-Less Multiplication, tedy násobení bez přenosu, se využívá v násobení v Galoisovom poli. Instrukce je k dispozici od mikroarchitektury Westmere, stejně jako AES-NI. Násobení bez přenosu je časovo náročná operace, která se využívá ve většine kryptografických systémech a standardech. Využívá ji i mód GCM, a to při násobení v Galoisovom poli. GCM násobí dvě 128 bitová čísla a výsledkem je 255 bitová hodnota. Tato operace se používá v první polovině výpočtu GHASH.

Tab. A.1: Použité AES-NI instrukce

Instrukce	**Popis**
AESENC xmm1,xmm2/m128	Provede se 1 runda šifrování dat v xmm1 s klíčem v xmm2/m128.
AESENCLAST xmm1,xmm2/m128	Provede poslední rundu šifrování dat v xmm1 s klíčem v xmm2/ m128.
AESKEYGENASSIST xmm1, xmm2/m128,imm8	Asistuje při expanzi rundovního klíče pomocí iterační konstanty v imm8. Pracuje se s daty v xmm2 a výsledek se uloží do xmm1.

Kromě CLMUL využívá výpočet a také instrukce bytového logického posunu vlevo/vpravo SLL/SRL (Shift Left/Right Logical), dále instrukce pro funkci OR a XOR a instrukce naplnění MOV. Následující tabulka popisuje instrukce bez přenosu, instrukce bytového posunu a jejich zápis.

Tab. A.2: Popis instrukcí CLMUL a SLL/SRL

Instrukce	**Popis**
PCLMULQDQ xmm1,xmm2/m128,imm8	Provede násobení dvou qwadwordů ze zdrojového a cílového operandu, které jsou vybrány pomocí masky, a uloží 128b výsledek do xmm1.
PSLLDQ xmm1,imm8	Provede logický bytový posun reg. xmm1 vlevo na základě masky. Posouvá se v nulách.
PSRLDQ xmm1,imm8	Provede logický bytový posun reg. xmm1 vpravo na základě masky. Posouvá se v nulách.

A.3 Zdrojové kódy

Zdrojové kódy jsou rozděleny na tři funkce:

- Šifrování.
- Dešifrovní.
- Autentizace.

Kódy odpovídají blokovému schématu A.1.

A.3.1 Šifrovací a dešifrovací část

V obou funkcích je použito makro pro expanzi klíče s jedním parametrem, kterým je maska. V samotném kódu se pak volá makro, čímž se kód spřehlední. Zdrojový kód makra:

```
%macro m_key_expansion 1      ; počet vstupních par. makra je 1
AESKEYGENASSIST xmm2,xmm1,%1
pshufd xmm2,xmm2,0xff      ; pomíchaní hodnot v XMM2 na základě masky
movdqu xmm3,xmm1      ; XMM3 = XMM1
pslldq xmm3,0x4      ; bytový posun vlevo na základě masky
pxor xmm1,xmm3      ; XMM3 XOR XMM1
movdqu xmm3,xmm1      ; XMM3 = XMM1
pslldq xmm3,0x4      ; bytový posun vlevo na základě masky
pxor xmm1,xmm3      ; XMM3 XOR XMM1
movdqu xmm3,xmm1      ; XMM3 = XMM1
pslldq xmm3,0x4      ; bytový posun vlevo na základě masky
pxor xmm1,xmm3      ; XMM3 XOR XMM1
pxor xmm1,xmm2      ; XMM2 XOR XMM1, XMM1 = výsledek
%endmacro      ; konec makra
```

Následuje zdrojový kód šifrování blokové šifry AES v módu CTR:

```
movdqu xmm0,[rcx]      ; čítač do XMM0
movdqu xmm8,[rdx]      ; otevřený text do XMM8
movdqu xmm1,[r8]      ; klíč do XMM1

pxor xmm0,xmm1      ; nultá iterace AddRoundKey
m_key_expansion 0x1      ; volání makra expanze klíče
AESENC xmm0,xmm1      ; 1. iterace šifrování
m_key_expansion 0x2      ; volání makra expanze klíče
AESENC xmm0,xmm1      ; 2. iterace šifrování
m_key_expansion 0x4      ; volání makra expanze klíče
AESENC xmm0,xmm1      ; 3. iterace šifrování
m_key_expansion 0x8      ; volání makra expanze klíče
AESENC xmm0,xmm1      ; 4. iterace šifrování
m_key_expansion 0x10
AESENC xmm0,xmm1
m_key_expansion 0x20
AESENC xmm0,xmm1
```

```
m_key_expansion 0x40
AESENC xmm0,xmm1
m_key_expansion 0x80
AESENC xmm0,xmm1
m_key_expansion 0x1b
AESENC xmm0,xmm1
m_key_expansion 0x36
AESENCLAST xmm0,xmm1        ; 10. iterace šifrování

; Šifrovaný text = Otevřený text XOR Zašifrovaný čítač
pxor xmm0,xmm8
movdqu [rax],xmm0        ; uložení výsledku do RAX
```

Následuje zdrojový kód dešifrování blokové šifry AES v módu CTR:

```
movdqu xmm0,[rcx]        ; čitač do XMM0
movdqu xmm8,[rdx]        ; šifrovaný text do XMM8
movdqu xmm1,[r8]         ; klíč do XMM1

pxor xmm0,xmm1           ; nultá iterace AddRoundKey
m_key_expansion 0x1      ; volání makra expanze klíče
AESENC xmm0,xmm1         ; 1. iterace šifrování
m_key_expansion 0x2      ; volání makra expanze klíče
AESENC xmm0,xmm1         ; 2. iterace šifrování
m_key_expansion 0x4      ; volání makra expanze klíče
AESENC xmm0,xmm1         ; 3. iterace šifrování
m_key_expansion 0x8      ; volání makra expanze klíče
AESENC xmm0,xmm1         ; 4. iterace šifrování
m_key_expansion 0x10
AESENC xmm0,xmm1
m_key_expansion 0x20
AESENC xmm0,xmm1
m_key_expansion 0x40
AESENC xmm0,xmm1
m_key_expansion 0x80
AESENC xmm0,xmm1
m_key_expansion 0x1b
AESENC xmm0,xmm1
m_key_expansion 0x36
```

```
AESENCLAST xmm0,xmm1        ; 10. iterace šifrování

; Dešifrovaný text = Šifrovaný text XOR Zašifrovaný čítač
pxor xmm0,xmm8
movdqu [rax],xmm0        ; uložení výsledku do RAX
```

A.3.2 Autentizační část

Ve funkce autentizace je použito makro pro násobení v Galoisovém poli. Makro nemá žádný vstupní parametr. V samotném kódu se pak volá makro, čímž se kód spřehlední. Zdrojový kód makra:

```
%macro m_gfmul 0        ; počet vstupních par. makra je 0
movdqu xmm3,xmm0        ; XMM3 = XMM0
pclmulqdq xmm3,xmm1,0        ; násobení bez přenosu
movdqu xmm4,xmm0        ; XMM4 = XMM0
pclmulqdq xmm4,xmm1,16        ; násobení bez přenosu
movdqu xmm5,xmm0        ; XMM5 = XMM0
pclmulqdq xmm5,xmm1,1        ; násobení bez přenosu
movdqu xmm6,xmm0        ; XMM6 = XMM0
pclmulqdq xmm6,xmm1,17        ; násobení bez přenosu
pxor xmm4,xmm5        ; XMM5 XOR XMM4
movdqu xmm5,xmm4        ; XMM5 = XMM4
psrldq xmm4,8        ; logický bytový posun vpravo
pslldq xmm5,8        ; logický bytový posun vlevo
pxor xmm3,xmm5        ; XMM3 XOR XMM5
pxor xmm6,xmm4        ; XMM6 XOR XMM4
movdqu xmm7,xmm3        ; XMM7 = XMM3
movdqu xmm8,xmm6        ; XMM8 = XMM6
pslld xmm3,1        ; logický bytový posun vlevo
pslld xmm6,1        ; logický bytový posun vlevo
pslld xmm7,31        ; logický bytový posun vlevo
pslld xmm8,31        ; logický bytový posun vlevo
movdqu xmm9,xmm7        ; XMM9 = XMM7
pslldq xmm8,4        ; logický bytový posun vlevo
pslldq xmm7,4        ; logický bytový posun vlevo
psrldq xmm9,12        ; logický bytový posun vpravo
por xmm3,xmm7        ; XMM7 OR XMM3
por xmm6,xmm8        ; XMM8 OR XMM6
```

```
por xmm6,xmm9       ; XMM9 OR XMM6
movdqu xmm7,xmm3        ; XMM7 = XMM3
movdqu xmm8,xmm3        ; XMM8 = XMM3
movdqu xmm9,xmm3        ; XMM9 = XMM3
pslld xmm7,31       ; logický bytový posun vlevo
pslld xmm8,30       ; logický bytový posun vlevo
pslld xmm9,25       ; logický bytový posun vlevo
pxor xmm7,xmm8      ; XMM5 XOR XMM4
pxor xmm7,xmm9      ; XMM5 XOR XMM4
movdqu xmm8,xmm7        ; XMM8 = XMM7
pslldq xmm7,12      ; logický bytový posun vlevo
psrldq xmm8,4       ; logický bytový posun vpravo
pxor xmm3,xmm7      ; XMM7 XOR XMM3
movdqu xmm2,xmm3        ; XMM2 = XMM3
movdqu xmm4,xmm3        ; XMM4 = XMM3
movdqu xmm5,xmm3        ; XMM5 = XMM3
psrld xmm2,1        ; logický bytový posun vpravo
psrld xmm4,2        ; logický bytový posun vpravo
psrld xmm5,7        ; logický bytový posun vpravo
pxor xmm2,xmm4      ; XMM4 XOR XMM2
pxor xmm2,xmm5      ; XMM5 XOR XMM2
pxor xmm2,xmm8      ; XMM8 XOR XMM2
pxor xmm3,xmm2      ; XMM2 XOR XMM3
pxor xmm6,xmm3      ; XMM3 XOR XMM6
; výsledek v registri XMM6
%endmacro       ; konec makra
```

Autentizační funkce má pak následující zdrojový kód:

```
movdqu xmm1,[r8]        ; klíč do XMM1
movdqu xmm14,[rcx]      ; čítač 0 do XMM14
movdqu xmm0,[HashKey]       ; Hashovací klíč do XMM0

pxor xmm0,xmm1      ; nultá iterace AddRoundKey
pxor xmm14,xmm1     ; nultá iterace AddRoundKey
m_key_expansion 0x1     ; volání makra expanze klíče
AESENC xmm0,xmm1        ; 1. iterace šifrování čítače
AESENC xmm14,xmm1       ; 1. iterace šifrování hash klíče
m_key_expansion 0x2     ; volání makra expanze klíče
```

```
AESENC xmm0,xmm1        ; 2. iterace šifrování čítače
AESENC xmm14,xmm1       ; 2. iterace šifrování hash klíče
m_key_expansion 0x4     ; volání makra expanze klíče
AESENC xmm0,xmm1        ; 3. iterace šifrování čítače
AESENC xmm14,xmm1       ; 3. iterace šifrování hash klíče
m_key_expansion 0x8     ; volání makra expanze klíče
AESENC xmm0,xmm1        ; 4. iterace šifrování čítače
AESENC xmm14,xmm1       ; 4. iterace šifrování hash klíče
m_key_expansion 0x10
AESENC xmm0,xmm1
AESENC xmm14,xmm1
m_key_expansion 0x20
AESENC xmm0,xmm1
AESENC xmm14,xmm1
m_key_expansion 0x40
AESENC xmm0,xmm1
AESENC xmm14,xmm1
m_key_expansion 0x80
AESENC xmm0,xmm1
AESENC xmm14,xmm1
m_key_expansion 0x1b
AESENC xmm0,xmm1
AESENC xmm14,xmm1
m_key_expansion 0x36
AESENCLAST xmm0,xmm1    ; 10. iterace šifrování čítače
AESENCLAST xmm14,xmm1   ; 10. iterace šifrování hash klíče

movdqu xmm1,[AuthData]      ; Dodatočná auten. data do XMM1
mov qword rbx,[NumberOfBlocks]      ; RBX = počet bloků

.gfmul:     ; cyklus nasobení v GF
movdqu xmm15,[rdx]      ; XMM15 = blok šifrovaného textu
m_gfmul     ; volání makra násobení v GF
pxor xmm6,xmm15     ; výsledek GF XOR šifrovaný text
movdqu xmm1,xmm6        ; subhash do xmm1
add rdx,16      ; posun ukazatele na šifrovaný text
dec rbx     ; dekrementace RBX o 1
jnz .gfmul      pokud RBX>0, skok na .gfmul
```

```
movdqu xmm15,[lenAC]        ; XMM15 = len(A) || len(C)
m_gfmul       ; volání makra násobení v GF
pxor xmm6,xmm15       ; výsledek GF XOR len(A) || len(C)
movdqu xmm1,xmm6       ; subhash do xmm1
m_gfmul       ; volání makra násobení v GF
pxor xmm6,xmm15       ; výsledek GF XOR zašifrovaný čítač 0 - v XMM6
je finální hash

movdqu [rax],xmm6       ; výsledek do RAX
```

A.3.3 Matlab - zdrojový kód skriptu testování

Po linkování a kompilování kódu do dynamické knihovny využijeme program Matlab pro testování všech tří funkcí. V Matlabu vytvoříme skript, který bude volat funkce z dynamické knihovny.

Skript v Matlabu:

```
%% Uzavření všech grafů a smazání všech proměnných
clear all; clc

%% Načítání DLL
hfile1 = 'aes_gcm.h';
[notfound,warnings]=loadlibrary('aes_gcm.dll', hfile1,'mfilename','aes_gcm_mx');

%% Definice operandů

Key = [swapbytes(uint32(hex2dec('2b7e1516')))
swapbytes(uint32(hex2dec('28aed2a6')))
swapbytes(uint32(hex2dec('abf71588')))
swapbytes(uint32(hex2dec('09cf4f3c')))];

Counter = [swapbytes(uint32(hex2dec('f0f1f2f3')))
swapbytes(uint32(hex2dec('f4f5f6f7')))
swapbytes(uint32(hex2dec('f8f9fafb')))
swapbytes(uint32(hex2dec('fcfdfeff')))];

Plaintext1 = [swapbytes(uint32(hex2dec('6bc1bee2')))
swapbytes(uint32(hex2dec('2e409f96')))
```

```
swapbytes(uint32(hex2dec('e93d7e11')))
swapbytes(uint32(hex2dec('7393172a')))];

Plaintext2 = [swapbytes(uint32(hex2dec('ae2d8a57')))
swapbytes(uint32(hex2dec('1e03ac9c')))
swapbytes(uint32(hex2dec('9eb76fac')))
swapbytes(uint32(hex2dec('45af8e51')))];

Plaintext3 = [swapbytes(uint32(hex2dec('30c81c46')))
swapbytes(uint32(hex2dec('a35ce411')))
swapbytes(uint32(hex2dec('e5fbc119')))
swapbytes(uint32(hex2dec('1a0a52ef')))];

Plaintext4 = [swapbytes(uint32(hex2dec('f69f2445')))
swapbytes(uint32(hex2dec('df4f9b17')))
swapbytes(uint32(hex2dec('ad2b417b ')))
swapbytes(uint32(hex2dec('e66c3710')))];

Ciphertext1 = [swapbytes(uint32(hex2dec('874d6191')))
swapbytes(uint32(hex2dec('b620e326')))
swapbytes(uint32(hex2dec('1bef6864')))
swapbytes(uint32(hex2dec('990db6ce')))];

Ciphertext2 = [swapbytes(uint32(hex2dec('9806f66b')))
swapbytes(uint32(hex2dec('7970fdff')))
swapbytes(uint32(hex2dec('8617187b')))
swapbytes(uint32(hex2dec('b9fffdff')))];

Ciphertext3 = [swapbytes(uint32(hex2dec('5ae4df3e')))
swapbytes(uint32(hex2dec('dbd5d35e')))
swapbytes(uint32(hex2dec('5b4f0902')))
swapbytes(uint32(hex2dec('0db03eab')))];

Ciphertext4 = [swapbytes(uint32(hex2dec('1e031dda')))
swapbytes(uint32(hex2dec('2fbe03d1')))
swapbytes(uint32(hex2dec('792170a0')))
swapbytes(uint32(hex2dec('f3009cee')))];
```

```
%% ŠIFROVÁNÍ

Counter_enc = Counter;

%ŠIFROVÁNÍ 1. BLOK
%První blok se šifruje pomocí Counter 1
Counter_enc(4) = swapbytes(Counter_enc(4))+1;
Counter_enc(4) = swapbytes(Counter_enc(4));
%Vytvorenie pointerov
Ciphertext = uint32([0;0;0;0]);
pCiphertext = libpointer('uint32Ptr', Ciphertext);
pCounter_enc = libpointer('uint32Ptr', Counter_enc);
pKey = libpointer('uint32Ptr', Key);
pPlaintext1 = libpointer('uint32Ptr', Plaintext1);

calllib('aes_gcm', 'enc_aes_gcm', pCounter_enc, pPlaintext1, pKey, pCipher-
text);

Ciphertext1_enc = get(pCiphertext, 'Value');
Ciphertext1_enc = dec2hex(swapbytes(Ciphertext1_enc()));

%ŠIFROVÁNÍ 2. BLOK
%Nový Counter
Counter_enc(4) = swapbytes(Counter_enc(4))+1;
Counter_enc(4) = swapbytes(Counter_enc(4));
%Vytvoření pointerů
Ciphertext = uint32([0;0;0;0]);
pCiphertext = libpointer('uint32Ptr', Ciphertext);
pCounter_enc = libpointer('uint32Ptr', Counter_enc);
pKey = libpointer('uint32Ptr', Key);
pPlaintext2 = libpointer('uint32Ptr', Plaintext2);

calllib('aes_gcm', 'enc_aes_gcm', pCounter_enc, pPlaintext2, pKey, pCipher-
text);

Ciphertext2_enc = get(pCiphertext, 'Value');
Ciphertext2_enc = dec2hex(swapbytes(Ciphertext2_enc()));
```

```
%ŠIFROVÁNÍ 3. BLOK
%Nový Counter
Counter_enc(4) = swapbytes(Counter_enc(4))+1;
Counter_enc(4) = swapbytes(Counter_enc(4));
%Vytvoření pointerů
Ciphertext = uint32([0;0;0;0]);
pCiphertext = libpointer('uint32Ptr', Ciphertext);
pCounter_enc = libpointer('uint32Ptr', Counter_enc);
pKey = libpointer('uint32Ptr', Key);
pPlaintext3 = libpointer('uint32Ptr', Plaintext3);

calllib('aes_gcm', 'enc_aes_gcm', pCounter_enc, pPlaintext3, pKey, pCipher-
text);

Ciphertext3_enc = get(pCiphertext, 'Value');
Ciphertext3_enc = dec2hex(swapbytes(Ciphertext3_enc()));

%ŠIFROVÁNÍ 4. BLOK
%Nový Counter
Counter_enc(4) = swapbytes(Counter_enc(4))+1;
Counter_enc(4) = swapbytes(Counter_enc(4));
%Vytvoření pointerů
Ciphertext = uint32([0;0;0;0]);
pCiphertext = libpointer('uint32Ptr', Ciphertext);
pCounter_enc = libpointer('uint32Ptr', Counter_enc);
pKey = libpointer('uint32Ptr', Key);
pPlaintext4 = libpointer('uint32Ptr', Plaintext4);

calllib('aes_gcm', 'enc_aes_gcm', pCounter_enc, pPlaintext4, pKey, pCipher-
text);

Ciphertext4_enc = get(pCiphertext, 'Value');
Ciphertext4_enc = dec2hex(swapbytes(Ciphertext4_enc()));

%% DEŠIFROVÁNÍ

Counter_dec = Counter;
```

```
%DEŠIFROVÁNÍ 1. BLOK
%Vytvoření pointerů
Plaintext_dec = uint32([0;0;0;0]);
pPlaintext_dec = libpointer('uint32Ptr', Plaintext_dec);
pCiphertext1 = libpointer('uint32Ptr', Ciphertext1);
pCounter_dec = libpointer('uint32Ptr', Counter_dec);
pKey = libpointer('uint32Ptr', Key);

calllib('aes_gcm', 'dec_aes_gcm', pCounter_dec, pCiphertext1, pKey, pPlain-
text_dec);

Plaintext1_dec = get(pPlaintext_dec, 'Value');
Plaintext1_dec = dec2hex(swapbytes(Plaintext1_dec()));

%DEŠIFROVÁNÍ 2. BLOK
%Nový Counter
Counter_dec(4) = swapbytes(Counter_dec(4))+1;
Counter_dec(4) = swapbytes(Counter_dec(4));
%Vytvoření pointerů
Plaintext_dec = uint32([0;0;0;0]);
pPlaintext_dec = libpointer('uint32Ptr', Plaintext_dec);
pCiphertext2 = libpointer('uint32Ptr', Ciphertext2);
pCounter_dec = libpointer('uint32Ptr', Counter_dec);
pKey = libpointer('uint32Ptr', Key);

calllib('aes_gcm', 'dec_aes_gcm', pCounter_dec, pCiphertext2, pKey, pPlain-
text_dec);

Plaintext2_dec = get(pPlaintext_dec, 'Value');
Plaintext2_dec = dec2hex(swapbytes(Plaintext2_dec()));

%DEŠIFROVÁNÍ 3. BLOK
%Nový Counter
Counter_dec(4) = swapbytes(Counter_dec(4))+1;
Counter_dec(4) = swapbytes(Counter_dec(4));
%Vytvoření pointerů
Plaintext_dec = uint32([0;0;0;0]);
pPlaintext_dec = libpointer('uint32Ptr', Plaintext_dec);
```

```
pCiphertext3 = libpointer('uint32Ptr', Ciphertext3);
pCounter_dec = libpointer('uint32Ptr', Counter_dec);
pKey = libpointer('uint32Ptr', Key);

calllib('aes_gcm', 'dec_aes_gcm', pCounter_dec, pCiphertext3, pKey, pPlain-
text_dec);

Plaintext3_dec = get(pPlaintext_dec, 'Value');
Plaintext3_dec = dec2hex(swapbytes(Plaintext3_dec()));

%DEŠIFROVÁNÍ 4. BLOK
%Nový Counter
Counter_dec(4) = swapbytes(Counter_dec(4))+1;
Counter_dec(4) = swapbytes(Counter_dec(4));
%Vytvoření pointerů
Plaintext_dec = uint32([0;0;0;0]);
pPlaintext_dec = libpointer('uint32Ptr', Plaintext_dec);
pCiphertext4 = libpointer('uint32Ptr', Ciphertext4);
pCounter_dec = libpointer('uint32Ptr', Counter_dec);
pKey = libpointer('uint32Ptr', Key);

calllib('aes_gcm', 'dec_aes_gcm', pCounter_dec, pCiphertext4, pKey, pPlain-
text_dec);

Plaintext4_dec = get(pPlaintext_dec, 'Value');
Plaintext4_dec = dec2hex(swapbytes(Plaintext4_dec()));

%% AUTENTIZACE

TagSender = uint32([0;0;0;0]);
TagReceiver = uint32([0;0;0;0]);
Ciphertext_auth = [Ciphertext1;Ciphertext2;Ciphertext3;Ciphertext4];

%Autentizace na straně ODESÍLATELE
pCounter = libpointer('uint32Ptr', Counter);
pCiphertext_auth = libpointer('uint32Ptr', Ciphertext_auth);
pKey = libpointer('uint32Ptr', Key);
pTagSender = libpointer('uint32Ptr', TagSender);
```

```
calllib('aes_gcm', 'auth_aes_gcm', pCounter, pCiphertext_auth, pKey, pTag-
Sender);

AuthTagSender = get(pTagSender, 'Value');
AuthTagSender = dec2hex(swapbytes(AuthTagSender()));

%Autentizace na straně PŘÍJMATELE
pCounter = libpointer('uint32Ptr', Counter);
pCiphertext_auth = libpointer('uint32Ptr', Ciphertext_auth);
pKey = libpointer('uint32Ptr', Key);
pTagReceiver = libpointer('uint32Ptr', TagReceiver);

calllib('aes_gcm', 'auth_aes_gcm', pCounter, pCiphertext_auth, pKey, pTag-
Receiver);

AuthTagReceiver = get(pTagReceiver, 'Value');
AuthTagReceiver = dec2hex(swapbytes(AuthTagReceiver()));

%% ZOBRAZENÍ VÝSLEDKŮ

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Plaintext4Blocks = ['6BC1BEE22E409F96E93D7E117393172A';
'AE2D8A571E03AC9C9EB76FAC45AF8E51';
'30C81C46A35CE411E5FBC1191A0A52EF';
'F69F2445DF4F9B17AD2B417BE66C3710']

Ciphertext1_enc = reshape(Ciphertext1_enc',1,32);
Ciphertext2_enc = reshape(Ciphertext2_enc',1,32);
Ciphertext3_enc = reshape(Ciphertext3_enc',1,32);
Ciphertext4_enc = reshape(Ciphertext4_enc',1,32);
Ciphertext4Blocks = [Ciphertext1_enc;Ciphertext2_enc;
Ciphertext3_enc;Ciphertext4_enc]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Plaintext1_dec = reshape(Plaintext1_dec',1,32);
```

```
Plaintext2_dec = reshape(Plaintext2_dec',1,32);
Plaintext3_dec = reshape(Plaintext3_dec',1,32);
Plaintext4_dec = reshape(Plaintext4_dec',1,32);
DecryptedPlaintext4Blocks = [Plaintext1_dec;Plaintext2_dec;
Plaintext3_dec;Plaintext4_dec]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
AuthTagSender = reshape(AuthTagSender',1,32)
AuthTagReceiver = reshape(AuthTagReceiver',1,32)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

if (Plaintext4Blocks) == (DecryptedPlaintext4Blocks)
disp('Šifrovaný a dešifrovaný Plaintext se zhoduje.')
else disp('Šifrovaný a dešifrovaný Plaintext se nezhoduje!')
end

if (AuthTagReceiver) == (AuthTagSender)
disp('Tag na straně odosílatele a příjmatele se zhoduje - autentizace úspěšná.')
else disp('Tag na straně odosílatele a příjmatele sa nezhoduje - autentizace neúspěšná.')
end

%% Unload DLL
unloadlibrary aes_gcm
```

B VÝPOČET VEKTOROVÝCH OPERACÍ NA PLATFORMĚ X86-64 POMOCÍ AVX A FMA

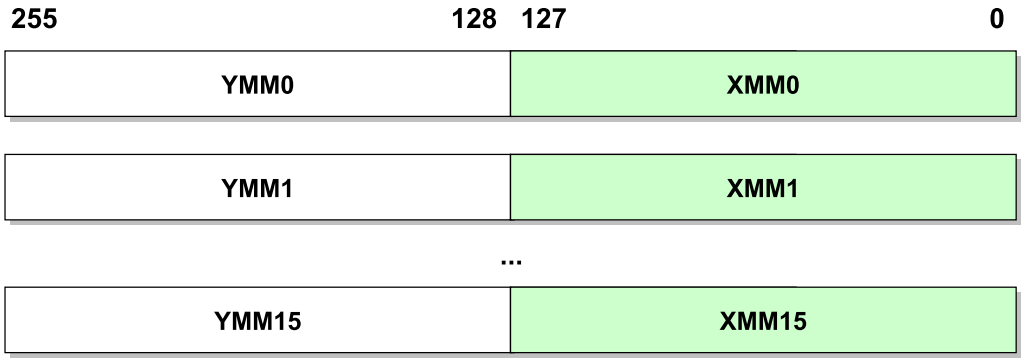
B.1 Teoretický úvod

B.1.1 Programové prostředí AVX a FMA

AVX (Advanced Vector Extensions) a FMA3 jsou instrukční sady SIMD, které přináší několik zásadních změn naproti SSE:

- AVX zavádí nové 256 bitové registry YMM0 až YMM7 (YMM0 až YMM15 v 64 bitové verzi OS), na kterých pracují dva nové datové typy.
- AVX používa nové kódovací schéma VEX-prefix (Vector Extension-prefix), díky čemu je možné pracovat s tzv. nedestruktivním operandem.
- FMA3 poskytuje soubor instrukcí, které slučují operace násobení a sčítání, nebo násobení a odečítaní, příp. jiných variant, do jednoho kroku. Největší výhodou je přesnost výpočtu.

Registry YMM se překrývají s registry XMM, které tvoří dolní polovinu šířky registrů YMM, tj. bity 0 až 127. YMM registry jsou zobrazeny na obrázku B.1. Díky



Obr. B.1: 256 bitové SIMD registry YMM

dvojnásobné velikosti registrů je tedy možné v jednom kroku pracovat s dvojnásobným počtem dat ve srovnání s výpočetní jednotkou SSE. Také je velkým přínosem dvakrát víc registrů v 64 bitovém režimu.

B.2 Vzorový příklad násobení vektoru maticí (násobení zleva)

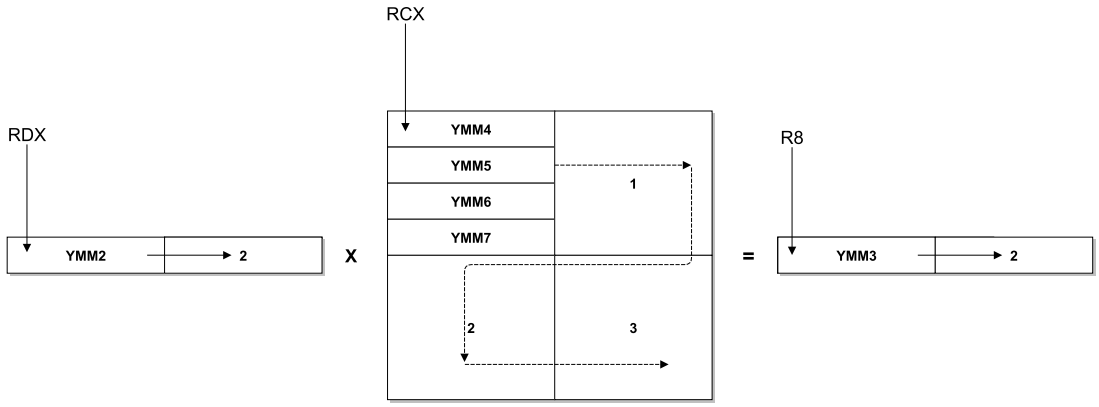
V tomto příkladu bude provedeno maticové násobení mezi vstupním vektorem a maticí s využitím instrukcí AVX a FMA, a datových typů, pracujících s číslama s plovoucí řádovou čárkou jak v jednoduché, tak v dvojité přesnosti.

Kromě nových YMM registrů budeme používat i standarní registry procesoru, přes které se budou předávat data z vnějšího programu (Matlab, registry RCX, RDX, R8, R9). Dále standarní registry použijeme na uložení hodnot, které definují posun po matici.

Následuje seznam standardních registrů s popisem jejich obsahu při výpočtu:

- RCX - ukazatel na matici.
- RDX - ukazatel na vstupní vektor.
- R8 - ukazatel na výstupní vektor.
- R9 = R10 = R11 - délka vektoru.
- R12 - hodnota posunu po matici vnitřního cyklu.
- R13 - hodnota posunu po matici zpět po ukončení vnitřního cyklu.

Při návrhu algoritmu, který bude realizovat násobení, je třeba si uvědomit, že YMM registry jsou vektorové (datové typy packed). V našem případě to znamená, že jednotlivé prvky jsou zpracovávány po osmi (jednoduchá přesnost), resp. po čtyřech (dvojitá přesnost). Maticě větších rozměrů jsou počítány pomocí submatice se zmíněným posunem po matici, což lépe vysvětluje obrázek B.2.



Obr. B.2: Maticové násobení, využití YMM registrů pro větší rozměry matic

B.2.1 Zdrojové kódy - jednoduchá přesnost

V tomto příkladu je vektor násoben maticí, kdy v jednom cyklu se pracuje se submaticí 8x8, jelikož čísla v jednoduché přenosti mají velikost 32 bitů, a tedy do jednoho YMM registru se jich vejde 8.

Výpočet pomocí AVX:

```
vmovups ymm4,[rcx]      ; submatice 8x8 do YMM4 až YMM11
add rcx,r12     ; posun ukazatele na prvky matice
vmovups ymm5,[rcx]
add rcx,r12
vmovups ymm6,[rcx]
add rcx,r12
vmovups ymm7,[rcx]
add rcx,r12
vmovups ymm8,[rcx]
add rcx,r12
vmovups ymm9,[rcx]
add rcx,r12
vmovups ymm10,[rcx]
add rcx,r12
vmovups ymm11,[rcx]

vbroadcastss ymm0,[esi]      ; 1. prvek vektoru v celém YMM0
vmulps ymm4,ymm0,ymm4      ; 1. řádek matice krát 1. prvek vektora
add esi,4      ; posun ukazatele na vst. vektor o 1 prvek

vbroadcastss ymm0,[esi]      ; 2. prvek vektoru v celém YMM0
vmulps ymm5,ymm0,ymm5      ; 2. řádek matice krát 2. prvek vektora
add esi,4      ; posun ukazatele na vst. vektor o 1 prvek

vbroadcastss ymm0,[esi]      ; 3. prvek vektoru v celém YMM0
vmulps ymm6,ymm0,ymm6      ; 3. řádek matice krát 3. prvek vektora
add esi,4      ; posun ukazatele na vst. vektor o 1 prvek

vbroadcastss ymm0,[esi]
vmulps ymm7,ymm0,ymm7
add esi,4
```

```
vbroadcastss ymm0,[esi]
vmulps ymm8,ymm0,ymm8
add esi,4

vbroadcastss ymm0,[esi]
vmulps ymm9,ymm0,ymm9
add esi,4

vbroadcastss ymm0,[esi]
vmulps ymm10,ymm0,ymm10
add esi,4

vbroadcastss ymm0,[esi]
vmulps ymm11,ymm0,ymm11
sub esi,28      ; posun ukazatele zpět na původní hodnotu

vaddps ymm1,ymm1,ymm4      ; sčítání mezivýsledů po násobení
vaddps ymm1,ymm1,ymm5      ; do výsledného subvektora v YMM1
vaddps ymm1,ymm1,ymm6
vaddps ymm1,ymm1,ymm7
vaddps ymm1,ymm1,ymm8
vaddps ymm1,ymm1,ymm9
vaddps ymm1,ymm1,ymm10
vaddps ymm1,ymm1,ymm11

vmovups [rax],ymm1      ; výslední subvektor do RAX
```

Výpočet pomocí FMA:

```
vmovups ymm4,[rcx]      ; submatice 8x8 do YMM4 až YMM11
add rcx,r12      ; posun ukazatele na prvky matice
vmovups ymm5,[rcx]
add rcx,r12
vmovups ymm6,[rcx]
add rcx,r12
vmovups ymm7,[rcx]
add rcx,r12
```

```
vmovups ymm8,[rcx]
add rcx,r12
vmovups ymm9,[rcx]
add rcx,r12
vmovups ymm10,[rcx]
add rcx,r12
vmovups ymm11,[rcx]

vbroadcastss ymm0,[esi]      ; 1. prvek vektoru v celém YMM0
vfmadd231ps ymm1,ymm0,ymm4      ; 1. řádek matice krát 1. prvek vektora a sčítání výsledku po násobení do YMM1 v jednom kroku
add esi,4      ; posun ukazatele na vst. vektor o 1 prvek

vbroadcastss ymm0,[esi]      ; 2. prvek vektoru v celém YMM0
vfmadd231ps ymm1,ymm0,ymm5      ; 2. řádek matice krát 2. prvek vektora a sčítání výsledku po násobení do YMM1 v jednom kroku
add esi,4      ; posun ukazatele na vst. vektor o 1 prvek

vbroadcastss ymm0,[esi]      ; 3. prvek vektoru v celém YMM0
vfmadd231ps ymm1,ymm0,ymm6      ; 3. řádek matice krát 3. prvek vektora a sčítání výsledku po násobení do YMM1 v jednom kroku
add esi,4      ; posun ukazatele na vst. vektor o 1 prvek

vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm7
add esi,4

vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm8
add esi,4

vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm9
add esi,4

vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm10
add esi,4
```

```
vbroadcastss ymm0,[esi]
vfmadd231ps ymm1,ymm0,ymm11
sub esi,28      ; posun ukazatele zpět na původní hodnotu

vmovups [rax],ymm1      ; výsledný subvektor do RAX
```

B.2.2 Zdrojové kódy - dvojitá přesnost

V tomto příkladu je vektor násoben maticí, kdy v jednom cyklu se pracuje se submaticí 4x4, jelikož čísla v dvojité přenosti mají velikost 64 bitů, a tedy do jednoho YMM registru se jich vejdou 4.

Výpočet pomocí AVX:

```
vmovupd ymm4,[rcx]      ; YMM4 = m4,m3,m2,m1
add rcx,r12      ; posun ukazatele na prvky matice
vmovupd ymm5,[rcx]      ; YMM5 = m8,m7,m6,m5
add rcx,r12      ; posun ukazatele na prvky matice
vmovupd ymm6,[rcx]      ; YMM6 = m12,m11,m10,m9
add rcx,r12      ; posun ukazatele na prvky matice
vmovupd ymm7,[rcx]      ; YMM7 = m16,m15,m14,m13
vmovupd ymm3,[rax]      ; výstupný vektor v YMM3

vbroadcastsd ymm1,[rsi]      ; 1. prvek vektora v celém YMM1
add rsi,8      ; posun ukazatele na vst. vektor o 1 prvek
vmulpd ymm4,ymm4,ymm1      ; 1. řádek matice krát 1. prvek vektora

vbroadcastsd ymm1,[rsi]      ; 2. prvek vektora v celém YMM1
add rsi,8      ; posun ukazatele na vst. vektor o 1 prvek
vmulpd ymm5,ymm5,ymm1      ; 2. řádek matice krát 2. prvek vektora

vbroadcastsd ymm1,[rsi]      ; 3. prvek vektora v celém YMM1
add rsi,8      ; posun ukazatele na vst. vektor o 1 prvek
vmulpd ymm6,ymm6,ymm1      ; 3. řádek matice krát 3. prvek vektora

vbroadcastsd ymm1,[rsi]      ; 4. prvek vektora v celém YMM1
sub rsi,24      ; posun ukazatele späť na pôvodnú hodnotu
vmulpd ymm7,ymm7,ymm1      ; 4. řádek matice krát 4. prvek vektora
```

```
vaddpd ymm4,ymm4,ymm5      ; sčítání mezivýsledků po násobení
vaddpd ymm6,ymm6,ymm7      ; do výsledného subvektora, který je
vaddpd ymm4,ymm4,ymm6      ; v YMM3
vaddpd ymm3,ymm3,ymm4
vmovupd [rax],ymm3         ; výsledný subvektor do RAX
```

Výpočet pomocí FMA:

```
vmovupd ymm4,[rcx]         ; YMM4 = m4,m3,m2,m1
add rcx,r12        ; posun ukazatele na prvky matice
vmovupd ymm5,[rcx]         ; YMM5 = m8,m7,m6,m5
add rcx,r12        ; posun ukazatele na prvky matice
vmovupd ymm6,[rcx]         ; YMM6 = m12,m11,m10,m9
add rcx,r12        ; posun ukazatele na prvky matice
vmovupd ymm7,[rcx]         ; YMM7 = m16,m15,m14,m13
vmovupd ymm2,[rax]         ; výstupný vektor v YMM2

vbroadcastsd ymm1,[rsi]        ; 1. prvek vektora v celém YMM1
add rsi,8        ; posun ukazatele na vst. vektor o 1 prvek
vfmadd231pd ymm2,ymm4,ymm1        ; 1. řádek matice krát 1. prvek vek-
toru a sčítání výsledku po násobení do YMM2 v jednom kroku

vbroadcastsd ymm1,[rsi]        ; 2. prvek vektoru v celém YMM1
add rsi,8        ; posun ukazatele na vst. vektor o 1 prvek
vfmadd231pd ymm2,ymm5,ymm1        ; 2. řádek matice krát 2. prvek vek-
toru a sčítání výsledku po násobení do YMM2 v jednom kroku

vbroadcastsd ymm1,[rsi]        ; 3. prvek vektoru v celém YMM1
add rsi,8        ; posun ukazatele na vst. vektor o 1 prvek
vfmadd231pd ymm2,ymm6,ymm1        ; 3. řádek matice krát 3. prvek vek-
toru a sčítání výsledku po násobení do YMM2 v jednom kroku

vbroadcastsd ymm1,[rsi]        ; 4. prvek vektoru v celém YMM1
sub rsi,24        ; posun ukazatele späť na pôvodnú hodnotu
vfmadd231pd ymm2,ymm7,ymm1        ; 4. řádek matice krát 4. prvek vek-
toru a sčítání výsledku po násobení do YMM2 v jednom kroku
```

```
vmovupd [rax],ymm2        ; výsledný subvektor do RAX
```

Po linkování a kompilování kódu do dynamické knihovny využijeme program Matlab pro otestování správnosti výpočtu a pro zjištění doby jednotlivých výpočtů. V Matlabu vytvoříme skript, který bude volat funkce z dynamické knihovny. Jako vzorový příklad využijeme diskrétní kosinovou transformaci (DCT).

Skript v Matlabu - jednoduchá přesnost:

```
% Uzavření všech grafů a smazání všech proměnných
clear all; clc

f1=1000;
f2=1100;
fs=8000;

N=33000;
n=0:N-1;
k=0:N-1;

% definice signalu v jednoduche presnosti
x=single(0.7*cos(2*pi*f1/fs*(0:N-1))+0.3*cos(2*pi*f2/fs*(0:N-1)));
%výpočet DCT verze IV; definice v jednoduche presnosti
M=single(cos(pi/N*(n'+1/2)*(k+1/2)));

disp('Výpočet v Matlabu:')
tic
x*M;
toc
y=x*M;

%počet volání funkce
Np=5;

%vykresleni koeficientu DCT po normalizaci
plot(y * sqrt(2/N),'r') %normalizace DCT
hold on

%nacteni DLL
hfile1 = 'kody64.h';
```

```
[notfound,warnings]=loadlibrary('kody64.dll', hfile1,'mfilename','kody64_mx');
pM = libpointer('singlePtr', M);
px = libpointer('singlePtr', x);

%% AVX
%Zahrivaci vypocet vxm_avx_sp
y0=zeros(1,N); py0 = libpointer('singlePtr', y0);
calllib('kody64', 'vxm_avx_sp', pM, px, py0, int64(length(x)/8));
disp('Zahřívací výpočet AVX ukončen')

%volani funkce vxm_avx_sp
disp('Výpočet DCT v ASM pomocí AVX:')
td=0;
for i=1:Np
y0=zeros(1,N); py0 = libpointer('singlePtr', y0);
tic;
calllib('kody64', 'vxm_avx_sp', pM, px, py0, int64(length(x)/8));
td=td+toc/Np;
end
disp(num2str(td,6))
y_avx=get(py0, 'Value');
plot(y_avx * sqrt(2/N),'y')

%% FMA
%Zahrivaci vypocet vxm_fma_sp
y0=zeros(1,N); py0 = libpointer('singlePtr', y0);
calllib('kody64', 'vxm_fma_sp', pM, px, py0, int64(length(x)/8));
disp('Zahřívací výpočet FMA ukončen')

%volani funkce vxm_fma_sp
disp('Výpočet DCT v ASM pomocí FMA:')
td=0;
for i=1:Np
y0=zeros(1,N); py0 = libpointer('singlePtr', y0);
tic;
calllib('kody64', 'vxm_fma_sp', pM, px, py0, int64(length(x)/8));
td=td+toc/Np;
end
```

```
disp(num2str(td,6))
y_fma=get(py0, 'Value');
plot(y_fma * sqrt(2/N),'m')

%%
hold off
unloadlibrary kody64
legend('DCT IV jako maticové násobení v Matlabu','DCT IV pomocí funkce VXM AVX SP','DCT IV pomocí funkce VXM FMA SP')
```

Skript v Matlabu - dvojitá přesnost:

```
% Uzavření všech grafů a smazání všech proměnných
clear all; clc

f1=1000;
f2=1100;
fs=8000;

N=33000;
n=0:N-1;
k=0:N-1;

% definice signalu v dvojite presnosti
x=double(0.7*cos(2*pi*f1/fs*(0:N-1))+0.3*cos(2*pi*f2/fs*(0:N-1)));
%výpočet DCT verze IV; definice v dvojite presnosti
M=double(cos(pi/N*(n'+1/2)*(k+1/2)));

disp('Výpočet v Matlabu:')
tic
x*M;
toc
y=x*M;

%počet volání funkce
Np=5;

%vykresleni koeficientu DCT po normalizaci
plot(y * sqrt(2/N),'r') %normalizace DCT
```

```
hold on

%nacteni DLL
hfile1 = 'kody64.h';
[notfound,warnings]=loadlibrary('kody64.dll', hfile1,'mfilename','kody64_mx');
pM = libpointer('doublePtr', M);
px = libpointer('doublePtr', x);

%% AVX
%Zahrivaci vypocet vxm_avx_dp
y0=zeros(1,N); py0 = libpointer('doublePtr', y0);
calllib('kody64', 'vxm_avx_dp', pM, px, py0, int64(length(x)/4));
disp('Zahřívací výpočet AVX ukončen')

%volani funkce vxm_avx_dp
disp('Výpočet DCT v ASM pomocí AVX:')
td=0;
for i=1:Np
y0=zeros(1,N); py0 = libpointer('doublePtr', y0);
tic;
calllib('kody64', 'vxm_avx_dp', pM, px, py0, int64(length(x)/4));
td=td+toc/Np;
end
disp(num2str(td,6))
y_avx=get(py0, 'Value');
plot(y_avx * sqrt(2/N),'y')

%% FMA
%Zahrivaci vypocet vxm_fma_dp
y0=zeros(1,N); py0 = libpointer('doublePtr', y0);
calllib('kody64', 'vxm_fma_dp', pM, px, py0, int64(length(x)/4));
disp('Zahřívací výpočet FMA ukončen')

%volani funkce vxm_fma_dp
disp('Výpočet DCT v ASM pomocí FMA:')
td=0;
for i=1:Np
y0=zeros(1,N); py0 = libpointer('doublePtr', y0);
```

```
tic;
calllib('kody64', 'vxm_fma_dp', pM, px, py0, int64(length(x)/4));
td=td+toc/Np;
end
disp(num2str(td,6))
y_fma=get(py0, 'Value');
plot(y_fma * sqrt(2/N),'m')

%%
hold off
unloadlibrary kody64
legend('DCT IV jako maticové násobení v Matlabu','DCT IV pomocí funkce
VXM AVX DP','DCT IV pomocí funkce VXM FMA DP')
```

C ZOZNAM INŠTRUKCIÍ PRE PODPORU KRYPTOGRAFICKÝCH

Táto príloha poskytuje zoznam postupne pridaných inštrukcií pre podporu kryptografických výpočtov v jednotlivých inštrukčných sadách.

C.1 Inštrukcie pre podporu blokovej šifry AES

AES-NI (Advanced Encryption Standard - New Instruction) sada inštrukcií je veľkým prínosom z programátorského hľadiska. Jedinou inštrukciou sa totiž vykoná sled transformácií jednej iterácie blokovej šifry AES, ktorý by inak vyžadoval veľké množstvo operácií a riadkov zdrojového kódu.

Tab. C.1: Prehľad AES-NI inštrukcií

Inštrukcia	**Popis**
AESENC xmm1,xmm2/m128	Vykoná jednu iteráciu (round) šifrovania 128 bitov dát v xmm1 so 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESENCLAST xmm1,xmm2/m128	Vykoná poslednú iteráciu (round) šifrovania 128 bitov dát v xmm1 so 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESDEC xmm1,xmm2/m128	Vykoná jednu iteráciu (round) dešifrovania 128 bitov dát v xmm1 so 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESDECLAST xmm1,xmm2/m128	Vykoná poslednú iteráciu (round) dešifrovania 128 bitov dát v xmm1 so 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESIMC xmm1,xmm2/m128	Vykoná transformáciu InvMixColumn nad 128 bitovým iteračným kľúčom (round key) v xmm2/m128 a výsledok uloží do xmm1.
AESKEYGENASSIST xmm1, xmm2/m128,imm8	Asistuje pri generovaní AES iteračného kľúča (round key) s využitím 8 bitovej iteračnej konštanty (round constant) špecifikovanej v maske imm8. Inštrukcia pracuje nad dátami v xmm2/m128 a výsledok uloží do xmm1.

C.2 Inštrukcia pre podporu násobenia bez prenosu

Inštrukcia násobenia bez prenosu - Carry-Less Multiplication - je veľkým prínosom v určitých kryptografických počtoch, najmä v prípade operačného módu GCM (Galois Counter Mode) blokovej šifry AES. GCM používa násobenie bez prenosu vo svojom algoritme násobenia v Galoisovom poli $GF(2^{128})$.

Tab. C.2: Zápis a popis inštrukcie násobenia bez prenosu

Inštrukcia	**Popis**
PCLMULQDQ xmm1, xmm2/m128,imm8	Inštrukcia vynásobí jeden dátový typ quadword (8 bytov) v xmm1 s jedným quadwordom v xmm2/m128 bez prenosu. Výsledkom je dátový typ double quadword (16 bytov) uložený v xmm1. Operand imm8 je maska, ktorá definuje, ktorý quadword xmm1 a xmm2/m128 sa má použiť.