VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

HARDWAROVÁ AKCELERACE ŠIFROVACÍCH ALGORITMŮ S TECHNOLOGIÍ XILINX ZYNQ

HARDWARE ACCELERATION OF ENCRYPTION ALGORITHMS USING XILINX ZYNQ TECHNOLOGY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MAREK LINNER**
AUTHOR

VEDOUCÍ PRÁCE — **Doc. Ing. JAN KOŘENEK, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Linner Marek**
Program: Informační technologie
Název: **Hardwarová akcelerace šifrovacích algoritmů s technologií Xilinx Zynq**
Hardware Acceleration of Encryption Algorithms Using Xilinx Zynq Technology
Kategorie: Počítačová architektura

Zadání:

1. Seznamte se s jazykem VHDL a algoritmy DES a AES pro šifrování a dešifrování dat.
2. Dále nastudujte možnosti a systém práce s přípravkem ZynqBerry.
3. Navrhněte vhodný způsob hardwarové akcelerace algoritmů DES a AES v FPGA logice na technologii Xilinx Zynq. Zaměřte se na vhodné rozdělení úlohy mezi softwarové a hardwarové prostředky.
4. Implementujte navržený způsob hardwarové akcelerace. Zjistěte hlavní parametry vytvořené implementace.
5. Diskutujte vlastnosti navrženého řešení a urychlení proti softwarovému zpracování.

Literatura:

- Dle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Kořenek Jan, doc. Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Abstrakt

Tato práce se zabývá dvěma standardy pro zabezpečení uložených dat a datových toků *Data Encryption Standard* (dále zkratka *DES*) a *Advanced Encryption Standard* (dále zkratka *AES*). Prezentuje srovnání několika volně dostupných implementací pro oba algoritmy, které potom integruje do měřících zdrojových kódů v jazyce C. Měřící program potom změří rychlost provádění algoritmu a vypočítá bitovou rychlost pro různé délky vstupního bloku zprávy. Dále je součástí práce implementace obou šifrovacích algoritmů DES i AES v jazyce VHDL, simulace syntetizovaných návrhů a odvození bitové rychlosti obvodů pomocí výpisů z Vivado simulátoru. Výsledné vypočítané rychlosti hardwarových implementací jsou poté srovnány s naměřenou rychlostí softwarových implementací. Součástí práce jsou i zdrojové soubory kódu užitého při měření v jazyce C, kód VHDL implementace, program v jazyce C# pro generování VHDL komponent a další program v jazyce C# užitý při automatickém testování.

Abstract

The main concern of this paper are two world standard encryption algorithms *Data Encryption Standard DES* (*DES* for short) and *Advanced Encryption Standard* (further mentioned as *AES*). For these two respective algorithms, three publicly available implementations are integrated into a benchmarking code in C programming language. The code has been executed, implementations measured with three different input block lengths and *bitrate* calculated for each implementation. The thesis also includes hardware implementation of both encryption algorithms DES and AES using VHDL language, simulation of the synthesised circuits and calculation of the hardware implementations' bitrate using Vivado simulator's timing reports. These measured bitrates are then compared with the bitrates of benchmarked software implementations. Paper includes all source codes of the benchmarking C program and VHDL implementation, along with program written in C# used to generate VHDL components and another C# program used for automated testing.

Klíčová slova

Advanced Encryption Standard, Data Encryption Standard, AES, DES, hardwarová akcelerace, C, Xilinx Zynq, OpenSSL

Keywords

Advanced Encryption Standard, Data Encryption Standard, AES, DES, hardware acceleration, C, Xilinx Zynq, OpenSSL

Citace

LINNER, Marek. *Hardwarová akcelerace šifrovacích algoritmů s technologií Xilinx Zynq*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. Ing. Jan Kořenek, Ph.D.

Hardwarová akcelerace šifrovacích algoritmů s technologií Xilinx Zynq

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Doc. Ing. Jana Kořenka, Ph.D.. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Marek Linner
11. května 2021

Poděkování

Především bych chtěl poděkovat Doc. Ing. Janu Kořenkovi, Ph. D. za jeho trpělivost, čas a energii strávenými nad technickou zprávou a velmi cennou zpětnou vazbu, která ve velké míře přispěla ke vzniku této bakalářské práce.

Obsah

1 Úvod **3**

2 Kryptografické šifry **6**
2.1 Symetrické šifry 6
2.2 Asymetrické šifry 7
2.3 Proudové šifry 7
2.4 Blokové šifry 7
2.4.1 Výplň 8
2.5 Režimy provozu 8

3 Algoritmus DES **10**
3.1 Princip 10
3.2 Analyzované implementace 17
3.2.1 Implementace z Rosetta Code 17
3.2.2 Implementace z Programming Algorithms 17
3.2.3 Použití knihovny OpenSSL 18
3.3 Výsledky měření 18

4 Algoritmus AES **20**
4.1 Princip 20
4.2 Analyzované implementace 27
4.2.1 Tiny AES 27
4.2.2 Použití knihovny OpenSSL 28
4.2.3 Luo Peng AES 128 28
4.3 Výsledky měření 28

5 Hardwarová implementace DES **30**
5.1 Implementace šifrování 30
5.2 Implementace dešifrování 32

6 Hardwarová implementace AES **35**
6.1 Implementace šifrování 35
6.2 Implementace dešifrování 37

7 Výsledky **39**
7.1 Implementace DES 39
7.2 Implementace AES 40
7.3 Testování 40

8 Závěr **41**

Literatura **42**

A Obsah přiloženého paměťového média **44**

B Manuál **47**

C DES S-boxy funkce *f* **48**

D Výpis časování DES šifrování **49**

E Výpis využití DES šifrování **55**

F Výpis časování DES dešifrování **59**

G Výpis využití DES dešifrování **63**

H Výpis časování AES šifrování **67**

I Výpis využití AES šifrování **72**

J Výpis časování AES dešifrování **76**

K Výpis využití AES dešifrování **80**

Kapitola 1

Úvod

S postupným pronikáním informačních technologií do běžného i profesního života stoupal i objem dat, se kterými lidé pracovali. Firmy, které často počítače používaly pro práci s účetnictvím, evidenci tržeb nebo jako přehled o stavu skladu, chtěly tyto citlivé informace ochránit před zneužitím. Podobně vláda USA také nestála o slabiny ve své komunikaci a uchování tajných informací. Jako jedno z řešení se nabízelo šifrování dat.

Tato potřeba přinutila tehdejší americký *National Bureau of Standards* přijít s šifrovacím algoritmem, který by byl schopen ochránit citlivá data a stal se obecně využívaným standardem. Proto 15. května roku 1973 vyhlásil konkurz, který by měl vybrat z navrhovaných kandidátů na tento algoritmus. Vítěz byl vybrán podle několika kritérií, například rychlost, s jakou je schopen zašifrovat a dešifrovat určitý objem dat, množství místa v paměti které využívá, možnost paralelizace nebo užití hardwarové akcelerace. Z všech přihlášených možností však ani jedna nesplňovala tyto náročné požadavky. Proto 27. 8. V roce 1974 byl nucen NBS vyhlásit 2. konkurz s pozměněnými kritérii. Do tohoto konkurzu již firma *International Business Machines Corporation* (známá pod zkratkou *IBM*) přihlásila vhodného, a později i vítězného kandidáta Data Encryption Standard. Algoritmus vycházel ze staršího algoritmu *Lucifer cipher*[11], vytvořeného *Horstem Feistelem.* Horst spolu se svým týmem Lucipher cipher dále rozvinuli, jejich verze byla ještě upravena americkou *National Security Agency NSA (česky národní bezpečnostní agentura)*, což vedlo k debatám o úmyslném oslabení a špatném vlivu na bezpečnost šifry. I přesto byla tato verze v listopadu roku 1976 schválena a prohlášena standardem pro šifrování dat *Data Encryption Standard DES.*

Šifrovací algoritmus úspěšně fungoval několik let. Během toho vzniklo několik jeho dalších verzí a modifikací, např. *3DES*[1], který byl hojně využíván místo samotného DES v době, kdy kvůli neustále se zvyšující výpočetní síle strojů DES už nebyl bezpečný. Ačkoliv to v době jeho návrhu a schválení prakticky nebylo možné, se strmým nárůstem výpočetní síly se čím dál více projevoval jeho nedostatek: DES není připraven na útoky hrubou silou. Během této doby navíc další firmy začaly využívat informačních technologií a po sítích cestovaly velké objemy citlivých dat. V roce 1997 byla poprvé dešifrována zpráva a tím DES překonán. Roku 1998 se povedlo šifru prolomit za 56 hodin, o rok poději už jen za 22 hodin. Bylo jasné, že je potřeba nový standard.

Proto 2. ledna roku 1997 vyhlásil americký *National Institute of Standards and Technology* (česky *Národní institut standardů a technologií*, zkratka *NIST*) nový konkurz, jehož vítěz se stane následníkem zastaralého algoritmu DES. Důležitou změnou oproti kritériím

[1] *Triple DES*, algoritmus 3DES aplikuje šifru DES na vstup třikrát.

výběru DES bylo to, že nový *Advanced Encryption Standard AES (česky standard pokročilého šifrování)* musí být schopen pracovat s klíči délky 128 bitů, 192 bitů a 256 bitů. Oproti 64 bitů dlouhému klíči DES je užití delších klíčů mnohonásobně odolnější vůči útokům hrubou silou. Z 15 kandidátu byla vybrána dne 2. října 2000 šifra s názvem *Rijndael*, jejíž název je přesmyčka z příjmení autorů Vincenta Rijmena a Joana Daemena. Tím začal asi rok trvající proces dalšího zkoumání a úprav, dokud se 26. listopadu 2001 nestal standardem *Federal Information Processing Standards (FIPS, česky Federální norma pro zpracování informací)*. Během procesu výběru standardu institut NIST pracoval transparentně a spolupracoval s kryptografickou komunitou, narozdíl od procesu výběru DES, což účastníci i výzkumníci notně ocenili. Za příkladný postup výběru nového standardu a samotnou volbu sklidil institut mnoho chvály.

Během let vzniklo několik různých útoků zneužívající různé slabiny nového AES, avšak žádný z těchto útoků není v praxi proveditelný. Pokud by i tak někdo chtěl co nejlépe zabezpečit svá data, kromě použití 256 bitového klíče může nastavit některý z režimů provozu algoritmu. Tato flexibilita a velmi vysoká odolnost vůči útokům hrubou silou zajistila šifře Rijndael pozici šifrovacího standardu na řadu let dopředu.

Protože oba algoritmy jsou využívány nejen pro uchování dat, ale i pro šifrování datových toků po síti, je jedním z důležitých parametrů bitová rychlost. Softwarové implementace algoritmů sice jsou schopny dosáhnout použitelných rychlostí, avšak vzhledem k množství využití algoritmů a dopadu zvýšení efektivity jeho chodu je mnohem výhodnější využít hardwarové akcelerace a výpočet provádět pomocí dedikovaného obvodu. Takový obvod je schopen dosáhnout několikrát vyšší bitové rychlosti s menším vytížením procesoru a nižšími nároky na energii. Díky těmto příznivým vlastnostem se instrukce algoritmu AES staly součástí instrukční sady mnoha moderních procesorů.

Pro zvýšení bitové rychlosti byl zvolen přípravek *"ZynqBerry"Module with Xilinx Zynq-7010 in Raspberry Pi Form Factor*[2]. Jedná se o systém na čipu (anglicky *system on a chip*), který spolu s vývojovým prostředím Xilinx Vivado, flexibilními FPGA hradly, pamětí RAM, procesorovou jednotkou, prostorem pro micro SD kartu, sběrnicí AXI a širokým výběrem portů rozhraní poskytují ideální prostor a zdroje pro tvorbu a testování jak nového hardwarového návrhu popsaného jazykem VHDL či Verilog, tak implementaci a ladění softwaru schopného jeho prostředky využívat.

Cílem práce je navrhnout a implementovat hardware, který je schopen šifrovat a dešifrovat zprávy pomocí algoritmů DES a AES. Vlastnosti implementací, jakými jsou využití zdrojů přípravku či bitová rychlost algoritmu, jsou zjištěny výpočtem vycházejících z výpisů časování Vivado simulátoru. Pro odvození bitové rychlosti jsou využity výpisy časování (anglicky *timing report*) zaznamenávající dobu, kterou trvá nejdelší cesta mezi komponenty v návrhu, nároky na zdroje přípravku je schopen zaznamenat výpis využití (anglicky *utilization report*). Pro získání lepšího přehledu je také součástí výběr množství volně dostupných softwarových implementací algoritmů v jazyce C, výběr tří z nich pro každý algoritmus a následně je integrovat do měřicího kódu. Měřicí kód v jazyce C má za úkol změřit bitovou rychlost tří implementací DES s délkou tajného klíče 64 bitů a dalších tří implementací AES s délkou klíče 128 bitů. Dále je všech šest implementací měřeno se vstupními bloky různé délky 128 bitů, 512 bitů a 2048 bitů. Výsledné naměřené bitové rychlosti, spolu s dalšími kritérii, jsou poté zapsány do tabulky a diskutováno jejich srovnání, výhody a nevýhody.

V kapitole 2 se nachází obecný přehled kryptografických šifer. Zde jsou popsány jejich vlastnosti, dělení, principy, využití i historie. V sekci 2.5 jsou rozebrány a vysvětleny režimy

[2]Dostupný na webové adrese výrobce https://shop.trenz-electronic.de/en/TE0726-03M-ZynqBerry-Module-with-Xilinx-Zynq-7010-in-Raspberry-Pi-Form-Factor?path=

provozu, ve kterých jsou algoritmy DES i AES schopny operovat. Kapitola 3 blíže pojednává o algoritmu DES, jeho principu, výhodách a nevýhodách. Dále obsahuje popis a srovnání tří vybraných volně dostupných softwarových implementací jak z hlediska bitové rychlosti, tak z hlediska uchopitelnosti a přehlednosti kódu. Obdobně kapitola 4 prozkoumá a blíže popíše, jak funguje v současnosti nejpoužívanější šifrovací algoritmus Rijndael. Také vybere a změří tři dostupné softwarové implementace a porovná jejich bitovou rychlost, přehlednost a uchopitelnost. V následující kapitole 5 se nachází rozbor nově vzniklého hardwarového návrhu algoritmu DES a popis, jak byl návrh implementován. Podobně se v kapitole 6 lze dočíst o postupu implementace a návrhu komponent šifrující a dešifrující zprávy algoritmem AES. V následující části 7 se nachází výsledné parametry nových implementací, jako jsou jejich rychlost, využití zdrojů a porovnání s měřenými softwarovými řešeními. V závěrečné kapitole 8 se nachází hodnocení výstupů práce.

Kapitola 2

Kryptografické šifry

Potřeba lidí utajovat informace nezačala až s příchodem počítačů. Zmínky o utajování obsahu písma se objevovaly již ve starém Egyptě, Mezopotámii a Indii. Staří Řekové vynalezli nejen mechanické ukrytí utajovaných zpráv (např. překrytí zprávy vyryté do dřevěné destičky voskem), ale v 5. století př. n. l. i první jednoduché šifrové systémy. Do historie nejen kryptografie se zapsal i Julius Caesar vynalezením šifry, která byla pojmenována jako Caesarova šifra[2].

Šifrování se velice rychle rozvíjelo a brzy vznikaly i velmi odolné šifry, které se podařilo prolomit až s použitím moderní technologie. Právě nástup počítačů v 1.polovině 20.století je milníkem v historii kryptografie, který dělí kryptografii na klasickou a moderní. Klasická kryptografie se vyznačovala tím, že k šifrování a dešifrování stačila pouze tužka a papír, případně jiné jednoduché pomůcky. Postupem času ale začaly vznikat různé sofistikovanější nástroje, které umožňovaly složitější postup při šifrování. Tím přibližně začala druhá část, která se nazývá moderní kryptografie. Další zlom v kryptografii se předpokládá s nástupem kvantových počítačů, které by svou výpočetní silou byly schopny prolomit většinu šifer. Problém ale už je postupně řešen pomocí kvantové kryptografie. Moderní šifry, o kterých je tato práce, lze dělit podle několika kritérií. Například na symetrické a asymetrické, blokové a proudové. Charakteristiku jednotlivých kategorií rozvádí následujících sekce, které čerpají z knihy Williama Stallingse s názvem *Cryptography and Network Security: Principles and Practice*[11].

2.1 Symetrické šifry

Podle knihy *Introduction to Cryptography: Principles and Applications*[1], jejíž autory jsou Hans Delfs a Helmut Knebl, jsou symetrické šifry podmnožinou kryptografických metod, které používají pro šifrování i dešifrování stejný tajný klíč, čímž zajišťují *důvěrnost* (znemožnění četní zprávy nepovolanou osobou). Tím se liší od šifer asymetrických, které ke svému fungování používají páry klíčů, veřejných a tajný. Nevýhodou symetrických šifer je skutečnost, že jak odesílatel, tak příjemce, musí znát společný klíč. Pokud by útočník získal přístup k symetrickému klíči, jednoduše může odposlouchávat odesílaná data. Na druhou stranu ale symetrické šifry mají jednu velkou výhodu. Jsou mnohonásobně rychlejší, než asymetrické. V praxi se symetrické a asymetrické šifry kombinují tak, že odesílaná zpráva se nejprve zašifruje pomocí vygenerovaného symetrického klíče. Tento klíč je potom zabezpečen pomocí asymetrického šifrování a odeslán po síti. Rozlišujeme symetrické šifry blokové (viz 2.4) a proudové (viz 2.3).

2.2 Asymetrické šifry

Asymetrické kryptografické šifry tvoří skupinu kryptografických metod, ve kterých se pro šifrování a dešifrování používají odlišné klíče, což je zásadní rozdíl oproti symetrickým šifrám. Ty používají k šifrování i dešifrování stejný klíč, což zajišťuje pouze důvěrnost zprávy. Asymetrická kryptografie je k tomu ještě schopna zajistit nejen důvěrnost, ale také *autenticitu* (jednoznačná identifikace autora zprávy) a *nepopiratelnost* (danou zprávu mohl napsat jen vlastník tajného klíče). Obecně se klíč pro asymetrickou kryptografii sestává z dvou částí: jedna část se používá pro šifrování zpráv, druhá pro dešifrování. Skutečnost, že odesilatel nemusí znát dešifrovací klíč a příjemce ten šifrovací eliminuje potřebu sdílení klíče, jako tomu je u symetrických šifer. Tato důležitá vlastnost asymetrických šifer úplně vylučuje množinu typů útoků na zašifrovaný proud dat. Nejčastěji se setkáváme s šiframi využívající jednoho páru klíčů: soukromého a veřejného. Veřejný klíč může jeho majitel volně sdílet s ostatními komunikujícími, jenž jsou schopni pomocí tohoto klíče zaslat zašifrovanou zprávu, kterou si může dešifrovat pouze majitel soukromého klíče. Velmi často jsou využívány asymetrické šifry k zajištění bezpečného přenosu symetrického klíče. K tomu se používají algoritmy jako RSA, *Digital Signature Algorithm* nebo *Elliptic Curve Digital Signature Algorithm* (zkratka *ECDSA*), které jsou blíže specifikovány v publikaci *FIPS PUB 186-4* [10].

2.3 Proudové šifry

Jeden z hlavních rozdílů mezi blokovými a proudovými šiframi je ten, že šifry proudové kombinují vstupní datový tok bit po bitu s pseudonáhodným *proudem hesla (anglicky keystream)*, který je generován pomocí klíče. Důležitou odlišností od blokových šifer je skutečnost, že proudové šifry aplikují na svůj vstupní tok transformace, které se s časem neustále mění v závislosti na současném stavu. Stav se potom může odvíjet od předchozího stavu, předchozího bitu nebo od dalších proměnných daného algoritmu. Oproti tomu blokové šifry používají konstantní transformace pro každý blok vstupu. Výstupem proudové šifry je zašifrovaný datový tok. Obecně jsou proudové šifry rychlejší než blokové a ke své implementaci potřebují jednodušší hardware. Jejich nevýhodou je však slabost proti kryptoanalytickým útokům, pokud je šifra nevhodně implementována.

2.4 Blokové šifry

Blokové šifrovací algoritmy jsou podmnožinou symetrických šifer, u nichž je vstup vždy na začátku rozdělen na bloky stejné délky. Pokud poslední blok vstupní zprávy není plný, pak je využita některá z metod přidání *výplně (anglicky padding)*. Různé varianty vkládání výplně jsou popsány v následující části 2.4.1. Blokové šifry lze obecně považovat za pseudonáhodné permutace, což znamená, že všechna celá čísla reprezentovatelná v jednom bloku převádí na jiné číslo v daném rozsahu. Důležitou vlastností blokových šifer je to, že na zašifrovanou hodnotu lze aplikovat inverzní permutaci, jejíž výstupem bude opět původní hodnota. To výrazně zjednodušuje jak softwarovou, tak hardwarovou implementaci blokových šifer. Jejich vlastní průběh jde většinou dále upravit užitím některých z *režimů provozu*. V některých režimech můžeme i původně blokovou šifru převést na proudovou, nebo modifikovat algoritmus tak, aby místo pouze jednoho vstupního bloku bral v potaz i

bloky předcházející, což výrazně zvyšuje sílu šifry. O některých vybraných režimech provozu pojednává podkapitola 2.5.

2.4.1 Výplň

Protože blokové šifry, jsou schopny pracovat pouze s bloky pevné délky, šifrovací algoritmus musí řešit i případ, kdy vstupní blok má bitů méně a musí se vyplnit nějakou výplní. Způsobů, jak doplnit délku existuje řada a tak jsou zde zmíněny jen ty nejpoužívanější.

Cryptographic Message Syntax metoda, známá také pod zkratkou CMS, doplní na volné pozice vstupního bloku M bajty s takovou hodnotu, kolik bajtů bylo doplněno do výsledného bloku M_{cms}.

Bit metoda nejprve přidá do výstupního bloku M_{bit} jeden bajt s hodnotou $0x80$, každý další přidaný bajt pak už má hodnotu $0x00$. (*null byte*).

ZeroLength metoda volné bajty nahradí hodnotou $0x00$ kromě posledního bajtu. Poslední bajt výstupního bloku M_{zl} má hodnotu takovou, kolik bajtů dohromady bylo doplněno.

Null metoda doplní na místa chybějících bajtů výstupního bloku M_{null} samé nuly. Metoda Null se používá pouze v případě, že pracuje s ASCII textem.

Space metoda vyplní blok M_{space} znaky pro mezeru, hexadecimálně $0x20$. Opět lze použít pouze při práci s ASCII textem.

Random metoda nahradí chybějící bajty náhodnými hodnotami, kromě posledního bajtu. Poslední bajt výsledného bloku M_{random}, podobně jako u metod ZeroLength a Cryptographic Message Syntax, nese hodnotu odpovídající počtu doplněných bajtů.

Následující příklad demonstruje jejich použití. Původní vstupní blok M, kterému chybí 4 bajty, je zmíněnými šesti metodami vyplňován a výstupní bloky vyplňovací metody zaznamenávány.

$$
\begin{aligned}
M &= 0x12\ 34\ 56\ 78 \\
M_{cms} &= 0x12\ 34\ 56\ 78\ 04\ 04\ 04\ 04 \\
M_{bit} &= 0x12\ 34\ 56\ 78\ 80\ 00\ 00\ 00 \\
M_{zl} &= 0x12\ 34\ 56\ 78\ 00\ 00\ 00\ 04 \\
M_{null} &= 0x12\ 34\ 56\ 78\ 00\ 00\ 00\ 00 \\
M_{space} &= 0x12\ 34\ 56\ 78\ 20\ 20\ 20\ 20 \\
M_{random} &= 0x12\ 34\ 56\ 78\ 2d\ 15\ b0\ 55
\end{aligned}
$$

2.5 Režimy provozu

Probírané algoritmy AES i DES podporují několik režimů provozu (anglicky *modes of operation*), které mohou podstatně zvýšit jejich bezpečnost, ale i výpočetní náročnost[6, 9]. Podporovaných režimů je celkem 5, z nichž má každý své výhody a nevýhody.

- **ECB - režim kódové knihy**
  Anglicky *electronic codebook* je základní a nejjednodušší režim provozu. Algoritmy DES i AES v tomto režimu aplikují na vstupní blok svůj postup nezávisle na ostatních

blocích, tedy při použití stejného klíče odpovídá stejnému bloku otevřeného textu stejný blok šifrového textu. I když je jeho výhodou skutečnost, že je odolný proti chybovosti díky nezávislosti správných ostatních bloků na chybném bloku, v dnešní kryptografii se nepoužívá kvůli nízké bezpečnosti.

- **CBC – řetězení šifrových bloků**
  V tomto režimu provozu, také známým pod jménem *cipher block chaining*, má každý blok přímý vliv na následující blok. Na každý blok je totiž před samotným šifrováním aplikována operace XOR. Na úplně první blok je použit inicializační vektor. To výrazně zvýší sílu šifry, avšak 2 nevýhody plynou z užívání zřetězené závislosti. Proces šifrování nelze paralelizovat. Paralelní zpracování je možné použít pouze pro dešifrování vstupních dat. Druhou nevýhodou je špatná obnova dat v případě, kdy nastane chyba. Pokud dojde k poškození některého z bloků dat, nejdou dešifrovat ani všichni jeho následníci.

- **CFB – šifrová zpětná vazba**
  Režim CFB (anglicky *cipher feedback*) je velmi podobný režimu CBC, pouze změní pořadí operací. Znamená to, že nejprve se provede šifrování, až poté mezivýsledek vstoupí spolu s předcházejícím zašifrovaným blokem do funkce XOR. Toto má výrazný implementační dopad: dešifrování probíhá obdobně jako šifrování, což eliminuje nutnost udržování dvou různých hardwarových implementací pro šifrování a pro dešifrování. Trpí ale stejnými nevýhodami jako režim CBC. Pokud je blok poškozen, ani žádný následující blok nelze dešifrovat, k tomu nelze paralelizovat proces šifrování. Proces dešifrování ale už může být parelelizován.

- **OFB – výstupní zpětná vazba**
  Tento režim, anglicky nazývaný *output feedback*, převádí původně blokovou šifru na šifru proudovou. V každém kroku je heslo zašifrováno použitou blokovou šifrou, první šifrování hesla potom vychází z inicializačního vektoru. Výstupem je výsledek operace XOR nad otevřenými daty a heslem.

- **CTR – čítačový režim**
  Obdobně jako režim OFB, i režim CTR (anglicky *counter mode*) převádí blokovou šifru na proudovou. Heslo, se kterým se otevřený blok podrobí operaci XOR, vzniká zašifrováním čítače blokovou cifrou. Před samotným šifrováním je obsah čítače nastaven inicializačním vektorem, po každém použití se obsah čítače zvětšuje o pevně danou hodnotu, většinu o 1.

Kapitola 3

Algoritmus DES

Data Encryption Standard je šifra vyvinutá v 70. letech ve firmě IBM. V roce 1976 byla zvolena za standard pro šifrování dat v civilních státních organizacích v USA a následně se rozšířila i do soukromého sektoru. Svou dominantní pozici si udržela až do roku 2001, kdy byla nahrazena algoritmem *Rijndael*, jenž byl uznán novým standardním šifrovacím algoritmem AES. DES je symetrická bloková šifra pracující s bloky délky 64 bitů, která používá pro šifrování i dešifrování jeden klíč stejné délky.

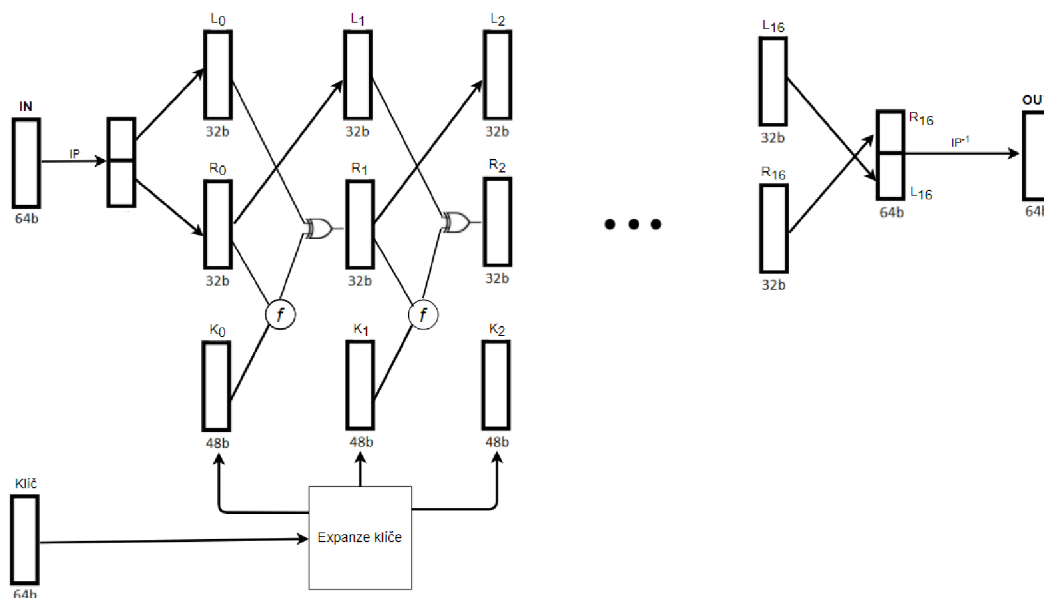
3.1 Princip

Vstupem algoritmu DES je vždy symetrický klíč a otevřený text zprávy, jeho princip je ilustrován na obrázku 3.1. V levé části je vstupem nezašifrovaná zpráva a tajný klíč, v pravé části se nachází výstupní zašifrovaná zpráva. Nejprve je text zprávy rozdělen do bloků po 64 bitech. Pokud je potřeba, na poslední blok je aplikována i výplň, popsaná v sekci 2.4.1. Ze vstupního klíče je potom pomocí procesu *expanze klíčů* vytvořeno 16 podklíčů, které jsou využity při iteračním šifrování. Blok otevřeného textu je nejprve podroben *počáteční permutaci* (anglicky *initial permutation*). Poté následuje výpočetně nejnáročnější část algoritmu: 16 kol šifrování a permutací. Během každé iterace je použit výsledek z předchozího kola spolu s jedním z 16 podklíčů vygenerovaných během expanze původního klíče. Současně rozdělí permutovaný blok na levou a pravou polovinu. Během každé další iterace potom algoritmus pracuje s dvěma polovinami dlouhými 32 bitů, a to tak, že nejprve přiřadí nové levé polovině hodnotu předchozí pravé poloviny. Hodnotu nové pravé poloviny pak zjistí z výsledku operace XOR nad levou polovinou předchozí iterace a výstupem funkce f pro pravou polovinu předchozího kola a odpovídající podklíč. Po konci 16. kola je na blok aplikována jedna poslední permutace, jejíž výsledkem je 64 bitová zašifrovaná zpráva. Při užití algoritmu DES se délka vstupních a výstupních dat neliší.

Expanze klíče

Proces expanze klíče ilustruje obrázek 3.2. Nejprve je nutno získat 16 podklíčů, právě jeden pro každou iteraci výpočtu. Ač je vstupní klíč dlouhý 64 bitů, 8 bitů tvoří kontrolní *paritní bity*. Efektivní délka je tedy pouze 56 bitů. Klíč je potřeba expandovat na 16 nových 48 bitových podklíčů, ale nejprve musí projít permutací. K tomu je využívána permutační tabulka PC-1 na obrázku 3.3.

To v praxi znamená, že 57. bit původního klíče je 1. bitem výsledného permutovaného klíče. 2. bitem permutovaného klíče je 49. bit původního a 3. bit permutovaného je 41. bit



Obrázek 3.1: Ilustrace principu fungování algoritmu DES.

původního. Posledním bitem permutovaného klíče je 4. bit původního. Vstupní klíč K této permutační tabulky by potom byl transformován na výstupní hodnotu permutovaného klíče K_{pc}. Ten má však pouze 56 bitů, protože během permutace byly paritní bity vypuštěny.

$$K = 00010011\ 00110100\ 01010111\ 01111001\ 10011011\ 10111100\ 11011111\ 11110001_2$$
$$K_{pc} = 1111000\ 0110011\ 0010101\ 0101111\ 0101010\ 1011001\ 1001111\ 0001111_2$$

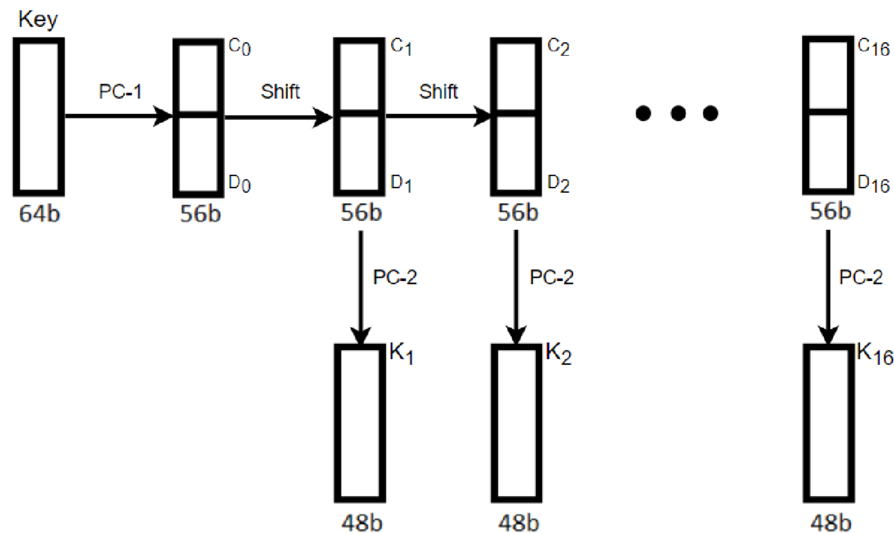
Tento klíč K_{pc} je pak rozdělen na dvě stejně dlouhé poloviny o délce 28 bitů, na levou část C_0 a pravou část D_0.

$$C_0 = 1111000\ 0110011\ 0010101\ 0101111_2$$
$$D_0 = 0101010\ 1011001\ 1001111\ 0001111_2$$

Pomocí páru hodnot C_0 a D_0 je postupně sestaveno 16 dalších párů C_1D_1, C_2D_2, C_3D_3 až po $C_{16}D_{16}$. Pro získání hodnoty následujícího páru hodnot $C_{n+1}D_{n+1}$ je potřeba hodnoty C_nD_n bitově posunout doleva (první hodnota se přesouvá na konec). Klíče C_1D_1, C_2D_2, C_9D_9 a $C_{16}D_{16}$ jsou posunuty oproti předchozímu páru hodnot o 1 bit, všechny ostatní jsou posunuty o 2 bity. Finálních 16 podklíčů lze získat z těchto 16 párů aplikací další permutační tabulky PC-2 z obrázku 3.4. Vstupem této tabulky je spojená levá a odpovídající pravá polovina bitově posunovaných párů C_nD_n o délce 56 bitů, výstupem je ale podklíč K_n o délce pouze 48 bitů[7]. Všechny z podklíčů K_n tak vzniknou permutací hodnot odpovídajícího páru C_nD_n. Pro zjištění hodnoty prvního podklíče K_1 je tedy třeba znát hodnotu C_1 a D_1.

$$C_1 = 1110000110011001010101011111_2$$
$$D_1 = 1010101011001100111100011110_2$$
$$C_1D_1 = 1110000110011001010101011111\ 1010101011001100111100011110_2$$
$$K_1 = 000110\ 110000\ 001011\ 101111\ 111111\ 000111\ 000001\ 110010_2$$

Každý takový 48 bitový podklíč bude využit při odpovídajícím kole šifrování v následujícím kroku.



Obrázek 3.2: Nákres kroku expanze klíče algoritmu DES. Vlevo vstupuje do procesu 64 bitový tajný klíč, ve spodní části se nachází výsledné 48 bitové podklíče.

PC-1						
57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

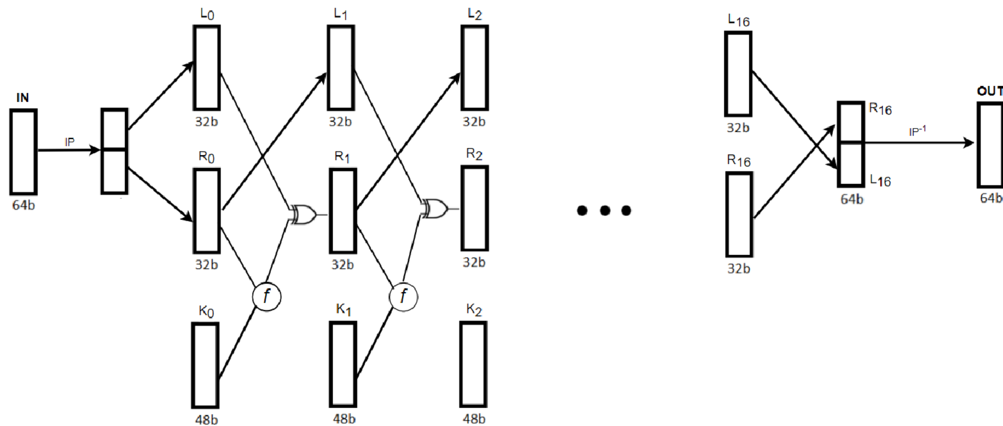
Obrázek 3.3: Permutační tabulka PC-1 pro expanzi klíče algoritmu DES[7].

PC-2					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Obrázek 3.4: Permutační tabulka PC-2 pro expanzi klíče algoritmu DES[7].

Permutace

Permutace je operace, která zamění pořadí prvků v uspořádané n-tici tak, že jednotlivé prvky se v nové permutované n-tici objeví právě jednou. Nákres popisující postup tohoto iterativního výpočtu je v obrázku 3.5. Algoritmus DES provádí celkově 16 iterací permutací, před samotnými koly šifrování je ale třeba na každý blok aplikovat počáteční permutaci (anglicky *initial permutation*). Tato permutace se řídí tabulkou IP v obrázku 3.6. Z této tabulky lze zjistit, že 1. bitem výstupní hodnoty je 58. bit vstupní hodnoty, 2. bitem výsledné hodnoty je 50. bit původní a obdobně posledním výsledným bitem je 7. bit původního bloku.



Obrázek 3.5: Nákres iterací permutací algoritmu DES. V levé části do procesu vstupuje nezašifrovaná vstupní zpráva, ve spodní části jsou vyobrazeny odpovídající podklíče a výsledná zašifrovaná zpráva je v pravé části obrázku.

IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Obrázek 3.6: Tabulka počáteční permutace šifrovacího algoritmu DES. Jejím vstupem je 64 bitů bloku vstupní zprávy, nad kterou provádí permutace. Výsledek permutace je stejné délky[7].

Pro vstupní blok M, hexadecimálně reprezentován jako $M = 0123456789ABCDEF_{16}$, tabulka po provedení permutace vystaví na výstup hodnotu M_p. Těchto 64 bitů potom DES rozdělí na dvě stejně dlouhé poloviny o délce 32 bitů, levou polovinu L_0 a pravou polovinu R_0.

$$
\begin{aligned}
M &= 01234567_{16} \\
M &= 00000001\ 00100011\ 01000101\ 01100111\ 10001001\ 10101011\ 11001101\ 11101111_2 \\
M_p &= 11001100\ 00000000\ 11001100\ 11111111\ 11110000\ 10101010\ 11110000\ 10101010_2 \\
L_0 &= 11001100\ 00000000\ 11001100\ 11111111_2 \\
R_0 &= 11110000\ 10101010\ 11110000\ 10101010_2
\end{aligned}
$$

Počáteční hodnoty vstupují do prvního kola permutace, kde pro každou iteraci algoritmu použije výsledek předchozího kola a odpovídající 48 bitový podklíč. Každou nově vypočítanou hodnotu L_n a R_n vypočítá podle vzorce 3.1[7], kde přirozené číslo n nabývá

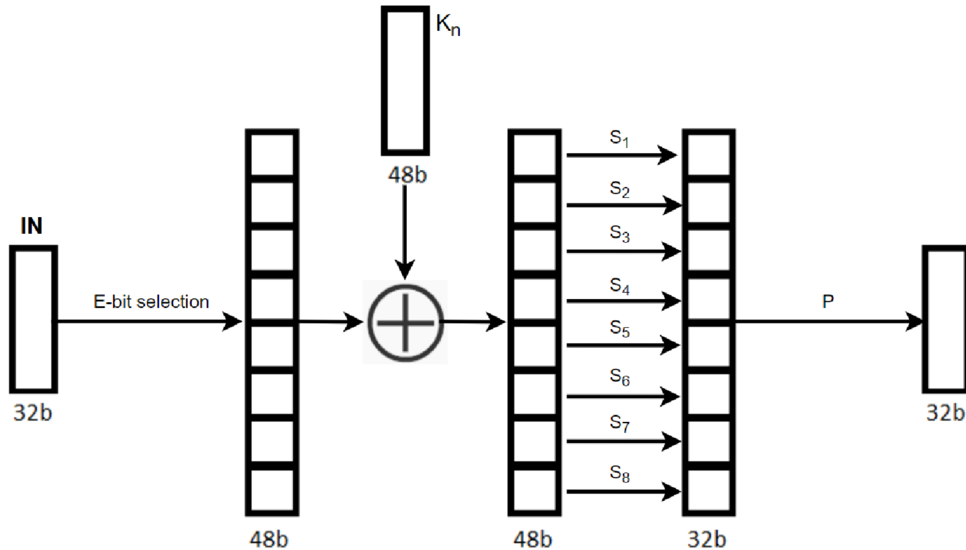
hodnoty od 1 po 16, v závislosti na pořadí kola permutace. Pro první permutaci je číslo n rovno 1, při poslední je rovno 16.

$$\begin{aligned} L_n &= R_{n-1} \\ R_n &= L_{n-1} \oplus f(R_{n-1}, K_n) \end{aligned} \tag{3.1}$$

Ze vzorce vyplývá, že při každém kole permutace je levé polovině přiřazena hodnota pravé poloviny z předchozího kola. Pravou polovinu získá po provedení logické operace *XOR* nad levou polovinou předchozího páru a výsledkem funkce *f*. Tento proces se opakuje, dokud algoritmus nezíská hodnoty L_{16} a R_{16}.

Funkce *f*

Funkce *f*, znázorněna na obrázku 3.7, je klíčová součást algoritmu DES. Vstupními parametry funkce f je hodnota pravé části z předchozího kola permutace R_{n-1} a hodnota odpovídajícího podklíče K_n. Funkce f nejprve expanduje hodnotu R_{n-1} z 32 bitů na 48 bitů. K tomu využívá tabulku *E-bit selection* 3.8, která je schopna hodnotu na 32 bitech rozšířit na hodnotu o 48 bitech.



Obrázek 3.7: Nákres hardwarové implementace funkce f algoritmu DES. V levé části vstupuje do procesu pravá polovina mezivýsledku permutace, v horní části je aplikován odpovídající podklíč K_n. V pravé části je výsledná hodnota o 32 bitech.

Tato tabulka funguje obdobně jako výše vyobrazené permutační tabulky, s tím rozdílem, že některé bity duplikuje. Například hodnota pravé strany permutovaného prvního bloku R_0 po rozšíření permutační E bit-selection tabulkou 3.8 nabude hodnoty $E(R_0)$.

$$\begin{aligned} R_0 &= 00000000\ 00000000\ 10000000\ 01100110_2 \\ E(R_0) &= 000000000000000000000001010000000000001100001100_2 \end{aligned}$$

Dalším krokem funkce f je provedení operace XOR nad předchozí expandovanou hodnotou pravé strany $E(R_{n-1})$ a podklíčem K_n. Za povšimnutí stojí skutečnost, že výsledek této operace má 48 bitů, avšak výstupem funkce f je pouze 32 bitů. Pro redukci počtu bitů se používají substituční tabulky, tzv. *S-boxes*. Vstupem každého z těchto S-boxů je 6 bitů,

E bit-selection					
32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Obrázek 3.8: Permutační tabulka *E-bit selection* využívána algoritmem DES pro řešení funkce *f*. Hodnota vstupující do operace řízené touto tabulkou má délku 32 bitů, výstupní hodnota má po duplikaci vybraných prvků délku 48 bitů[7].

jeho výstupem jsou pouze bity 4. Toho lze využít, pokud je 48 bitů rozděleno na 8 skupin B_i tak, že každá skupina má právě 6 bitů $B_1B_2B_3B_4B_5B_6B_7B_8$. Výstupem po použití S-boxů je potom obecně $S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$. Všechny S-boxy, které jsou využívány jsou k dohledání v příloze C. Pokud tedy do S-boxu S_1 3.9 vstoupí hodnota B_1, výsledkem bude 4 bity dlouhá hodnota V.

S1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Obrázek 3.9: Substituční tabulka *(S-box)* používána algoritmem DES při řešení funkce *f*. Konkrétně tato tabulka je aplikována na první bajt vstupního bloku. Vstupem tabulky je šestibitová hodnota, jejím výstupem jsou bity čtyři[7].

První a poslední bit z hodnoty B reprezentují dohromady v dvojkové soustavě číslo od 0 do 3. Toto číslo se nazývá i. Čtyři prostřední bity hodnoty B reprezentují v dvojkové soustavě číslo od 0 do 15, nazývá se j. Hodnota i ukazuje na řádek tabulky, hodnota j na sloupec. Díky tomu, že buňky tabulky obsahují pouze hodnoty od 0 do 15, mohou být reprezentovány pouze na 4 bitech. Právě tyto 4 bity jsou výstupem operace s touto tabulkou.

$$
\begin{aligned}
B &= 011011_2 \\
i &= 01_2 \\
j &= 1101_2 \\
S_1(B) &= 0101
\end{aligned}
$$

Posledním krokem funkce f je permutace P výstupu z S-boxů. Permutace probíhá stejně, jako v předchozích operacích s tím rozdílem, že je řízena tabulkou 3.10. Výsledná hodnota tedy začíná 16. bitem původní hodnoty, následuje původní 7. bit a je zakončena 25. bitem původní hodnoty. Výstup této permutace je zároveň výsledkem funkce f. Pomocí této nově získané hodnoty lze zjistit hodnotu pravé strany R_{n+1}.

P			
16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Obrázek 3.10: Permutační tabulka algoritmu DES. Ta provádí permutaci, jejíž výstupem je zároveň výsledek funkce f[7].

Finální výsledek algoritmu

Po získání hodnot R_{16} a L_{16} je pořadí těchto 32 bitových hodnot přehozeno, čímž vznikne 64 bitová hodnota $R_{16}L_{16}$. Na získanou hodnotu DES aplikuje poslední permutaci podle tabulky IP^{-1} v obrázku 3.11.

IP^{-1}							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Obrázek 3.11: Permutační tabulka IP^{-1}, která definuje úplně poslední krok v algoritmu DES. Vstupem je 64 bitů pravé a levé poloviny $R_{16}L_{16}$. Jejím výstupem je potom finální zašifrovaná zpráva[7].

Pro zprávu $M = 0123456789ABCDEF_{16}$ a klíč $K = 133457799BBCDFF1_{16}$ bude výstupem algoritmu DES šifrovaný text C.

$$L_{16} = 01000011\ 01000010\ 00110010\ 00110100_2$$
$$R_{16} = 00001010\ 01001100\ 11011001\ 10010101_2$$
$$R_{16}L_{16} = 00001010\ 01001100\ 11011001\ 10010101\ 01000011\ 01000010\ 00110010\ 00110100_2$$
$$C = 10000101\ 11101000\ 00010011\ 01010100\ 00001111\ 00001010\ 10110100\ 00000101_2$$
$$C = 85E813540F0AB405_{16}$$

Dešifrování probíhá stejně, pouze s jednou odchylkou. Během 16 kol permutací je pořadí použitých podklíčů obráceno, do prvního kola tedy vstupuje podklíč K_{16}, do druhého K_{15} a do posledního podklíč K_1. Ve všech ostatních krocích algoritmus pracuje stejně.

3.2 Analyzované implementace

Tato podkapitola nabízí srovnání tří implementací algoritmu DES jak z pohledu hodnoty bitové rychlosti, tak z pohledu kvality a uchopitelnosti kódu. Podmínkou pro to, aby implementace vůbec mohla být srovnávána, je možnost s ní volně a zdarma pracovat. Dále hraje svou roli i struktura a funkčnost rozhraní, čitelnost kódu nebo jak snadno ji lze integrovat do měřicího kódu. Měřeny a srovnávány jsou volání funkcí knihovny OpenSSL, implementace z webu Rosetta Code a zdrojový kód dostupný na stránce Programming Algorithms.

3.2.1 Implementace z Rosetta Code

Tato implementace je sice volně dostupná na stránce Rosetta Code[1]. Jedná se o web, kde programátoři sdílí řešení jednoho problému v co největším počtu jazyků. Velkou výhodou implementace je to, že je snadno čitelná a přehledná. Kód je sice v jednom obsáhlém zdrojovém kódu, obsahuje ale komentáře ke každé funkci, významnému logickému celku či těžko porozumitelnému příkazu. Během integrace do měřicího kódu byla tato implementace rozšířena o možnost zpracování různých délek zpráv a režie volání jeho hlavní funkce sníženy na minimum. Práce s implementací a integrace do kódu pro měření byla snadná a přímočará. Za zmínku stojí poznámka, že tato implementace je překlad z jazyka *D*. Verze z jazyka D je pak překladem verze z jazyka *Kotlin*. Všechny zmíněné a mnoho dalších implementací je dostupných na stejné URL adrese.

3.2.2 Implementace z Programming Algorithms

Zdrojový kód implementace byl převzat z webové stránky[2] Programming Algorithms, jehož cílem je pomáhat dalším vývojářům ve studiu a použití populárních algoritmů v jazycích jako C, C/C++, C#, VB.Net nebo PHP. Nevýhodou této implementace je to, že bez předchozího studia fungování algoritmu DES je prakticky nečitelná a celý zdrojový kód neobsahuje žádné komentáře. Implementace na začátku definuje S-boxy a další potřebné tabulky, ale nemá metodu `main`, pouze `encrypt`. Jen pro člověka s detailními znalostmi DES jsou deskriptory funkcí pochopitelné, příkazy v tělech funkcí však už také vyžadovaly čas k pochopení významu. I přes to je možné zdrojový kód relativně snadno upravit a po minimalizaci režií integrovat do měřicího kódu.

[1] https://rosettacode.org/wiki/Data_Encryption_Standard
[2] https://www.programmingalgorithms.com/algorithm/des/c/

3.2.3 Použití knihovny OpenSSL

Tato implementace využívá prostředků knihovny OpenSSL. OpenSSL je open-source projekt pracující na knihovně implementací protokolů SSL a TLS. I když jsou knihovní funkce psány v jazyce C, mohou být využívány ve velkém množství dalších programovacích jazyků díky tzv. *wrapperům*, tedy mezivrstvou mezi funkcemi OpenSSL v C a kódem v jiném jazyce. Knihovna OpenSSL je dostupná pro většinu Unixových systémů (včetně Linuxu, macOS nebo BSD) a Microsoft Windows. Velkým problémem při použití této knihovny byla skutečnost, že příklady jejího užití v C jsou velmi vzácné a většina z nich je buď nefunkční, nebo zcela nevhodná. Dalším byl fakt, že poměrně nedávno se rozhodli tvůrci OpenSSL změnit rozhraní pro práci s knihovnou, což ztížilo práci se staršími zdroji. Bylo tedy nutno tento kód vytvořit podle aktuální dokumentace OpenSSL[3] pro jeho správné fungování.

3.3 Výsledky měření

Cílem měření bylo zjistit, s jakou rychlostí jsou schopny vybrané implementace z předchozí podkapitoly zašifrovat a dešifrovat zprávy. Jakožto veličina pro srovnání byla zvolena bitová rychlost, která vyjadřuje množství zpracovaných informací za vteřinu. Měření probíhalo pomocí kódu v jazyce C. Během jednoho běhu probíhá vždy zašifrování a dešifrování předem definované zprávy, tato operace se opakuje 5000x. Výsledky v tabulce jsou aritmetickým průměrem hodnot odpovídajících algoritmů a délky zprávy. Algoritmus pracuje s několika délkami zpráv: 128 bitů, 512 bitů a 2048 bitů. Inicializace potřebných struktur, generování klíče či volání funkcí představuje určitou režii. To se nejvíce projeví u šifrování krátkých zpráv, kde režie činnosti programu zaberou podstatnou část celkové doby běhu kódu. Pro získání lepšího přehledu a přesnějších výsledků měření probíhalo s několika délkami zpráv. Výstup měřicího programu byl zpracován do tabulky 3.12.

Název implementace	Čas publikace	Poslední aktivita	Popularita	Průměrný bitrate [kB/s]		
				128b zpráva	512b zpráva	2048b zpráva
Rosetta	x	07.09.2020	x	372	672	823
OpenSSL	23.12.1998	20.11.2020	14.3k stars, 6.4k forks, 908 watched	130 781	327 551	558 733
Programming Algorithms	x	x	x	33 179	129 514	526 567

Obrázek 3.12: Výsledná tabulka měření 3 implementací algoritmu DES.

O výsledcích lze obecně říct, že s rostoucí délkou zprávy se dopad režií běhu programu snižoval, čímž výrazně stoupla i bitová rychlost algoritmu. To se obzvláště projevilo u kódu Programming Algorithms, jehož šifrovnání 2048 bitových zpráv má skoro 16x větší bitovou rychlost, než práce s 128 bitovými zprávami. Implementace z Rosetty se při porovnání rychlosti šifrování mezi 128 bitovou zprávou a 2048 bitovou svůj bitrate více než zdvojnásobila, rychlost OpenSSL se v tomto porovnání narostl více než 4x.

Z tabulky lze také vyčíst, že knihovna OpenSSL je schopna pracovat nejrychleji, následovaná implementací webu Programming Algorithms. Ta při měření s použitím dlouhých 2048 bitových zpráv dosáhla zhruba stejné rychlosti, jako knihovna OpenSSL, při práci s kratšími zprávami je ale již o poznání pomalejší. O několik řádů pomalejší a v praxi ne-

[3] https://www.openssl.org/docs/manpages.html

použitelná je potom implementace z webu Rosetta. V šifrování 128 bitových zpráv je 351x pomalejší než je knihovna OpenSSL, s 2048 bitovými zprávami je dokonce 679x pomalejší.

Kapitola 4

Algoritmus AES

V roce 1976 americký institut NIST zvolil algoritmus DES, od firmy IBM, standardem pro šifrování dat jakožto ochranu před únikem informací. V době svého vzniku a řadu let poté byl algoritmus bezpečný a spolehlivý, protože neexistovaly technické prostředky, pomocí kterých by bylo možné šifru prolomit. Jak ale množství výpočetní síly postupně narůstalo, začali se útočníci pokoušet překonat šifru hrubou silou. Proto byl nucen NIST vyhlásit dne 2. ledna 1997 nový konkurz na nový standard šifrovacího algoritmu. Po čtyřech letech schvalovací procedury NIST schválil 26. listopadu 2001 několik šifer z rodiny šifer Rijndael jako nejvhodnější návrh z patnácti předložených. AES je první šifra dostupná široké veřejnosti, která byla zároveň uznaná Národní bezpečností agenturou NSA jako vhodná k šifrovaní utajených dokumentů. Dne 26. května 2002 začala být ke svému účelu používána jako federální standard USA.

Advanced Encryption Standard je dodnes používaný algoritmus používaný k šifrování dat. Původní název šifry Rijndael vznikl jako přesmyčka příjmení dvou autorů, *Joana Daemena* a *Vincenta Rijmena*, žijících v belgické Lovaně. Tito autoři spolu vytvořili celou rodinu šifer Rijndael, kde každá z šifer se vyznačovala svými odlišnými vlastnostmi. Z šifer bylo vybráno pouze několik, které byly schopny pracovat s klíči délky 128 bitů, 192 bitů či 256 bitů, ty byly potom vyhlášeny standardem AES.

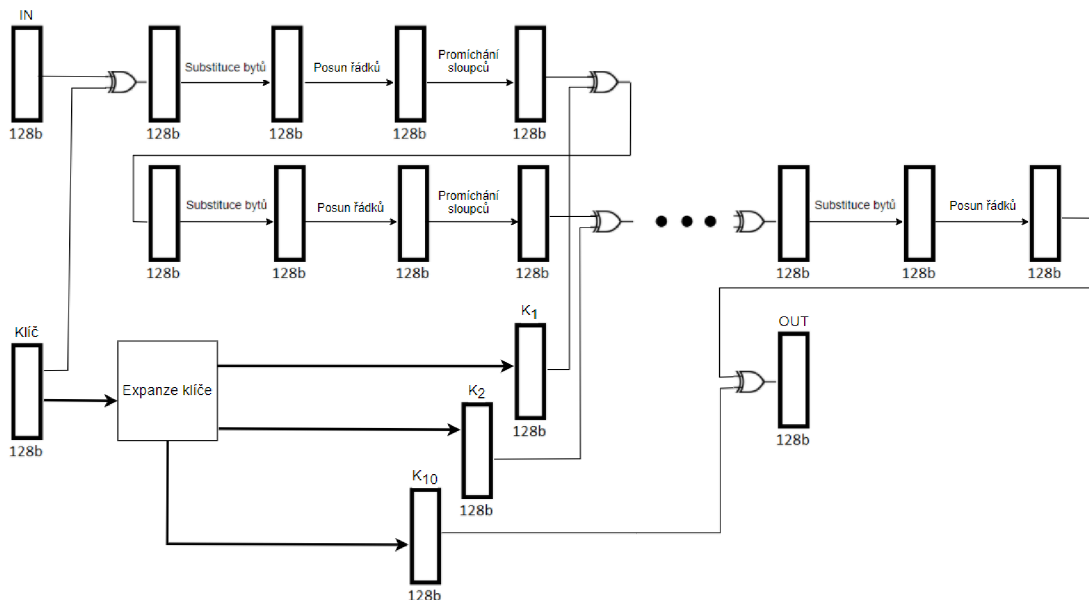
Podobně jako DES i AES je bloková šifra, která pracuje s bloky pevně dané délky 128 bitů (viz 2.4) a k šifrování i dešifrování používá jeden klíč, což z ní dělá šifru symetrickou (viz 2.1). Délka výstupních zašifrovaných dat je stejná, jako délka dat vstupních a pokud zpráva nedokáže přesně vyplnit všechny bloky, je potřeba použít výplň, o postupech jak vyplnit poslední blok již pojednává podkapitola 2.4.1. Původní záměr americké vlády bylo vyvinout šifru, která by chránila utajené informace před útoky. Postupem doby si ale našla mnoho uplatnění, dnes například pomáhá chránit přenosy souborů po internetu nebo i bezdrátovou komunikaci. Velmi často chrání Wi-Fi připojení v kombinací se zabezpečením bezdrátových sítí jménem *WPA2 (Wi-Fi Protected Access)*, dohromady je pak označujeme jako *AES/WPA2*, jenž nahrazuje starý bezpečnostní protokol *TKIP (Temporal Key Integrity Protocol)*. Algoritmus AES je využíván mimo jiné například při zabezpečení SSL nebo k ochraně dat na discích.

4.1 Princip

Fungování algoritmu je znázorněno v obrázku 4.1. Do algoritmu AES vstupuje symetrický klíč a vstupní data, jeho výstupem jsou vždy data stejné délky. Volitelným vstupním

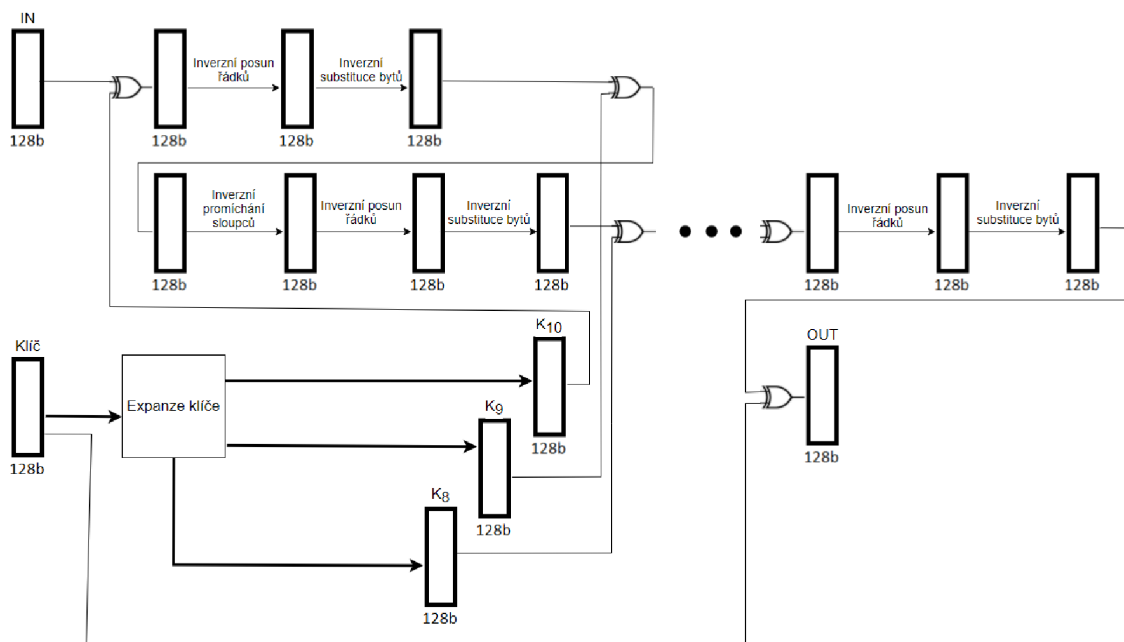
parametrem je i inicializační vektor, který je však ve většině případů užití náhodně generován. Nejprve je vstupní text zprávy rozdělen do bloků po 128 bitech, pokud je potřeba, na poslední blok je aplikována i výplň (viz podkapitola 2.4.1). Ze vstupního klíče je potom pomocí procesu *expanze klíčů* (anglicky *key schedule*) vytvořeno buď 10, 12 nebo 14 podklíčů, které jsou využity při iteračním šifrování. Počet podklíčů závisí na délce symetrického klíče, pro 128 bitový klíč je vygenerováno 10 podklíčů, pro 192 bitový klíč 12 podklíčů a pro nejdelší 256 bitový klíč je vygenerováno podklíčů 14.

Poté následuje nejnáročnější část výpočtu: 10, 12 nebo 14 kol permutací. Během každé iterace je použit výsledek z předchozí iterace spolu s jedním z podklíčů vygenerovaných během expanze původního klíče. V každé iteraci proběhnou čtyři mezikroky: substituce bajtů podle substituční tabulky, posun řádků tabulky vyjadřující daný blok, promíchání sloupců pomocí násobení předem pevně danou maticí a nakonec je mezivýsledek podroben operaci XOR s příslušným podklíčem pro dané kolo. Po konci poslední iterace (která vynechává mezikrok promíchání sloupců) permutací je už známa podoba zašifrovaného textu, který je výstupem z poslední operace v iteraci. Pro dešifrování, které je popsáno obrázkem 4.2, je třeba uplatnit opačný postup, ke kterému stačí převrátit pořadí operací v iteračním výpočtu a invertovat substituční tabulku. V posledním kole je vynechán krok inverzního promíchání sloupců. Výstupem dešifrování je pak opět původní text zprávy.



Obrázek 4.1: Vysokoúrovňový nákres šifrování algoritmu AES.

Každý vstupní blok M o délce 16 bajtů b je rozmístěn do tabulky, nad kterou potom proběhne pevně daný počet kol permutací. Pro klíč délky 128 bitů AES provede 10 kol, pro 192 bitový klíč bude kol 12 a pro klíč o 256 bitech bude provedeno 14 kol permutací. Každý šifrovaný blok je rozdělen do pomyslné mřížky 4.3, kde každá buňka obsahuje právě jeden bajt vstupního bloku. bajty se do mřížky vkládají tak, že po sobě následující čtveřice bajtů vyplní jeden sloupec.



Obrázek 4.2: Vysokoúrovňový nákres dešifrování algoritmu AES.

b_0	b_4	b_8	b_{12}
b_1	b_5	b_9	b_{13}
b_2	b_6	b_{10}	b_{14}
b_3	b_7	b_{11}	b_{15}

Obrázek 4.3: Mřížka ilustrující výsledek rozvržení 16 bajtů algoritmem AES.

Expanze klíče

Algoritmus AES podobně jako DES potřebuje ke svému iteračnímu výpočtu expandovat svůj klíč na určitý počet podklíčů. Pro délku klíče 128 bitů expanduje původní klíč na pole w o délce 176 bajtů (44 slov). Z původního 128 bitového klíče pak vznikne 10 nových podklíčů, z 192 bitového jich bude 12 a při užití 256 bitového klíče jich vznikne 14, přičemž podklíčem pro první kolo je samotný klíč.

Pro klíč o délce 128 bitů jsou všechna 4 slova klíče nejprve překopírovány do pole slov w. Všechny další slova v poli w potom vzniknou jako výsledek operace XOR mezi předcházejícím slovem a slovem, které se nachází 4 položky vlevo. To znamená, že pro zjištění hodnoty jakéhokoliv slova, jehož index není násobkem 4, může být využit vzorec 4.1. Výjimkou z tohoto postupu je každé čtvrté slovo, tedy slova na pozicích w[4], w[8] atd.. Hodnoty takových slov jsou získány pomocí funkce g.

$$w[i] = w[i-1] \oplus w[i-4] \tag{4.1}$$

Funkce *g*

Funkce g je součástí postupu algoritmu AES, kde je během procesu expanze tajného klíče využívána pro výpočet hodnoty každého čtvrtého slova. Princip této funkce je vyobrazen v

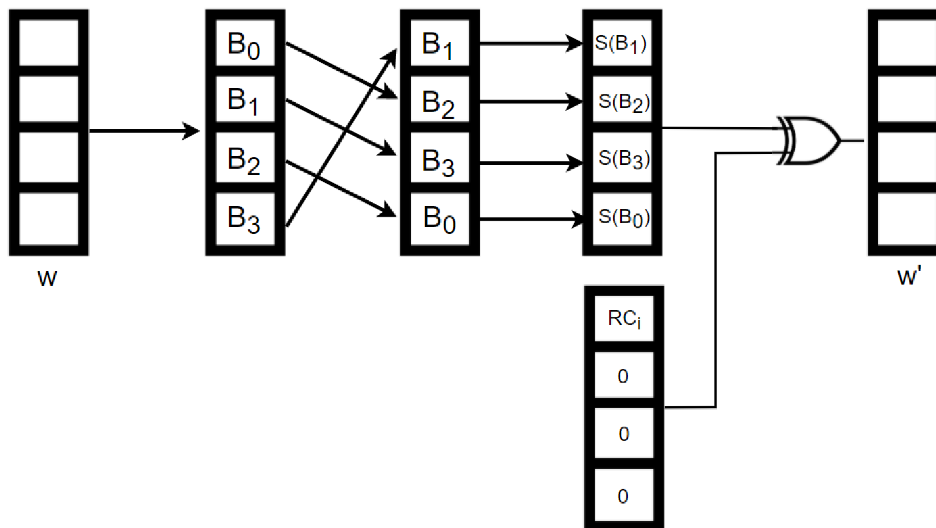
nákresu 4.5. Tímto zvyšuje bezpečnost algoritmu s použitím pouze několika velmi rychlých výpočtů. Nejprve je vstupní slovo posunuto doleva o 8 bitů. Následně je každý z těchto bajtů substituován podle Rijndael S-boxu (viz 4.7). Levé 4 bity substituovaného bajtu slouží jako číslo řádku i, kde se nová hodnota nachází. Poslední pravé 4 bity ukazují správný index sloupce j. Jak řádky, tak sloupce substituční tabulky jsou číslovány od 0. Příklad 4.2 ilustruje postup funkce g.

$$\begin{aligned} B &= 00100010_2 \\ i &= 0100_2 = 4 \\ j &= 0010_2 = 2 \\ S(B) &= 2c_{16} \end{aligned} \tag{4.2}$$

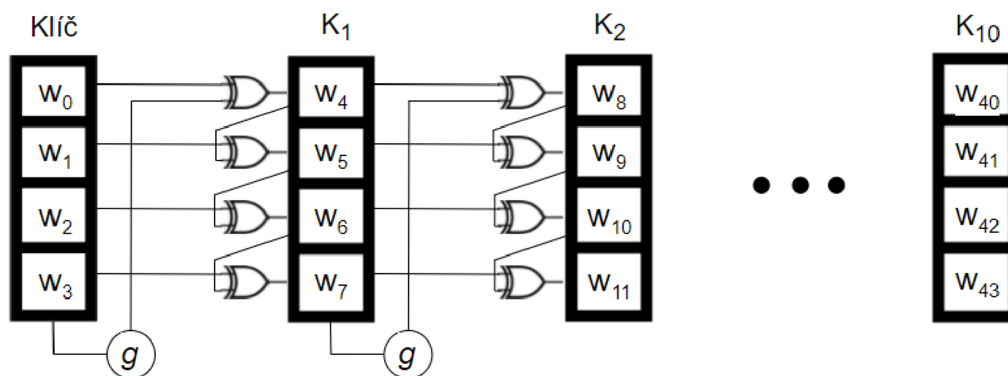
Výstupem funkce g je výsledek operace XOR provedené nad substituovaným slovem z předcházejícího kroku a hodnotou konstanty (anglicky *Round constant*). Hodnoty konstant jsou pevně dané pro každé kolo, jejich hodnoty jsou zapsány v tabulce 4.4 v hexadecimální soustavě. Celý algoritmus pro expanzi klíčů algoritmem AES je znázorněn obrázku 4.6.

j	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
RC[j]	1	2	4	8	10	20	40	80	1B	36

Obrázek 4.4: Tabulka RC (round constant) je užívána při kroku expanze klíče algoritmem AES. Obecně hodnota konstanty pro kolo i lze zjistit výpočtem 2^{i-1}[8]. Hodnoty v tabulce jsou zapsány v hexadecimálním tvaru.



Obrázek 4.5: Nákres postupu fungování funkce g, vlevo se nachází vstupní w slovo o délce 32 bitů, vpravo je výsledné slovo *w'*.



Obrázek 4.6: Grafický popis postupu algoritmu AES během generování podklíčů.
.

Permutace

Během práce algoritmu AES dochází k iteračnímu výpočtu permutací. Počet kol je pevně daný, těch může být 10, 12, nebo 14 v závislosti na délce tajného klíče. Každé kolo permutace se skládá ze 4 podkroků, kromě kola posledního, které vynechává krok promíchání sloupců. Výsledný blok po skončení všech iterací permutací je zároveň výstupem algoritmu.

Substituce bajtů

V prvním kole výpočtu je vstupem substituce blok vstupních dat, v dalších kolech je vstupem výsledek ze předcházejícího kola. Během substituce bajtů je každá z buněk ve vstupní mřížce bloku substituována jinou, předem danou hodnotou, k čemuž je využívána substituční tabulka Rijndael S-box. Nová nahrazená hodnota, řízená právě Rijndael S-boxem, je výstupem podkroku substituce bajtů. Hodnoty S-boxu (viz 4.7) nebyly zvoleny náhodně, nýbrž tak, aby dále komplikovaly a znesnadňovaly prolomení šifry. Ke zvýšení bezpečnosti jim pomohl velmi důležitý fakt: všechny hodnoty, se kterými algoritmus pracuje, tvoří dohromady konečné těleso, díky čemuž je algoritmus schopen provádět základní logické operace nad prvky hodnot, jejíž výsledek bude opět vždy patřit do konečného tělesa. Toho využili tvůrci šifry a vytvořili Rijndael S-box, který je jedním z důvodů, proč je AES tak těžké překonat. V tomto konečném tělese neexistují žádné pevné body, tedy hodnoty, které by se zobrazovaly samy do sebe. Stejně tak neexistuje žádný opačný pevný bod, kdy by se bitově reprezentovaná hodnota zobrazila do bitové převrácené hodnoty. Stejný Rijndael S-box je také využíván již během expanze klíče funkcí g.

		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	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

Obrázek 4.7: Substituční tabulka S-box využívána algoritmem AES. Používá ji pro substituci bajtů během procesu expanze klíče i během samotných iterací šifrování. V takovém případě do tabulky vstupuje výsledek předchozího kola iterace (pokud právě probíhá první iterace, vstupem je blok zprávy), výstupem je permutovaná mřížka. Při procesu expanze klíče je tabulka využívána funkcí g[8].

Substituce podle tabulky 4.7 pak probíhá tak, že první 4 bity nahrazované hodnoty i určují sloupec. Druhá čtveřice bitů pak určuje sloupec j.

$$\begin{aligned} b &= 00101101_2 \\ i &= 0010_2 = 2_{10} \\ j &= 1101_2 = 13_{10} \\ S(b) &= D8_{16} \end{aligned}$$

Výstupem tohoto kroku je tabulka obsahující stejný počet bitů, jako je vstupní tabulka, vyplněna substituovanými hodnotami (viz 4.8). Při dešifrování algoritmus využívá inverzní tabulky S-box 4.10, kterou sice již nevyužívá k expanzi klíče, ale jinak s ní pracuje stejně, jako s původním Rijndael S-boxem.

$S(b_0)$	$S(b_4)$	$S(b_8)$	$S(b_{12})$
$S(b_1)$	$S(b_5)$	$S(b_9)$	$S(b_{13})$
$S(b_2)$	$S(b_6)$	$S(b_{10})$	$S(b_{14})$
$S(b_3)$	$S(b_7)$	$S(b_{11})$	$S(b_{15})$

Obrázek 4.8: Obecný tvar výstupu S-boxu algoritmu AES z mřížky 4.3.

$S(b_0)$	$S(b_4)$	$S(b_8)$	$S(b_{12})$
$S(b_5)$	$S(b_9)$	$S(b_{13})$	$S(b_1)$
$S(b_{10})$	$S(b_{14})$	$S(b_2)$	$S(b_6)$
$S(b_{15})$	$S(b_3)$	$S(b_7)$	$S(b_{11})$

Obrázek 4.9: Mřížka z kroku substituce bajtů 4.8 po provedení posunu řádků.

		y															
		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
x	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

Obrázek 4.10: Inverzní substituční tabulka S-box využívána algoritmem AES. Používá ji pro substituci bajtů během iterací dešifrování. V takovém případě do tabulky vstupuje výsledek předchozího kola iterace (pokud právě probíhá první iterace, vstupem je blok zprávy), výstupem je permutovaná mřížka[8].

Posun řádků

V dalším kroku AES posune každý řádek tabulky doleva o hodnotu x, kde x je index řádku tabulky. Znamená to, že 1. řádek tabulky posune o 0 bajtů, 2. řádek s indexem 1 o 1 bajt, 3. řádek je posunut o 2 bajty a poslední o 3 bajty doleva, stejně jako by byl posunut o jeden sloupec doprava. Výstupem posunu řádků je další permutace provedena nad substituovanými hodnotami z předešlého kroku substituce bajtů.

Promíchání sloupců

Krok promíchání sloupců spočívá v násobení každého sloupce předem danou maticí 4.11. Prvním činitelem je právě tato předem daná matice, druhým činitelem je vektor reprezentující daný sloupec vstupní tabulky. Tato matice pomáhá dále znesnadňovat prolomení jakožto další permutace prováděná nad blokem, zároveň však byly úmyslně zvoleny malé hodnoty čísel v tabulce, a to s ohledem na efektivitu a hardwarovou akceleraci.

2	3	1	1
1	2	3	1
1	1	2	3
3	1	1	2

Obrázek 4.11: Matice algoritmu AES, kterou je násobena vstupní mřížka 4.9 během kroku promíchání sloupců[8].

Je důležité mít na paměti, že na 8 bitech lze vyjádřit pouze hodnoty celých čísel v uzavřeném intervalu od 0 do 255. Pokud by výsledná hodnota po násobení maticí měla nabýt hodnoty vyšší nebo rovné 2^8, je potřeba tento výraz vydělit tzv. *Galoisovým ireducibilním polynomem (anglicky Galois irreducible polynomial) P*. Tato operace zajistí, že výsledek V zůstane v mezích hodnot reprezentovatelných na 8 bitech. Protože výsledek násobení

mezi hodnotami A a B nelze vyjádřit na 8 bitech, je třeba jej tímto polynomem P vydělit. Příkladem tohoto postupu je výpočet 4.3, kde V značí výsledek a T množinu celých čísel reprezentovatelných na 8 bitech.

$$\begin{aligned} P &= 2^8 + 2^4 + 2^3 + 2^1 + 1 \\ A &= 212,\ B = 2 \\ V &= A \times B \ \wedge \ V \in T \\ A \times B &= 424 \implies A \times B \notin T \\ V &= \frac{A \times B}{P} = \frac{2^8 + 2^7 + 2^5 + 2^3}{2^8 + 2^4 + 2^3 + 2^1 + 1} = 4 \ \wedge \ 4 \in T \end{aligned} \tag{4.3}$$

Tento proces opakuje pro každou buňku tabulky. Krok promíchání sloupců se provádí v každém kole permutace kromě posledního, kde by docházelo pouze ke zbytečnému zpomalování algoritmu.

Aplikace podklíče

Do podkroku aplikace podklíče vstupuje blok po promíchání sloupců a podklíč, který patří příslušnému kolu. Operace spočívá pouze v užití logické funkce XOR právě nad podklíčem a vstupním blokem. Výstup funkce XOR, a tedy i celého kroku, potom vstupuje do následující iterace. Výjimka nastává v úplně posledním kole, kde výstup tohoto kroku je již finální zašifrovaný výstup šifry AES.

4.2 Analyzované implementace

Kapitola se zaměřuje na srovnání dvou implementací algoritmu AES a užití knihovny OpenSSL, které byly integrovány do měřicího kódu a srovnány z hlediska rychlosti. Aby implementace vůbec mohla být srovnávána, musí v první řadě být volně dostupná. Dalšími důležitými faktory při rozhodování, které implementace srovnávat byly přehlednost rozhraní a snadnost dále modifikovat kód.

4.2.1 Tiny AES

Velkou výhodou implementace[1] je skutečnost, že zabírá velmi málo místa v paměti. Zkompilovaná pro ARM během svého běhu zabírá méně než 200 bajtů RAM 1 až 2 kB ROM. Podporuje také módy šifrování CBC, CTR a ECB. Implementace samotná nezahrnuje výplň bloků, tento problém však již vyřešil uživatel erev0s[2] ve svém Github repozitáři[3], který nabízí zdrojové kódy pro vyplnění neúplných bloků. Součástí zdrojových kódů jsou i testy nebo ukázky použití funkcí.

Sám autor v README zmiňuje, že síla Tiny AES není v rychlosti, ale v minimálních režiích pamětí, což z ní činí vhodnou implementaci pro případy užití, kde je limitovaná paměť. Veřejný repozitář obsahuje několik souborů, včetně CMakeLists souboru pro snadnou kompilaci zdrojového kódu pomocí nástroje CMake[4] a testů. Rozhraní knihovny je velmi strohé, nabízí dohromady pouze 8 funkcí.

[1] Zdrojové soubory jsou dostupné v github repozitáři https://github.com/kokke/tiny-AES-c
[2] https://gist.github.com/erev0s
[3] https://gist.github.com/erev0s/4c13a5173b607bcea2a1f273981ccdfc#file-pkcs7_padding-c
[4] Open-source nástroj užívaný při kompilaci nebo testování, dostupný na adrese https://cmake.org/

4.2.2 Použití knihovny OpenSSL

Implementace využívá prostředků knihovny OpenSSL. K integraci do měřicího kódu je třeba důkladně prostudovat dokumentaci, neboť kvůli poměrně nedávné změně API je většina ukázek použití online nevhodná.

OpenSSL je open-source projekt pracující na knihovně implementací protokolů *SSL* a *TLS*. I když jsou funkce knihovny psány v jazyce C, mohou být využívány ve velkém množství dalších programovacích jazyků díky tzv. *wrapperům*, tedy mezivrstvou mezi funkcemi OpenSSL v C a kódem v jiném jazyce. OpenSSL je dostupná pro většinu Unixových systémů (včetně Linuxu, macOS nebo BSD) a Microsoft Windows.

Velkým problémem při použití této knihovny byla skutečnost, že demonstrační příklady jejího užití v C jsou poměrně vzácné a většina z nich je buď nefunkční, nebo zcela nevhodná. Dalším byl fakt, že po zveřejnění příkladného užití knihovny se rozhodli tvůrci OpenSSL změnit rozhraní pro práci s knihovnou. Bylo tedy nutno tento kód vzít a upravit podle aktuální dokumentace OpenSSL.

4.2.3 Luo Peng AES 128

Autorem implementace pojmenované jednoduše *aes_128* je Luo Peng[5], student Čínské akademie věd. Jeho Github repozitář[6] kromě implementace vyhledávací tabulky, testů, Makefile a hlavních zdrojových souborů *aes.c* a *aes.h*, obsahuje i soubor *main.c*, který demonstruje užití funkcí.

Integrace implementace do měřicího kódu byla jednoduchá a přímočará. Stačilo do měřicího kódu přesunout soubory *aes.c* a *aes.h*, ty již nemají, kromě standardních knihoven, žádné další závislosti. Kód obsahuje poměrně dost komentářů jak u pomocných funkcí, tak u hlavních funkcí rozhraní, díky čemuž se v něm jednoduše orientuje a je snadno pochopitelný. Navíc si díky kontrolním tiskům bloku dat před šifrováním, po šifrování a po dešifrování lze velmi snadno ověřit jeho funkčnost.

4.3 Výsledky měření

Cílem měření bylo zjistit, s jakou rychlostí jsou schopny vybrané implementace z předchozí podkapitoly zašifrovat a dešifrovat zprávy. Jakožto veličina pro srovnání byla zvolena bitová rychlost, která vyjadřuje množství zpracovaných informací za vteřinu. Měření probíhalo pomocí kódu v jazyce C. Během jednoho běhu probíhá vždy zašifrování a dešifrování předem definované zprávy, těchto běhů je ve výchozím nastavení 5 000, výsledky v tabulce jsou aritmetickým průměrem hodnot odpovídajících algoritmů a délky zprávy. Algoritmus pracuje s několika délkami zpráv: 128 bitů, 512 bitů a 2048 bitů. Inicializace potřebných struktur, generování klíče či volání funkcí má totiž své režie. To se nejvíce projeví u šifrování krátkých zpráv, kde tyto činnosti zaberou podstatnou část celkové doby vykonání kódu. Proto pro získání lepšího přehledu a přesnějších výsledků měření probíhá s několika délkami zpráv. Výstup měřicího programu byl zpracován do tabulky 4.12.

Z tabulky lze vyčíst, že knihovna OpenSSL podává vynikající výsledky. Už při práci se 128 bitů dlouhými zprávami je schopna zašifrovat a dešifrovat bloky dat s rychlostí 160 MB/s, a to je nezanedbatelně zatížena režiemi volání funkcí a zbylými náklady měřicího kódu. Dopad těchto faktorů lze redukovat tím, že jako vstup algoritmu je použita delší

[5] Github profil: https://github.com/openluopworld

[6] Dostupný na adrese https://github.com/openluopworld/aes_128

Název implementace	Čas publikace	Poslední aktivita	Popularita	Průměrný bitrate [kB/s]		
				128b zpráva	512b zpráva	2048b zpráva
tiny-AES	01.02.2019	01.07.2020	2.3k stars, 890 forks, 128 watched	2 052	2 038	2 091
OpenSSL	23.12.1998	20.11.2020	14.3k stars, 6.4k forks, 908 watched	160 000	533 333	711 111
Luo Peng	01.09.2017	04.12.2020	81 stars, 45 forks, 12 watched	28 144	29 029	28 269

Obrázek 4.12: Výsledná tabulka měření 3 implementací algoritmu AES.

zpráva. S čtyřikrát delší 512 bitovou zprávou se bitová rychlost algoritmu zvýší více než 3x. Při těchto parametrech se průměrná bitová rychlost pohybuje okolo 533 MB/s, ve které se již více promítla efektivita samotného šifrování a dešifrování. Pokud použijeme ještě čtyřikrát delší zprávu o celkové délce 2048 bitů, dosáhneme ještě 1,3x lepšího výsledku o hodnotě 711 MB/s.

Dále lze z tabulky zjistit, že bitová rychlost není u Tiny AES prioritou. Jak její tvůrci avizovali, algoritmus pracoval se všemi třemi délkami zpráv se zhruba stejnou bitovou rychlostí okolo 2 050 kB/s. Zajímavé na tomto výsledku je to, že při změně zprávy ze 128 bitů na 512 bitů se bitová rychlost algoritmu zmenšila o 14 kB/s. Obecně platí, že s délkou zprávy rychlost šifrovacího algoritmu roste, přičemž v tomto měření klesá. Rychlost pro nejdelší 2048 bitovou zprávu je dle předpokladů nejvyšší ze všech zaznamenaných, s bitovou rychlostí 2091 kB/s – pouze 1,02x větší rychlost oproti práci se 128 bitovými zprávami. V porovnání s knihovnou OpenSSL je však výrazně pomalejší, při šifrování nejkratších 128 bitových zpráv je 78x pomalejší, pro ty nejdelší zprávy o 2048 bitech je dokonce 340x pomalejší.

Poslední měřenou implementaci její autor Luo Peng jednoduše pojmenoval *aes_128*. Ač její tvůrce upozorňuje, že rychlost není hlavní prioritou jeho práce, bitová rychlost jeho implementace se pro 128 bitové, 512 bitové i 2048 bitové pohybuje mezi 28 a 29 MB/s. Podobně jako u Tiny AES, i zde se rychlost mění v rozporu s očekáváním. Při přechodu z 128 bitové zprávy na 512 bitovou se bitová rychlost zvýší o 885 kB/s, přičemž vzápětí s prodloužením zprávy na 2048 bitů jeho rychlost klesá o 760 kB/s. I tak je v průměru více než 14x rychlejší než Tiny AES, oproti knihovně OpenSSL je však pro krátké zprávy 5,7x pomalejší, pro zprávy délky 2048 bitů je až 25x pomalejší.

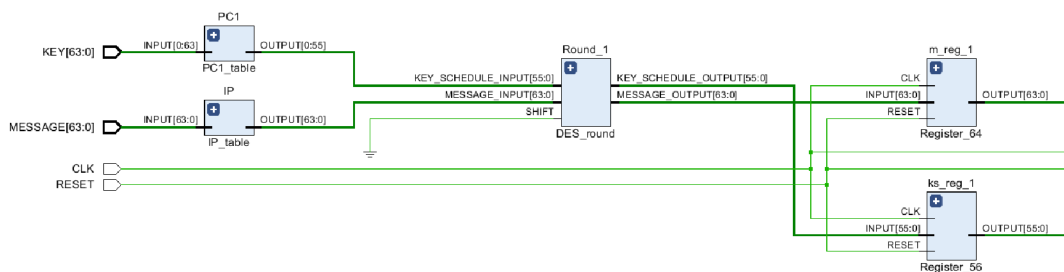
Kapitola 5

Hardwarová implementace DES

Celý postup algoritmu DES lze rozdělit na několik kroků. Dílčí kroky jsou poté poměrně jednoduše hardwarově implementovatelné, čemuž napomáhá skutečnost, že jedním z požadavků na tento algoritmus byla právě možnost jednoduché a rychlé hardwarové implementace. Každý z těchto podkroků potom lze ve výsledné implementaci popsat pomocí jazyka VHDL. Mnoho základních komponent algoritmu obsahuje jen velmi málo logiky, jejich ruční implementace by byla velmi pracná a představovala by velké riziko vzniku chyby. Proto jsem raději vytvořil kód v jazyce C#, který vygeneroval potřebné permutační a substituční tabulky, včetně testovací entity. Tímto vznikly komponenty tabulek E-bit selection (3.8), IP (3.6), inverzní IP (3.11), PC1 (3.3), PC2 (3.4), P (3.10) a všech 8 S-boxů (C).

5.1 Implementace šifrování

Během procesu šifrování lze využít postupu zřetězeného zpracování (anglicky *pipelining*), který pomocí umístění registrů mezi jednotlivá kola výpočtu výrazně zvýší bitovou rychlost implementace. Postupu jsem využil tak, že v jednom kole současně algoritmus vytvoří nový podklíč, který rovnou použije pro zašifrování bloku. Jak výsledek šifrování, tak mezivýsledek a vstup dalšího kola expanze klíče jsou uloženy v registrech, které synchronizuje hodinový signál. V jednu chvíli tak hardware může šifrovat až 16 různých zpráv.

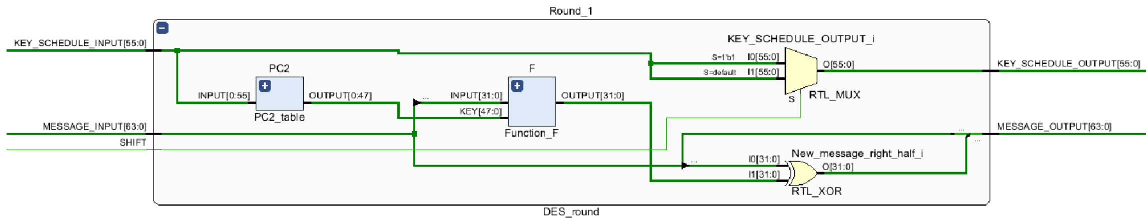


Obrázek 5.1: Schéma bodu, kde do komponenty vstupuje v levé části klíč a zpráva. Vstupy se podrobí odpovídající permutaci a vstupují do kola šifrování, po kterém jsou uloženy v registrech v pravé části.

- **Vstupní část komponenty hlavní úrovně** je vyobrazena ve schématu 5.1. Vstupy v levé části tvoří kromě $RESET$ a hodinového CLK signálu také signál KEY, který je vstupním tajným klíčem algoritmu. Dále vstupuje do komponenty signál $MESSAGE$,

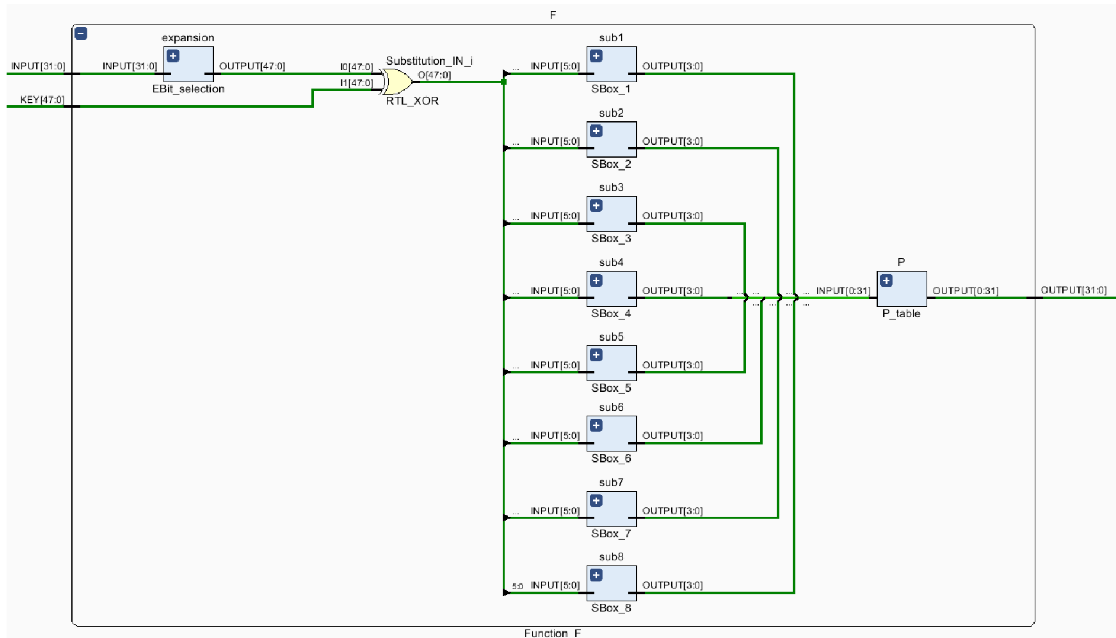
který reprezentuje šifrovaný vstupní blok. Tajný klíč nejprve vstoupí do komponenty $PC1$, kde je na něj aplikována permutační tabulka, ze které vystoupí již jen 56 bitů klíče bez paritních bitů. Vstupní blok zprávy se také podrobí permutaci v komponentě IP, a spolu s permutovaným klíčem a informací o posunu $SHIFT$ vstupují do prvního kola šifrování. Hodnota signálu $SHIFT$ je daná specifikací a neřídí se žádnou další logikou. Výstupy komponenty kola šifrování jsou blok zprávy a mezivýsledek expanze klíče a oba tyto signály jsou uloženy do registrů řízených hodinovým signálem. Výstupy registrů opět vedou do dalšího kola šifrování, čímž aplikují zřetězené zpracování. Znázorněná komponenta hlavní úrovně se skládá z dalších, menších komponent.

- **Iterace výpočtu** je popsána schématem v obrázku 5.2. Zde vlevo do komponenty vstupují mezivýsledky z registrů expanze klíče $KEY_SCHEDULE_INPUT$ a šifrované zprávy $MESSAGE_INPUT$, spolu s informací o posunu $SHIFT$. Mezivýsledek expanze klíče vstupuje do komponenty aplikující permutaci řízenou tabulkou $PC2$, jejíž výstup je novým podklíčem pro současnou iteraci. Nový podklíč spolu s mezivýsledkem šifrování zprávy vstupuje do komponenty reprezentující funkci f, jejíž výstup se podrobuje operaci XOR s levým blokem zprávy. Pravý blok zprávy se stává novým levým blokem zprávy a spolu s výstupem hradla XOR, jež dalo za vznik nové pravé polovině zprávy, tvoří po jejich spojení výstup komponenty iterace šifrování. Tento výstup je poté uložen do registru pro mezivýsledky šifrování vstupní zprávy. Vstupní signál hodnoty expanze klíče je také rozdělen na dvě poloviny a obě jsou posunuty o jeden, nebo dva bity vlevo. Právě zde hraje roli hodnota signálu $SHIFT$, která určuje, o kolik bitů se hodnoty posunou.



Obrázek 5.2: Schéma obvodu jednoho kola šifrování. V levé části vstupuje výsledek expanze klíčů spolu s šifrovaným blokem a informací, o kolik má být blok podklíče posunut. Vpravo se nachází jeho výsledky.

- **Funkce *f*** je schematicky popsána na obrázku 5.3. Zde v levé části do obvodu vstupuje pravý blok zprávy $INPUT$ a podklíč daného kola KEY. Signál bloku $INPUT$ o délce 32 bitů se podrobí permutaci řízené tabulkou *Ebit_selection*, jejíž výstupem je signál o délce 48 bitů. Tento výstup spolu s podklíčem vstupují do hradla XOR a výsledek se rozdělí na 8 úseků o délce 6 bitů. Pro každou šestici bitů existuje S-box daný specifikací, do jejichž komponent šestice vstupují. Čtveřice výstupních bitů S-boxů jsou poté znovu spojeny v jeden signál o délce 32 bitů, který je podroben permutaci řízené tabulkou P. Výsledek permutační komponenty P je zároveň výstupem funkce f.



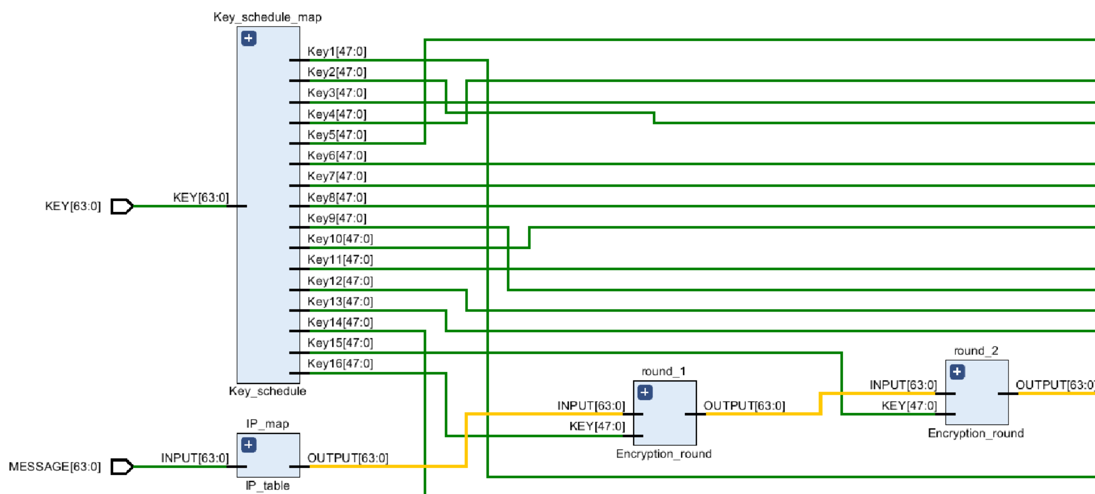
Obrázek 5.3: Schéma komponenty reprezentující funkci *f*. V levé části vstupuje klíč spolu s pravou polovinou šifrovaného bloku, poté se podrobuje permutacím, operaci XOR a substituci, než je v pravé části vystaven na výstup.

5.2 Implementace dešifrování

I když je využití zřetězeného zpracování velmi výhodné, bohužel ne vždy je jednoduché jej v návrhu využít. Právě při dešifrování zprávy algoritmem DES vzniká problém, poněvadž pro první kolo dešifrování je potřeba znát poslední podklíč. Kvůli tomu je nutné nejprve projít celým procesem expanze klíčů, abychom až poté mohli začít s dešifrováním. Implementace dešifrování algoritmem DES je tak pouze jednodušší, nezřetězený synchronní obvod.

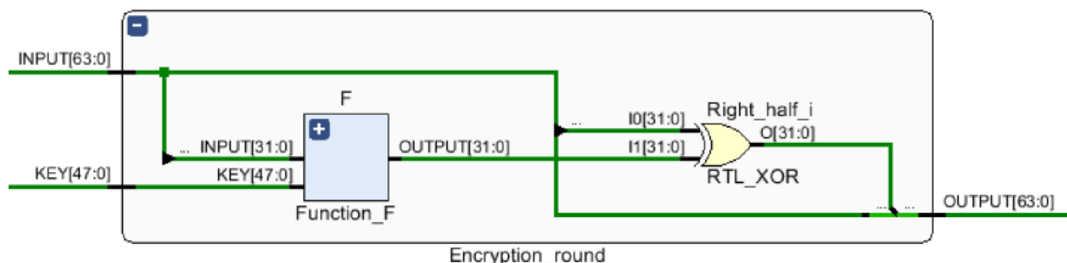
- **Vstupní část komponenty hlavní úrovně** je popsána schématem 5.4. Vlevo zde vstupují do entity dva signály o délce 64 bitů, jeden pro zprávu $MESSAGE$ a druhý pro tajný klíč KEY. Protože pro dešifrování je třeba v prvním kole poslední expandovaný klíč, vstupuje ihned signál KEY do komponenty *Key_schedule*, která má na svém výstupu 16 signálů reprezentující 16 expandovaných podklíčů. Signál $MESSAGE$ se vstupním blokem zprávy se nejprve podrobuje permutaci tabulkou IP, jejíž výstup tvoří spolu s posledním 16. podklíčem vstup komponenty iterace dešifrování *Encrytion_round*. Komponent kol dešifrování je v návrhu dohromady 16 a narozdíl od implementace šifrování, v tomto návrhu se mezi jednotlivými koly dešifrování nenachází registry pro uchování mezivýsledků.

- **Kolo dešifrování** je vyobrazeno ve schématu 5.5. Zde v levé části do obvodu vstupuje podklíč KEY pro dané kolo dešifrování a vstupní blok $INPUT$ zašifrované zprávy. Podklíč, spolu s pravou polovinou dešifrovaného bloku vstupují do entity funkce *f*, jejíž výstup, stejně jako u šifrování, se podrobí operaci XOR s levou polovinou dešifrovaného bloku. Signál vedoucí z hradla XOR tvoří novou pravou polovinu výsledného bloku a před vystavením na výstup komponenty kola $OUTPUT$ je spojen s pravou



Obrázek 5.4: Schematický popis první části komponenty. V levé části vstupuje klíč do obvodu pro expanzi klíče na 16 podklíčů, dále také vstupní zpráva vstupuje do permutační tabulky. Cesta bloku zprávy je žlutě vyznačena a vede mezi 16 koly.

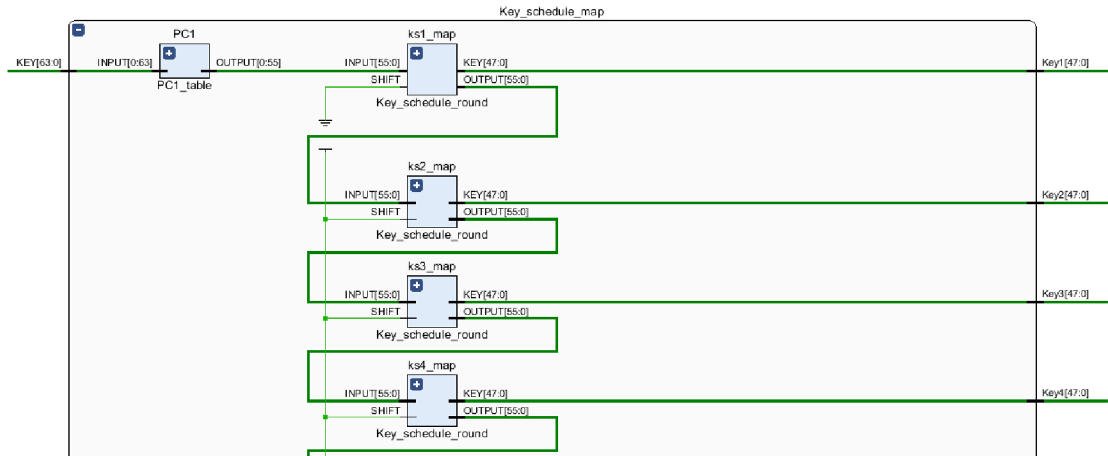
polovinou původního bloku. Pravá polovina původního bloku zprávy se tedy stává novou levou polovinou výstupního signálu.



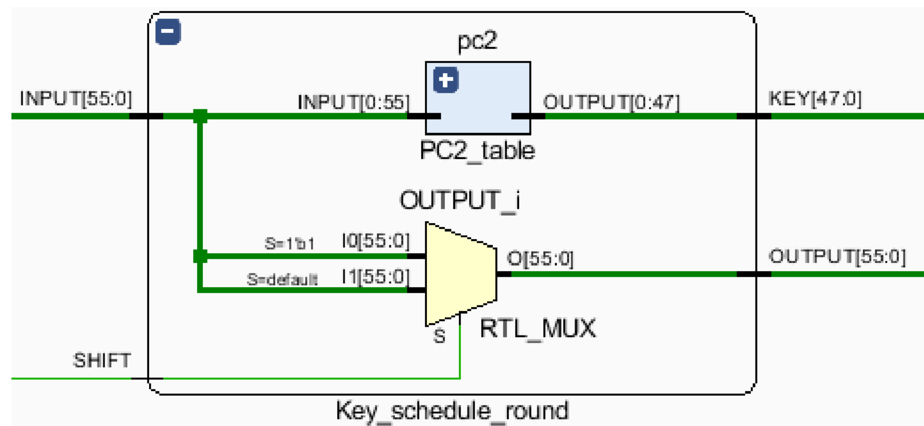
Obrázek 5.5: Schéma obvodu reprezentující kolo dešifrování. Vpravo vstupuje blok spolu s podklíčem, vpravo se nachází výstup.

- **Komponenta expanze klíče** popsaná schématem 5.6 se skládá z jedné permutační tabulky $PC1$ a 16 komponent $Key_schedule_round$. Vstupní tajný klíč KEY se zde podrobuje permutaci řízené tabulkou $PC1$, jejíž výstup se stává vstupem komponenty prvního kola. Výstupy každého kola jsou dva: jeden nový podklíč a jeden mezivýsledek, který vstupuje do následujícího kola expanze podklíče. 16 nově získaných podklíčů $Key1$, $Key2$ až $Key16$ tvoří výstupy komponenty v pravé části.

- **Kolo expanze klíče** je součástí hiearchicky nadřazené komponenty expanze klíče. Ze schématu 5.7 lze vyčíst, že tato komponenta ze svého vstupního signálu $INPUT$ aplikací permutační tabulky $PC2$ vystavuje na svůj výstup KEY nově vytvořený podklíč. Signál $INPUT$ také dále rozpůlí na dvě stejně dlouhé části a bitově posune o jeden, nebo dva bity. Hodnotu bitového posunu určuje vstupní signál $SHIFT$, jehož hodnota je pevně dána specifikací algoritmu. Nově vzniklá posunutá hodnota je poté vystavena na výstupní signál komponenty $OUTPUT$.



Obrázek 5.6: Část schématu komponenty zajišťující expanzi vstupního klíče v levé části na 16 podklíčů v pravé části. Zobrazeny jsou pouze první čtyři kola.



Obrázek 5.7: Popis fungování kola expanze klíče. Vlevo vstupuje blok, po jehož permutaci vznikne nový podklíč. Vstupní blok je zároveň posunut o jeden, nebo dva bity a vystaven na výstup v pravé části.

Z nákresů lze vyčíst, že celá implementace algoritmu se skládá z jednoduchých komponent: hradel XOR, multiplexorů, substitučních a permutačních tabulek a registrů. Substituční i permutační tabulky lze ve VHDL popsat jakožto paměť, kde je vstupní hodnota využita jako adresa paměti, samotné hodnoty v paměti jsou jejich náhradami. Všechny tyto základní komponenty jsou spojeny v jeden velký blok, který obstarává komunikaci mezi CPU a FPGA. Výstupní signály komponent hlavní úrovně vedou do zapisovacích blokových pamětí přítomných na čipu a z dalších čtecích blokových pamětí získávají tajné klíče a vstupní bloky zprávy.

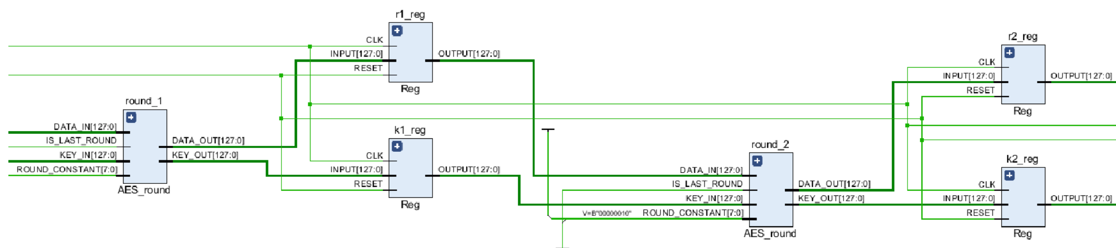
Kapitola 6

Hardwarová implementace AES

Při výběru AES byl kladen důraz na možnost algoritmus jednoduše a efektivně hardwarově implementovat, což se také projevilo. Základní komponenty obsahující jen velmi málo logiky, jejichž tvorba by byla časově náročná a náchylná na chyby, byly vygenerovány pomocí programu v jazyce C#. Tím byly strojově vygenerovány entity a architektury Rijndael S-boxu 4.7, inverzního S-boxu 4.10 a testovací entity komponenty hlavní úrovně, která obsahuje 277 testů užitých při testování šifrování i dešifrování.

6.1 Implementace šifrování

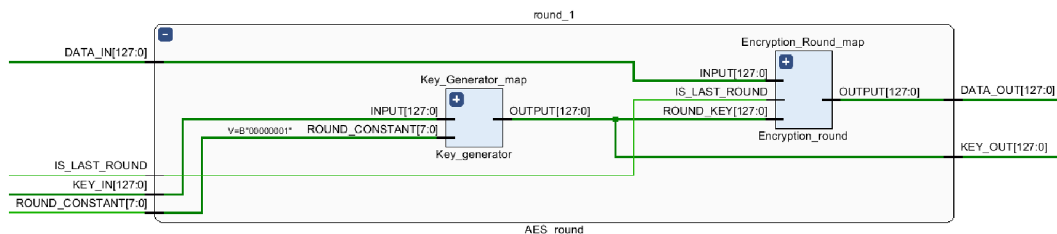
V návrhu obvodu šifrování AES lze využít techniky zřetězeného zpracování (anglicky *pipelining*). Techniku jsem v implementaci využil tak, že jsem mezi jednotlivé komponenty reprezentující iterace výpočtu zapojil registry řízené společným hodinovým signálem CLK a signálem $RESET$. V každé iteraci je vygenerován nový podklíč, který je ještě v té samé entitě použit pro zašifrování bloku zprávy. Výsledek expanze klíče i mezivýsledek šifrování je poté uložen do registrů. Díky zřetězenému zpracování jsem docílil velmi výrazného zvýšení bitové rychlosti a obvod může v jednom hodinovém cyklu šifrovat až 10 zpráv.



Obrázek 6.1: Schematický popis návaznosti jednotlivých kol šifrování na registry umožňující zřetězené zpracování.

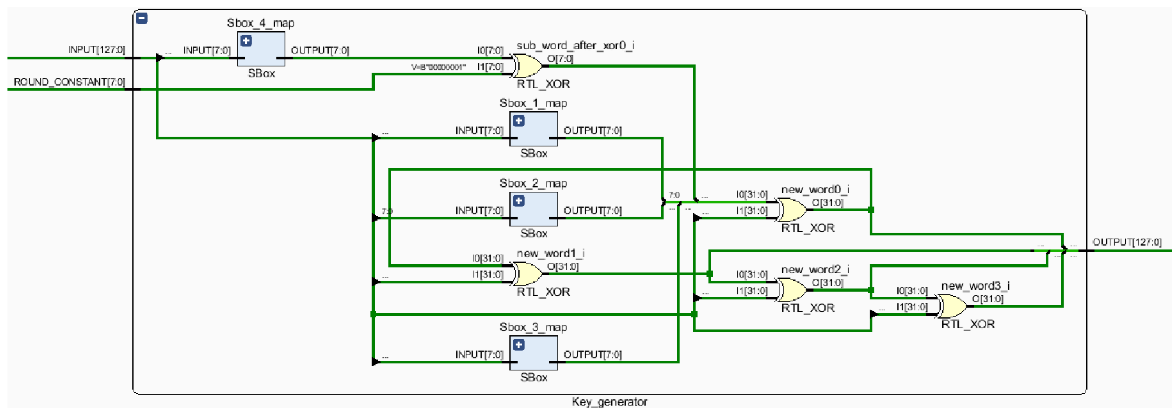
- **Propojení kol a registrů v komponentě hlavní úrovně** je schématicky popsáno na obrázku 6.1, zde jsou mezi komponentami reprezentující iterace výpočtu *AES_round* umístěny registry *Reg* pro uchování mezivýsledků řízené společným hodinovým signálem CLK a signálem $RESET$. V levé části vstupuje do prvního kola podklíč, blok zprávy, konstanta kola a informace, zda se jedná o poslední kolo. Výstup kola je poté uložen v registrech pro klíč a pro zprávu, odkud opět vstupuje do dalšího kola šifrování.

- **Iterace šifrování** je ilustrována pomocí schématu 6.2, kde v levé části do entity vstupuje šifrovaný blok $DATA_IN$, podklíč z předchozího kola KEY_IN, konstanta kola $ROUND_CONSTANT$ a signál IS_LAST_ROUND nesoucí informaci, zda se jedná o poslední iteraci výpočtu. Podklíč KEY_IN vstupuje spolu s konstantou kola do komponenty $Key_generator$, která na výstup vystaví nově vygenerovaný podklíč. Nový podklíč poté vstoupí spolu s blokem zprávy a signálem IS_LAST_ROUND do komponenty $Encryption_round$, která provede kolo šifrování s daným podklíčem. Výstup $Encryption_round$ slouží jako výstupní signál komponenty $DATA_OUT$, stejně tak i výstupní signál entity $Key_generator$ je zároveň výstupem iterace šifrování KEY_OUT. Hodnoty obou výstupních signálů jsou poté uloženy do registrů kvůli zřetězenému zpracování.



Obrázek 6.2: Popis jednoho kola šifrování. Vlevo vstupuje klíč, blok zprávy, konstanta kola a informace, zda se jedná o kolo poslední. Uvnitř bloku poté proběhne jak expanze klíče, tak i jeho okamžitá aplikace v šifrování. V pravé části z bloku vystupuje klíč a blok zprávy.

- **Kolo expanze klíče** popsáno schématem 6.3 přijímá na svém vstupu v levé části signál $INPUT$, který nabývá hodnoty podklíče z předchozí iterace spolu s konstantou kola $ROUND_CONSTANT$. Uvnitř entity se úseky vstupního bloku $INPUT$ podrobují substitucím a operacím XOR, přesně podle obrázku 4.6. Výstupem kola expanze klíče je signál $OUTPUT$, který nabývá hodnoty nového podklíče.

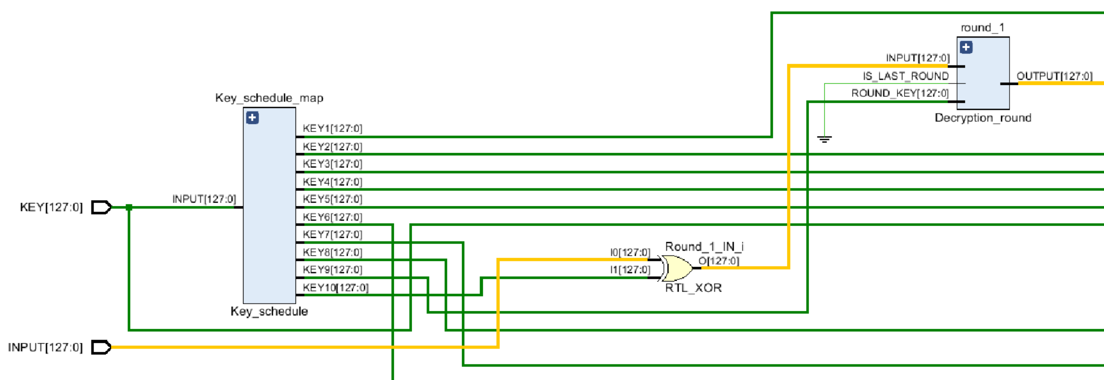


Obrázek 6.3: Komponenta generující podklíč, do níž vlevo vstupuje předešlý podklíč a konstanta kola. Uvnitř bloku proběhne proces jeho expanze a nový podklíč je vpravo vystaven na výstup.

6.2 Implementace dešifrování

Pro hardwarový návrh dešifrování je užití techniky zřetězeného zpracování obtížnější, poněvadž pro první kolo dešifrování je potřeba poslední vygenerovaný podklíč. Proto je nutno nejprve provést celý krok expanze klíče ještě před prvním kolem. Implementace dešifrování algoritmu AES je tedy nezřetězený synchronní obvod.

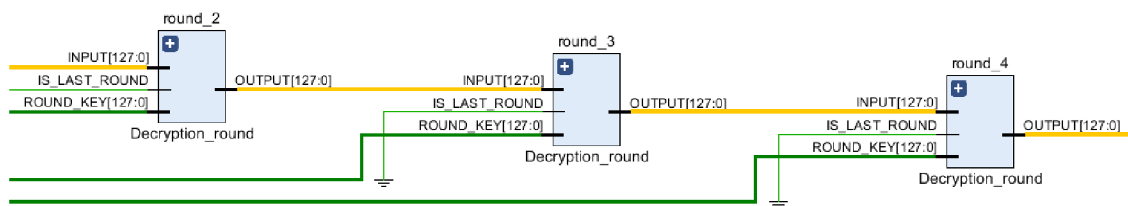
- **Vstupní bod komponenty hlavní úrovně dešifrování** je zobrazen ve schématu 6.4. V levé části se nachází jeho vstupní signál KEY pro tajný klíč, vede do komponenty $Key_schedule$. Tato komponenta obstará expanzi tajného klíče a na svých výstupních 10 signálů vystaví odpovídajících 10 podklíčů, které poté dále vstupují do kol dešifrování. Vstupní signál $INPUT$ reprezentuje zašifrovaný blok zprávy, ten se podrobuje operaci XOR s 10. podklíčem a výsledek slouží jako vstup komponenty kola dešifrování.



Obrázek 6.4: Schéma bodu, kde do hlavní komponenty vstupuje zpráva a klíč. Klíč je ihned využit v obvodu pro expanzi klíče, jejíž 10. podklíč vstupuje do operace XOR se zprávou. Výsledek operace XOR poté funguje jako vstup kola dešifrování.

- **Návaznost kol dešifrování** je popsána schématem 6.5. Zde do každého kola vstupuje odpovídající podklíč, dešifrovaný blok a informace, zda se jedná o kolo poslední. Komponenta kola na svůj vstupní signál $INPUT$ aplikuje operace inverzního posunu řádků, inverzní substituce bajtů, podrobí jej operaci XOR s podklíčem a nakonec provede inverzní promíchání sloupců. Pokud je hodnota signálu indikujícího, že se jedná o poslední kolo kladná, pak je krok promíchání sloupců vynechán.

Z výše popsaných návrhů lze zjistit, že pro hardwarovou implementaci algoritmu AES stačí jen registry, substituční a permutační tabulky, logická hradla XOR a multiplexory. Substituční i permutační tabulky jsou implementovány jako paměti, kde vstupem paměti je adresa a výstupem je hodnota na této adrese. Všechny tyto jednotlivé komponenty jsou poté hiearchicky uspořádány do nadřazeného bloku, který obstarává komunikaci mezi CPU a FPGA. Výsledná šifrovací i dešifrovací komponenta hlavní úrovně je poté schopna komunikovat pomocí čtení vstupních dat a zápisu výstupů do blokových paměti, které jsou sdílí s procesorem.



Obrázek 6.5: Popis návaznosti jednotlivých kol dešifrování. Vlevo vstupuje do kola blok zprávy a je dále žlutě vyznačen. Do každého kola také vstupuje odpovídající podklíč a informace, zda se jedná o kolo poslední.

Kapitola 7

Výsledky

V následujících sekcích jsou popsány výsledky mé bakalářské práce. K nim patří syntetizovatelný obvod popsaný jazykem VHDL pro šifrování i dešifrování algoritmů AES a DES, kód v jazyce C pro měření bitové rychlosti porovnávaných implementací a kód v jazyce C#[1], jenž byl vytvořen pro generování VHDL popisu pro substituční a permutační tabulky a testovacích entit. Pro odvození bitových rychlostí byly využity výpisy časování syntetizovaných obvodů programu Vivado, informace o využití zdrojů poskytují výpisy využití.

7.1 Implementace DES

Jak již bylo zmíněno v kapitole 5, pro proces šifrování bylo využito techniky zřetězeného zpracování, pro dešifrování však nikoliv. Do prvního kola dešifrování totiž vstupuje poslední expandovaný podklíč, což užití této techniky ztěžuje. Zynqberry samotný v simulaci pracuje s periodou 20 ns, tedy s frekvencí 50 MHz.

Šifrování, díky využití zřetězeného zpracování, je schopno během této periody vytvořit nový klíč a rovnou ho i uplatnit v iteraci šifry. Z reportu v příloze D lze zjistit, že nejdelší cesta v návrhu je z čtecí paměti do prvního registru uchovávající první mezivýsledek, která trvá 5,210 ns. Díky skutečnosti, že perioda simulačních hodin Zynqberry je 20 ns, je schopna na přípravku Zynqberry implementace pracovat na maximální frekvenci 50 MHz, čímž by dosáhla bitové rychlosti 3,2 Gb/s. Touto rychlostí překonává všechny měřené softwarové implementace. Pokud bychom chtěli získat maximální dosažitelnou rychlost zkrácením periody hodin na zpoždění cesty 5,210 ns, měli bychom být teoreticky schopni dosáhnout bitové rychlosti 12,28 Gb/s.

Implementace šifrování DES na čipu zabírá 2983 vyhledávacích tabulek (anglicky *LUT*), tedy 16,95 % z maximálních dostupných. Registrů využívá 3939, čímž využívá pouze 11,19 % ze všech volných registrů (anglicky *Flip-flops*, dále zkratka *FF*). O podrobnějším popisu využití zdrojů se lze dozvědět více v příloze E.

Proces dešifrování nevyužívá zřetězeného zpracování, proto trvá cesta ze čtecí paměti do zapisovací paměti až 36,272 ns, oproti frekvenci Zynqberry se opožďuje o 16,272 ns. Na hardwarovém přípravku tedy nesplňuje časování a dešifrovací komponenty musí být od Zynqberry asynchronně odděleny. Z toho vyplývá, že navržený obvod je schopen pracovat na frekvenci 27 569 kHz, tedy zhruba 27 MHz. Obvod je schopen dešifrovat s bitovou rychlostí až 1 764 Mb/s, čímž i tak překonal všechny měřené softwarové implementace. Podrobnosti o časování se nachází v příloze F.

[1] https://docs.microsoft.com/cs-cz/dotnet/csharp/

Komponenta pro dešifrování DES na čipu zabírá dohromady 2597 LUT a 2083 FF registrů. Z celkových LUT čipu využívá 14,76 %, z dostupných FF registrů využije díky absenci zřetězeného zpracování pouze 5,92 %. Další informace o využití zdrojů jde nalézt v příloze G.

7.2 Implementace AES

Proces šifrování využívá zřetězeného zpracování, což významně zkracuje trvání nejdelší cesty, jak se lze dočíst ze zprávy v příloze H. Z výpisu vyplývá, že nejdelší cesta trvá pouze 6,285 ns, čímž je algoritmus schopen pracovat na simulační frekvenci čipu 50 MHz. Na této frekvenci dosahuje bitové rychlosti 6,4 Gb/s, čímž lehce překoná všechny měřené softwarové implementace. Pokud bychom periodu hodin zkrátili na 6,285 ns, tedy frekvenci 159 MHz, mohli bychom dosáhnout bitové rychlosti 20,366 Gb/s. Knihovní funkce OpenSSL pracují bitovou rychlostí 711 Mb/s, což je 20,5x nižší bitová rychlost, než jaké může dosáhnout můj obvod.

Implementace šifrování AES na Zynqberry čipu zabírá 12715 LUT, tedy 72,24 % zdrojů. Dále také využívá 2083 FF registrů, což činí 5,92 % ze všech dostupných registrů. Podrobné informace o využití zdrojů se nachází v příloze I.

Implementace dešifrování bloků algoritmem AES neobsahuje zřetězené zpracování, jeho bitová rychlost je tak o poznání nižší. Ze zprávy o časování návrhu v příloze J lze zjistit, že cesta procházející obvodem z čtecí do zápisové paměti trvá 52,740 ns, čímž znatelně překračuje periodu simulačních hodin Zynqberry o délce 20 ns. Je tedy nutno dešifrovací komponenty od Zynqberry asynchronně oddělit. Můj návrh je tedy schopen pracovat na nižší frekvenci 18,96 MHz, na které je schopen dešifrovat zprávy s rychlostí 2,427 Gb/s. I s takovou hodnotou je však stále výrazně rychlejší, než všechny měřené softwarové implementace. V porovnání se 711 Mb/s knihovny OpenSSL je dešifrovací obvod 3,4x rychlejší.

Ze zprávy Vivado simulátoru v příloze K lze zjistit, že návrh zabírá 14162 vyhledávacích tabulek LUT, což činí 80,47 % z tabulek přítomných na čipu. Ke svému běhu dále využívá 2083 FF registrů, tedy 5,92 % ze všech dostupných.

7.3 Testování

Po provedení implementace jsem se zaměřil na detailní ověření funkce vytvořených komponent. K tomu jsem vytvořil program v jazyce C#, který ze vstupního souboru testovacích vektorů vygeneroval testovací entity pro šifrování i dešifrování DES a AES. VHDL návrh algoritmů je tedy otestován v simulaci pomocí vygenerovaných entit, které výstup VHDL popisu zapisují do textových souborů. Ke kontrole obsahu a správnosti jsem poté vytvořil další konzolovou aplikaci v jazyce C#, která výstupní textové soubory porovná s očekávanými výsledky a případné chyby zaznamená na výstupu. Návrhy AES byly otestovány s vyžitím testovacích vektorů NIST[5][4], jako zdroj testů algoritmu DES posloužily testy z repozitáře *PHP Quality Checker*[3].

Kapitola 8

Závěr

Cílem práce bylo navrhnout hardwarovou akceleraci algoritmů DES a AES pro přípravek Xilinx Zynqberry a tento cíl se podařilo splnit.

Ke splnění úkolů jsem se musel důvěrně seznámit jak s algoritmem DES, tak i AES. Dále jsem si musel připomenout principy a syntax jazyka VHDL a zjistit, jak pracovat s přípravkem Xilinx Zynq a programem Vivado. Stejně tak jsem se seznámil s jeho vnitřní architekturou, abych mohl Zynqberry využívat efektivně a s pochopením. Výsledky studia kryptografických šifer se nachází v kapitole 2, kapitola 3 pojednává o algoritmu DES a kapitola 4 obsahuje informace o algoritmu AES.

V další fázi jsem implementoval návrhy obvodů v jazyce VHDL, jež jsou schopny šifrovat i dešifrovat zprávy pomocí algoritmů DES a AES. Návrhy šifrování obou algoritmů využily techniky zřetězeného zpracování, díky čemuž splňují podmínky časování simulačních hodin přípravku Zynqberry[1]. Implementace dešifrovacích komponent této techniky však nevyužívají, a tak se oproti periodě hardwaru zpožďují a maximální frekvence nedosahují. Kapitola 5 se věnuje hardwarovému návrhu šifry DES, o implementaci AES obsahuje více informací kapitola 6.

Pro získání lepšího srovnání jsem také vybral tři volně dostupné implementace algoritmů DES i AES v jazyce C, ty integroval to měřicího kódu a jejich rychlost změřil. Zjistil jsem tak, že mé hardwarové implementace jsou mnohonásobně rychlejší, než integrované softwarové implementace. O výsledných bitových rychlostech hardwarových implementací a jejich srovnání se softwarovými pojednává kapitola 7.

Dále jsem implementoval mimo měřicí kód v jazyce C a VHDL popisu DES a AES i program v jazyce C#, který mi pomohl generovat substituční a permutační tabulky. Program jsem také využil k vygenerování testovací entity, díky které jsem v simulaci ověřil funkčnost implementací. Další program v jazyce C# poté zpracovává a porovnává výstupy VHDL návrhů a automaticky kontroluje správnost hodnot.

Implementace by šlo dále vylepšit o možnosti užití více režimů provozu, aplikaci zřetězeného zpracování pro dešifrovací návrhy či reálné nasazení při řešení praktického problému.

[1] Dostupný na stránkách výrobce https://shop.trenz-electronic.de/en/TE0726-03M-ZynqBerry-Module-with-Xilinx-Zynq-7010-in-Raspberry-Pi-Form-Faktor

Literatura

[1] Delfs, H. a Knebl, H. *Introduction to Cryptography: Principles and Applications*. Springer Berlin Heidelberg, 2007. Information Security and Cryptography. ISBN 9783540492443. Dostupné z: https://books.google.cz/books?id=hNykrv6OZIoC.

[2] Dooley, J. *History of Cryptography and Cryptanalysis: Codes, Ciphers, and Their Algorithms*. Springer International Publishing, 2018. History of Computing. ISBN 9783319904429. Dostupné z: https://books.google.cz/books?id=BfkrtQEACAAJ.

[3] Ferrara, A. *Github - ircmaxell/quality-checker* [online]. 2012 [cit. 2021-05-05]. Dostupné z: https://github.com/ircmaxell/quality-checker/blob/master/tmp/gh_18/PHP-PasswordLib-master/test/Data/Vectors/des.test-vectors.

[4] Gil, D. L. *Github - coruus/nist-testvectors ECBVarKey128.rsp* [online]. 2014 [cit. 2021-05-05]. Dostupné z: https://github.com/coruus/nist-testvectors/blob/master/csrc.nist.gov/groups/STM/cavp/documents/aes/KAT_AES/ECBVarKey128.rsp.

[5] Gil, D. L. *Github - coruus/nist-testvectors ECBVarTxt128.rsp* [online]. 2014 [cit. 2021-05-05]. Dostupné z: https://github.com/coruus/nist-testvectors/blob/master/csrc.nist.gov/groups/STM/cavp/documents/aes/KAT_AES/ECBVarTxt128.rsp.

[6] National Institude of Standards and Technology. *FIPS 81, DES Modes of Operation*. 1980 [cit. 2021-1-25]. Dostupné z: https://csrc.nist.gov/csrc/media/publications/fips/81/archive/1980-12-02/documents/fips81.pdf.

[7] National Institude of Standards and Technology. *FIPS 46-3, Data Encryption Standard(DES)*. 1999 [cit. 2021-1-25]. Dostupné z: https://csrc.nist.gov/CSRC/media/Publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf.

[8] National Institude of Standards and Technology. *FIPS 197, Advanced Encryption Standard (AES)*. 2001 [cit. 2021-1-25]. Dostupné z: https://csrc.nist.gov/csrc/media/publications/fips/197/final/documents/fips-197.pdf.

[9] National Institude of Standards and Technology. *NIST SP 800-38A, Recommendation for Block Cipher Modes of Operation*. 2001 [cit. 2021-1-25]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

[10] National Institude of Standards and Technology. *FIPS PUB 186-4, Digital Signature Standard (DSS)*. 2013 [cit. 2021-1-25]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[11] STALLINGS, W. *Cryptography and Network Security: Principles and Practice.* Prentice Hall, 1999. ISBN 9780138690175. Dostupné z: https://books.google.cz/books?id=Dam9zrViJjEC.

Příloha A

Obsah přiloženého paměťového média

V příloze se nachází strom popisující obsah paměťového média a jeho popis.

Složka *DES* obsahuje VHDL popis jak šifrovacích, tak dešifrovacích komponent algoritmu DES. Obdobně složka *AES* obsahuje komponenty určené k šifrování a dešifrování algoritmem AES. Ve složce *VHDLgenerator* se nachází kód v jazyce C#, jenž byl využit pro generování substitučních a permutačních tabulek v jazyce VHDL, včetně testovacích entit použitých v simulaci. Výstupy této testovací simulace poté ověřuje program, jehož zdrojové soubory jsou umístěny ve složce *Tests*. Ve složce *Benchmarking* se nachází zdrojové kódy v jazyce C, které byly využity pro měření bitové rychlosti softwarových implementací.

```
root
└─ DES
   ├─ EBit_selection.vhd
   ├─ Encryption_round_DES.vhd
   ├─ Function_F.vhd
   ├─ IP_table.vhd
   ├─ IP_table_inv.vhd
   ├─ Key_schedule.vhd
   ├─ Key_schedule_round.vhd
   ├─ PC1_table.vhd
   ├─ PC2_table.vhd
   ├─ P_table.vhd
   ├─ SBox_1.vhd
   ├─ SBox_2.vhd
   ├─ SBox_3.vhd
   ├─ SBox_4.vhd
   ├─ SBox_5.vhd
   ├─ SBox_6.vhd
   ├─ SBox_7.vhd
   ├─ SBox_8.vhd
   ├─ TOP_decryption.vhd
   ├─ DES_round.vhd
   └─ Register_56.vhd
```

```
│   ├── Register_64.vhd
│   ├── TOP_encryption.vhd
│   ├── Test_encryption_DES.vhd
│   └── Test_decryption_DES.vhd
├── AES
│   ├── Decryption_round.vhd
│   ├── Key_generator.vhd
│   ├── Key_schedule.vhd
│   ├── Mix_columns.vhd
│   ├── Mix_columns_inv.vhd
│   ├── SBox.vhd
│   ├── SBox_inv.vhd
│   ├── Shift_rows_inv.vhd
│   ├── AES_top_decrypt.vhd
│   ├── AES_round.vhd
│   ├── Encryption_round_AES.vhd
│   ├── Mix_columns.vhd
│   ├── Reg.vhd
│   ├── Shift_rows.vhd
│   ├── TOP.vhd
│   ├── Test_encryption_AES.vhd
│   └── Test_decryption_AES.vhd
├── VHDLgenerator
│   ├── ComponentGenerator.cs
│   ├── Program.cs
│   ├── TableValueReader.cs
│   └── TestBenchGenerator.cs
├── Tests
│   ├── Program.cs
│   ├── DES_test_1.txt
│   ├── DES_test_2.txt
│   ├── AES_test_encrypt.txt
│   ├── AES_test_decrypt.txt
│   ├── TestRunner.cs
│   ├── TestValues.cs
│   └── Program.cs
└── Benchmarking
    ├── Makefile
    ├── openluopworld_aes.h
    ├── openssl_aes.h
    ├── openssl_des.h
    ├── programmingalgorithms_des.h
    ├── rosetta_des.h
    ├── tiny_aes.h
    ├── openluopworld_aes.c
    ├── openssl_aes.c
    ├── openssl_des.c
    └── programmingalgorithms_des.c
```

```
│
├── rosetta_des.c
├── tiny_aes.c
└── main.c
```

Příloha B

Manuál

Pro spuštění a otestování komponentů je nutno mít nainstalován program Vivado. Zde je potřeba založit nový projekt, který staví na přípravku TE0726-03M. Během tvorby projektu v kroku přidávání zdrojů klikněte na tlačítko *Add File* a vyberte všechny VHDL soubory. Funkcionalitu lze poté jednoduše ověřit tak, že jednu z testovacích entit (soubory začínající řetězcem *Test_*) nastavíte jako komponentu hlavní úrovně a spustíte simulaci. V grafickém výstupu simulace si rychle můžete ověřit funkčnost implementace, případně ke spuštění automatizovaných testů lze použít C# kód ve složce *Tests*. Zde po spuštění do příkazové řádky zadejte cestu ke složce obsahující výstupní soubory generované testovací entitou.

Měřící program v C lze spustit velmi jednoduše pomocí programu CMake. Stačí otevřít složku obsahující zdrojové kódy v příkazové řádce a použít příkaz *make*.

Příloha C

DES S-boxy funkce *f*

V této příloze je vyobrazeno 8 substitučních tabulek využívaných algoritmem DES podle specifikací FIPS[7].

S1															
14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

S2															
15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

S3															
10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

S4															
7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

S5															
2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

S6															
12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

S7															
4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

S8															
13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Příloha D

Výpis časování DES šifrování

V této příloze se nachází výňatek z *Výpisu časování* programu Vivado pro šifrování algoritmem DES s využitím pipeliningu. Z reportu lze vyčíst, že nejdelší cesta vede z čtecí paměti do registru, kde je uložen výsledek prvního kola šifrování. Tato cesta splňuje požadavek periody 20 ns. Podrobný popis této cesty a další cesty, které jí byly velmi podobné, byly z reportu pro přehlednost odstraněny.

```
Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
----------------------------------------------------------------------------

| Tool Version : Vivado v.2019.2 (win64) Build 2708876 Wed Nov 6 21:40:23
   MST 2019
| Date : Fri May 7 03:10:54 2021
| Host : PXDMLI running 64-bit major release (build 9200)
| Command : report_timing_summary -delay_type min_max -report_unconstrained
     -check_timing_verbose -max_paths 10 -input_pins -routable_nets -cells
    [get_cells -hierarchical *TOP_encr*] -name timing_2 -file C:/Users/
    Marek/Desktop/timing_report_through_cell_DES_enc.txt
| Design : design_1_wrapper
| Device : 7z010-clg225
| Speed File : -1 PRODUCTION 1.11 2014-09-11
----------------------------------------------------------------------------


Timing Summary Report

----------------------------------------------------------------------------

| Timer Settings
| --------------
----------------------------------------------------------------------------


  Enable Multi Corner Analysis : Yes
  Enable Pessimism Removal : Yes
  Pessimism Removal Resolution : Nearest Common Node
```

```
  Enable Input Delay Default Clock : No
  Enable Preset / Clear Arcs : No
  Disable Flight Delays : No
  Ignore I/O Paths : No
  Timing Early Launch at Borrowing Latches : No
  Borrow Time for Max Delay Exceptions : Yes
  Merge Timing Exceptions : Yes

  Corner Analyze Analyze
  Name Max Paths Min Paths
  ------ --------- ---------
  Slow Yes Yes
  Fast Yes Yes


------------------------------------------------------------------------------

| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------


    WNS(ns) : 14.372
    TNS(ns) : 0.000
    TNS Failing Endpoints : 0
    TNS Total Endpoints : 5440
    WHS(ns) : 0.037
    THS(ns) : 0.000
    THS Failing Endpoints : 0
    THS Total Endpoints : 5440
    WPWS(ns) : 9.500
    TPWS(ns) : 0.000
    TPWS Failing Endpoints : 0
    TPWS Total Endpoints : 3648


All user specified timing constraints are met.


------------------------------------------------------------------------------

| Clock Summary
| -------------
------------------------------------------------------------------------------


Clock : clk_fpga_0
Waveform(ns) : {0.000 10.000}
```

```
Period(ns) : 20.000
Frequency(MHz) : 50.000


----------------------------------------------------------------------------

| Intra Clock Table
| -----------------
----------------------------------------------------------------------------


Clock : clk_fpga_0
WNS(ns) : 14.372
TNS(ns) : 0.000
TNS Failing Endpoints : 0
TNS Total Endpoints : 3648
WHS(ns) : 0.037
THS(ns) : 0.000
THS Failing Endpoints : 0
THS Total Endpoints : 3648
WPWS(ns) : 9.500
TPWS(ns) : 0.000
TPWS Failing Endpoints : 0
TPWS Total Endpoints : 364

----------------------------------------------------------------------------

| Other Path Groups Table
| -----------------------
----------------------------------------------------------------------------


Path Group : **async_default**
From Clock : clk_fpga_0
To Clock : clk_fpga_0
WNS(ns) : 18.034
TNS(ns) : 0.000
TNS Failing Endpoints : 0
TNS Total Endpoints : 1792
WHS(ns) : 0.347
THS(ns) : 0.000
THS Failing Endpoints : 0
THS Total Endpoints : 1792



----------------------------------------------------------------------------
```

```
| Timing Details
| --------------
-----------------------------------------------------------------------------


From Clock: clk_fpga_0
  To Clock: clk_fpga_0

Setup :
    0 Failing Endpoints
    Worst Slack 14.372ns
    Total Violation 0.000ns
Hold :
    0 Failing Endpoints
    Worst Slack 0.037ns
    Total Violation 0.000ns
PW :
    0 Failing Endpoints
    Worst Slack 9.500ns
    Total Violation 0.000ns
-----------------------------------------------------------------------------



Max Delay Paths
-----------------------------------------------------------------------------

Slack (MET) : 14.372ns (required time - arrival time)
  Source: design_1_i/blk_mem_gen_read/U0/inst_blk_mem_gen/gnbram.
      gnative_mem_map_bmg.native_mem_map_blk_mem_gen/valid.cstr/ramloop[0].
      ram.r/prim_noinit.ram/DEVICE_7SERIES.WITH_BMM_INFO.TRUE_DP.
      SIMPLE_PRIM36.TDP_SP36_NO_ECC_ATTR.ram/CLKBWRCLK
                          (rising edge-triggered cell RAMB36E1 clocked by
                              clk_fpga_0 {rise@0.000ns fall@10.000ns period
                              =20.000ns})
  Destination: design_1_i/TOP_encryption_0/U0/m_reg_1/OUTPUT_reg[0]/D
                          (rising edge-triggered cell FDCE clocked by
                              clk_fpga_0 {rise@0.000ns fall@10.000ns period
                              =20.000ns})
  Path Group: clk_fpga_0
  Path Type: Setup (Max at Slow Process Corner)
  Requirement: 20.000ns (clk_fpga_0 rise@20.000ns - clk_fpga_0 rise@0.000ns
      )
  Data Path Delay: 5.210ns (logic 2.826ns (54.245%) route 2.384ns (45.755%)
      )
  Logic Levels: 3 (LUT2=2 LUT6=1)
  Clock Path Skew: -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD): 1.505ns = ( 21.505 - 20.000 )
```

```
    Source Clock Delay (SCD): 1.700ns
    Clock Pessimism Removal (CPR): 0.050ns
  Clock Uncertainty: 0.302ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter (TSJ): 0.071ns
    Total Input Jitter (TIJ): 0.600ns
    Discrete Jitter (DJ): 0.000ns
    Phase Error (PE): 0.000ns

  -------------------------------------------------------------------
                       required time 21.282
                       arrival time -6.910
  -------------------------------------------------------------------
                       slack 14.372



----------------------------------------------------------------------------

Path Group: **async_default**
From Clock: clk_fpga_0
  To Clock: clk_fpga_0

Setup : 0 Failing Endpoints, Worst Slack 18.034ns, Total Violation 0.000ns
Hold : 0 Failing Endpoints, Worst Slack 0.347ns, Total Violation 0.000ns
----------------------------------------------------------------------------



Max Delay Paths
----------------------------------------------------------------------------

Slack (MET) : 18.034ns (required time - arrival time)
  Source: design_1_i/proc_sys_reset_0/U0/ACTIVE_LOW_PR_OUT_DFF[0].
      FDRE_PER_N/C
                          (rising edge-triggered cell FDRE clocked by
                              clk_fpga_0 {rise@0.000ns fall@10.000ns period
                              =20.000ns})
  Destination: design_1_i/TOP_encryption_0/U0/ks_reg_1/OUTPUT_reg[0]/CLR
                          (recovery check against rising-edge clock
                              clk_fpga_0 {rise@0.000ns fall@10.000ns period
                              =20.000ns})
  Path Group: **async_default**
  Path Type: Recovery (Max at Slow Process Corner)
  Requirement: 20.000ns (clk_fpga_0 rise@20.000ns - clk_fpga_0 rise@0.000ns
      )
  Data Path Delay: 0.939ns (logic 0.456ns (48.562%) route 0.483ns (51.438%)
      )
  Logic Levels: 0
```

```
Clock Path Skew: -0.145ns (DCD - SCD + CPR)
  Destination Clock Delay (DCD): 1.505ns = ( 21.505 - 20.000 )
  Source Clock Delay (SCD): 1.700ns
  Clock Pessimism Removal (CPR): 0.050ns
Clock Uncertainty: 0.302ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
  Total System Jitter (TSJ): 0.071ns
  Total Input Jitter (TIJ): 0.600ns
  Discrete Jitter (DJ): 0.000ns
  Phase Error (PE): 0.000ns

-------------------------------------------------------------------
                     required time 20.673
                     arrival time -2.639
-------------------------------------------------------------------
                     slack 18.034
```

Příloha E

Výpis využití DES šifrování

Tato příloha obsahuje *výpis využití* programu Vivado pro dešifrování algoritmem DES. Sekce reportu, kde bylo využití nulové, byly vynechány.

```
Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
-----------------------------------------------------------------------------
| Tool Version : Vivado v.2019.2 (win64) Build 2708876 Wed Nov 6
    21:40:23 MST 2019
| Date         : Tue May  4 06:36:13 2021
| Host         : PXDMLI running 64-bit major release  (build 9200)
| Command      : report_utilization -file C:/Users/Marek/Desktop/
    utilization_report_DES_enc.txt -name utilization_1
| Design       : design_1_wrapper
| Device       : 7z010clg225-1
| Design State : Synthesized
-----------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. IO and GT Specific
4. Clocking
5. Primitives
```

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	2983	0	17600	16.95
LUT as Logic	2887	0	17600	16.40
LUT as Memory	96	0	6000	1.60
LUT as Distributed RAM	0	0		
LUT as Shift Register	96	0		
Slice Registers	3939	0	35200	11.19
Register as Flip Flop	3939	0	35200	11.19
Register as Latch	0	0	35200	0.00
F7 Muxes	34	0	8800	0.39
F8 Muxes	0	0	4400	0.00

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
1856	Yes	-	Reset
13	Yes	Set	-
2070	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	8	0	60	13.33
RAMB36/FIFO*	8	0	60	13.33
RAMB36E1 only	8			
RAMB18	0	0	120	0.00

* Note: Each Block RAM Tile only has one FIFO logic available
and therefore can accommodate only one FIFO36E1 or one FIFO18E1.
However, if a FIFO18E1 occupies a Block RAM Tile, that tile
can still accommodate a RAMB18E1

3. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	0	0	54	0.00
Bonded IPADs	0	0	2	0.00
Bonded IOPADs	86	86	130	66.15
PHY_CONTROL	0	0	2	0.00
PHASER_REF	0	0	2	0.00
OUT_FIFO	0	0	8	0.00
IN_FIFO	0	0	8	0.00
IDELAYCTRL	0	0	2	0.00
IBUFDS	0	0	54	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	8	0.00
PHASER_IN/PHASER_IN_PHY	0	0	8	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	100	0.00
ILOGIC	0	0	54	0.00
OLOGIC	0	0	54	0.00

4. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	1	0	32	3.13
BUFIO	0	0	8	0.00
MMCME2_ADV	0	0	2	0.00
PLLE2_ADV	0	0	2	0.00
BUFMRCE	0	0	4	0.00
BUFHCE	0	0	48	0.00
BUFR	0	0	8	0.00

5. Primitives

Ref Name	Used	Functional Category
FDRE	2070	Flop & Latch
FDCE	1856	Flop & Latch
LUT2	1507	LUT
LUT6	920	LUT
LUT3	505	LUT
LUT5	314	LUT
LUT4	180	LUT
LUT1	139	LUT
SRLC32E	92	Distributed Memory
BIBUF	86	IO
MUXF7	34	MuxFx
FDSE	13	Flop & Latch
RAMB36E1	8	Block Memory
SRL16E	4	Distributed Memory
CARRY4	3	CarryLogic
PS7	1	Specialized Resource
BUFG	1	Clock

Příloha F

Výpis časování DES dešifrování

Příloha obsahuje *výpis využití* programu Vivado pro dešifrování algoritmem DES, bez zřetězeného zpracování. Kvůli skutečnosti, že návrh je kombinatorickým obvodem, se hlavní cesta obvodu opožďuje o 16,272 ns oproti požadovaným 20 ns. Podrobnosti této cesty, včetně cest jí velmi podobných či nedůležitých, nejsou v reportu zaznamenány.

```
Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
----------------------------------------------------------------------------

| Tool Version : Vivado v.2019.2 (win64) Build 2708876 Wed Nov 6 21:40:23
   MST 2019
| Date : Tue May 4 07:09:10 2021
| Host : PXDMLI running 64-bit major release (build 9200)
| Command : report_timing_summary -delay_type min_max -report_unconstrained
    -check_timing_verbose -max_paths 10 -input_pins -routable_nets -name
   timing_2 -file C:/Users/Marek/Desktop/timing_report._DES_dec.txt
| Design : design_1_wrapper
| Device : 7z010-clg225
| Speed File : -1 PRODUCTION 1.11 2014-09-11
----------------------------------------------------------------------------


Timing Summary Report

----------------------------------------------------------------------------

| Timer Settings
| --------------
----------------------------------------------------------------------------


  Enable Multi Corner Analysis : Yes
  Enable Pessimism Removal : Yes
  Pessimism Removal Resolution : Nearest Common Node
  Enable Input Delay Default Clock : No
  Enable Preset / Clear Arcs : No
```

```
  Disable Flight Delays : No
  Ignore I/O Paths : No
  Timing Early Launch at Borrowing Latches : No
  Borrow Time for Max Delay Exceptions : Yes
  Merge Timing Exceptions : Yes

  Corner Analyze Analyze
  Name Max Paths Min Paths
  ------ --------- ---------
  Slow Yes Yes
  Fast Yes Yes


------------------------------------------------------------------------------------

| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------


    WNS(ns) : -16.272
    TNS(ns) : -982.401
    TNS Failing Endpoints : 64
    TNS Total Endpoints : 5618
    WHS(ns) : 0.007
    THS(ns) : 0.000
    THS Failing Endpoints : 0
    THS Total Endpoints : 5618
    WPWS(ns) : 9.020
    TPWS(ns) : 0.000
    TPWS Failing Endpoints : 0
    TPWS Total Endpoints : 2196


Timing constraints are not met.


------------------------------------------------------------------------------------

| Clock Summary
| -------------
------------------------------------------------------------------------------------


Clock : clk_fpga_0
Waveform(ns) : {0.000 10.000}
Period(ns) : 20.000
Frequency(MHz) : 50.000
```

```
------------------------------------------------------------------------------

| Intra Clock Table
| -----------------
------------------------------------------------------------------------------


Clock : clk_fpga_0
WNS(ns) : -16.272
TNS(ns) : -982.401
TNS Failing Endpoints : 64
TNS Total Endpoints : 5618
WHS(ns) : 0.007
THS(ns) : 0.000
THS Failing Endpoints : 0
THS Total Endpoints : 5618
WPWS(ns) : 9.020
TPWS(ns) : 0.000
TPWS Failing Endpoints : 0
TPWS Total Endpoints : 2196

------------------------------------------------------------------------------

| Timing Details
| --------------
------------------------------------------------------------------------------


From Clock : clk_fpga_0
To Clock : clk_fpga_0

Setup :
    64 Failing Endpoints
    Worst Slack -16.272ns
    Total Violation -982.401ns
Hold :
    0 Failing Endpoints
    Worst Slack 0.007ns
    Total Violation 0.000ns
PW :
    0 Failing Endpoints
    Worst Slack 9.020ns
    Total Violation 0.000ns
------------------------------------------------------------------------------
```

```
Max Delay Path
----------------------------------------------------------------------------

Slack (VIOLATED) : -16.272ns (required time - arrival time)
  Source: design_1_i/blk_mem_gen_read/U0/inst_blk_mem_gen/gnbram.
      gnative_mem_map_bmg.native_mem_map_blk_mem_gen/valid.cstr/ramloop[0].
      ram.r/prim_noinit.ram/DEVICE_7SERIES.WITH_BMM_INFO.TRUE_DP.
      SIMPLE_PRIM36.TDP_SP36_NO_ECC_ATTR.ram/CLKBWRCLK (rising edge-
      triggered cell RAMB36E1 clocked by clk_fpga_0 {rise@0.000ns fall@10
      .000ns period=20.000ns})
  Destination: design_1_i/blk_mem_gen_write/U0/inst_blk_mem_gen/gnbram.
      gnative_mem_map_bmg.native_mem_map_blk_mem_gen/valid.cstr/ramloop[0].
      ram.r/prim_noinit.ram/DEVICE_7SERIES.WITH_BMM_INFO.TRUE_DP.
      SIMPLE_PRIM36.TDP_SP36_NO_ECC_ATTR.ram/DIBDI[0] (rising edge-
      triggered cell RAMB36E1 clocked by clk_fpga_0 {rise@0.000ns fall@10
      .000ns period=20.000ns})
  Path Group: clk_fpga_0
  Path Type: Setup (Max at Slow Process Corner)
  Requirement: 20.000ns (clk_fpga_0 rise@20.000ns - clk_fpga_0 rise@0.000ns
      )
  Data Path Delay: 35.087ns (logic 6.794ns (19.363%) route 28.293ns
      (80.637%))
  Logic Levels: 35 (LUT2=3 LUT3=4 LUT4=4 LUT5=7 LUT6=17)
  Clock Path Skew: -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD): 1.505ns = ( 21.505 - 20.000 )
    Source Clock Delay (SCD): 1.700ns
    Clock Pessimism Removal (CPR): 0.050ns
  Clock Uncertainty: 0.302ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter (TSJ): 0.071ns
    Total Input Jitter (TIJ): 0.600ns
    Discrete Jitter (DJ): 0.000ns
    Phase Error (PE): 0.000ns

  -------------------------------------------------------------------
                      required time 20.516
                      arrival time -36.788
  -------------------------------------------------------------------
                      slack -16.272
```

Příloha G

Výpis využití DES dešifrování

V příloze je *výpis využití* programu Vivado pro dešifrovací obvod algoritmu DES. Sekce reportu, kde využití bylo nulové, jsou vynechány.

```
Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
--------------------------------------------------------------------------------
| Tool Version : Vivado v.2019.2 (win64) Build 2708876 Wed Nov 6
    21:40:23 MST 2019
| Date         : Tue May  4 10:02:19 2021
| Host         : PXDMLI running 64-bit major release  (build 9200)
| Command      : report_utilization -file C:/Users/Marek/Desktop/
    utilization_report_DES_dec.txt -name utilization_1
| Design       : design_1_wrapper
| Device       : 7z010clg225-1
| Design State : Synthesized
--------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. IO and GT Specific
4. Clocking
5. Primitives
```

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	2597	0	17600	14.76
LUT as Logic	2501	0	17600	14.21
LUT as Memory	96	0	6000	1.60
LUT as Distributed RAM	0	0		
LUT as Shift Register	96	0		
Slice Registers	2083	0	35200	5.92
Register as Flip Flop	2083	0	35200	5.92
Register as Latch	0	0	35200	0.00
F7 Muxes	34	0	8800	0.39
F8 Muxes	0	0	4400	0.00

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
0	Yes	-	Reset
13	Yes	Set	-
2070	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	8	0	60	13.33
RAMB36/FIFO*	8	0	60	13.33
RAMB36E1 only	8			
RAMB18	0	0	120	0.00

* Note: Each Block RAM Tile only has one FIFO logic available
and therefore can accommodate only one FIFO36E1 or one FIFO18E1.
However, if a FIFO18E1 occupies a Block RAM Tile, that tile
can still accommodate a RAMB18E1

3. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	0	0	54	0.00
Bonded IPADs	0	0	2	0.00
Bonded IOPADs	86	86	130	66.15
PHY_CONTROL	0	0	2	0.00
PHASER_REF	0	0	2	0.00
OUT_FIFO	0	0	8	0.00
IN_FIFO	0	0	8	0.00
IDELAYCTRL	0	0	2	0.00
IBUFDS	0	0	54	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	8	0.00
PHASER_IN/PHASER_IN_PHY	0	0	8	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	100	0.00
ILOGIC	0	0	54	0.00
OLOGIC	0	0	54	0.00

4. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	1	0	32	3.13
BUFIO	0	0	8	0.00
MMCME2_ADV	0	0	2	0.00
PLLE2_ADV	0	0	2	0.00
BUFMRCE	0	0	4	0.00
BUFHCE	0	0	48	0.00
BUFR	0	0	8	0.00

5. Primitives

Ref Name	Used	Functional Category
FDRE	2070	Flop & Latch
LUT6	968	LUT
LUT3	697	LUT
LUT5	634	LUT
LUT4	372	LUT
LUT2	371	LUT
LUT1	139	LUT
SRLC32E	92	Distributed Memory
BIBUF	86	IO
MUXF7	34	MuxFx
FDSE	13	Flop & Latch
RAMB36E1	8	Block Memory
SRL16E	4	Distributed Memory
CARRY4	3	CarryLogic
PS7	1	Specialized Resource
BUFG	1	Clock

Příloha H

Výpis časování AES šifrování

V příloze se nachází *Výpis časování* z programu Vivado pro komponentu zajišťující šifrování pomocí algoritmu AES. Obvod využívá techniky zřetězeného zpracování a díky tomu je schopen pracovat na požadované frekvenci 20 ns.

```
Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
----------------------------------------------------------------------------

| Tool Version : Vivado v.2019.2 (win64) Build 2708876 Wed Nov 6 21:40:23
   MST 2019
| Date : Tue May 4 05:21:08 2021
| Host : PXDMLI running 64-bit major release (build 9200)
| Command : report_timing_summary -delay_type min_max -report_unconstrained
    -check_timing_verbose -max_paths 10 -input_pins -routable_nets -name
   timing_3 -file C:/Users/Marek/Desktop/timing_report._AES_enc.txt
| Design : design_1_wrapper
| Device : 7z010-clg225
| Speed File : -1 PRODUCTION 1.11 2014-09-11
----------------------------------------------------------------------------


Timing Summary Report

----------------------------------------------------------------------------

| Timer Settings
| --------------
----------------------------------------------------------------------------


  Enable Multi Corner Analysis : Yes
  Enable Pessimism Removal : Yes
  Pessimism Removal Resolution : Nearest Common Node
  Enable Input Delay Default Clock : No
  Enable Preset / Clear Arcs : No
  Disable Flight Delays : No
```

```
  Ignore I/O Paths : No
  Timing Early Launch at Borrowing Latches : No
  Borrow Time for Max Delay Exceptions : Yes
  Merge Timing Exceptions : Yes

  Corner Analyze Analyze
  Name Max Paths Min Paths
  ------ --------- ---------
  Slow Yes Yes
  Fast Yes Yes

----------------------------------------------------------------------------

| Design Timing Summary
| ---------------------
----------------------------------------------------------------------------


    WNS(ns) : 13.224
    TNS(ns) : 0.000
    TNS Failing Endpoints : 0
    TNS Total Endpoints : 8882
    WHS(ns) : 0.007
    THS(ns) : 0.000
    THS Failing Endpoints : 0
    THS Total Endpoints : 8882
    WPWS(ns) : 9.020
    TPWS(ns) : 0.000
    TPWS Failing Endpoints : 0
    TPWS Total Endpoints : 4500

Timing constraints are met.

----------------------------------------------------------------------------

| Clock Summary
| -------------
----------------------------------------------------------------------------


Clock : clk_fpga_0
Waveform(ns) : {0.000 10.000}
Period(ns) : 20.000
Frequency(MHz) : 50.000

----------------------------------------------------------------------------

| Intra Clock Table
```

```
| -----------------
---------------------------------------------------------------------------


Clock : clk_fpga_0
WNS(ns) : 13.224
TNS(ns) : 0.000
TNS Failing Endpoints : 0
TNS Total Endpoints : 6578
WHS(ns) : 0.007
THS(ns) : 0.000
THS Failing Endpoints : 0
THS Total Endpoints : 6578
WPWS(ns) : 9.020
TPWS(ns) : 0.000
TPWS Failing Endpoints : 0
TPWS Total Endpoints : 4500


---------------------------------------------------------------------------

| Timing Details
| --------------
---------------------------------------------------------------------------



---------------------------------------------------------------------------

From Clock: clk_fpga_0
  To Clock: clk_fpga_0

Setup :
    0 Failing Endpoints
    Worst Slack 13.224ns
    Total Violation 0ns
Hold :
    0 Failing Endpoints
    Worst Slack 0.007ns
    Total Violation 0.000ns
PW :
    0 Failing Endpoints
    Worst Slack 9.020ns
    Total Violation 0.000ns
---------------------------------------------------------------------------
```

```
Max Delay Paths
--------------------------------------------------------------------------------

Slack (MET) : 13.224ns (required time - arrival time)
  Source: design_1_i/axi_interconnect_0/s00_couplers/auto_us/inst/
      gen_upsizer.gen_full_upsizer.axi_upsizer_inst/si_register_slice_inst/
      aw.aw_pipe/m_payload_i_reg[45]/C
                        (rising edge-triggered cell FDRE clocked by
                           clk_fpga_0 {rise@0.000ns fall@10.000ns period
                           =20.000ns})
  Destination: design_1_i/axi_interconnect_0/s00_couplers/auto_us/inst/
      gen_upsizer.gen_full_upsizer.axi_upsizer_inst/USE_WRITE.
      write_addr_inst/GEN_CMD_QUEUE.cmd_queue/USE_RTL_FIFO.data_srl_reg
      [31][6]_srl32/D
                        (rising edge-triggered cell SRLC32E clocked by
                           clk_fpga_0 {rise@0.000ns fall@10.000ns period
                           =20.000ns})
  Path Group: clk_fpga_0
  Path Type: Setup (Max at Slow Process Corner)
  Requirement: 20.000ns (clk_fpga_0 rise@20.000ns - clk_fpga_0 rise@0.000ns
      )
  Data Path Delay: 6.285ns (logic 1.828ns (29.085%) route 4.457ns (70.915%)
      )
  Logic Levels: 6 (CARRY4=1 LUT5=1 LUT6=4)
  Clock Path Skew: -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD): 1.505ns = ( 21.505 - 20.000 )
    Source Clock Delay (SCD): 1.700ns
    Clock Pessimism Removal (CPR): 0.050ns
  Clock Uncertainty: 0.302ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter (TSJ): 0.071ns
    Total Input Jitter (TIJ): 0.600ns
    Discrete Jitter (DJ): 0.000ns
    Phase Error (PE): 0.000ns

  -------------------------------------------------------------------
                      required time 21.209
                      arrival time -7.985
  -------------------------------------------------------------------
                      slack 13.224

--------------------------------------------------------------------------------

Path Group: **async_default**
From Clock: clk_fpga_0
  To Clock: clk_fpga_0

Setup :
    0 Failing Endpoints
```

```
    Worst Slack 18.034ns
    Total Violation 0.000ns
Hold :
    0 Failing Endpoints
    Worst Slack 0.347ns
    Total Violation 0.000ns
----------------------------------------------------------------------------



Max Delay Paths
----------------------------------------------------------------------------

Slack (MET) : 18.034ns (required time - arrival time)
  Source: design_1_i/proc_sys_reset_0/U0/ACTIVE_LOW_PR_OUT_DFF[0].
      FDRE_PER_N/C
                         (rising edge-triggered cell FDRE clocked by
                            clk_fpga_0 {rise@0.000ns fall@10.000ns period
                            =20.000ns})
  Destination: design_1_i/AES_top_0/U0/k1_reg/OUTPUT_reg[0]/CLR
                         (recovery check against rising-edge clock
                            clk_fpga_0 {rise@0.000ns fall@10.000ns period
                            =20.000ns})
  Path Group: **async_default**
  Path Type: Recovery (Max at Slow Process Corner)
  Requirement: 20.000ns (clk_fpga_0 rise@20.000ns - clk_fpga_0 rise@0.000ns
      )
  Data Path Delay: 0.939ns (logic 0.456ns (48.562%) route 0.483ns (51.438%)
      )
  Logic Levels: 0
  Clock Path Skew: -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD): 1.505ns = ( 21.505 - 20.000 )
    Source Clock Delay (SCD): 1.700ns
    Clock Pessimism Removal (CPR): 0.050ns
  Clock Uncertainty: 0.302ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter (TSJ): 0.071ns
    Total Input Jitter (TIJ): 0.600ns
    Discrete Jitter (DJ): 0.000ns
    Phase Error (PE): 0.000ns


  -------------------------------------------------------------------
                      required time 20.673
                      arrival time -2.639
  -------------------------------------------------------------------
                      slack 18.034
```

Příloha I

Výpis využití AES šifrování

Příloha obsahuje *výpis využití* programu Vivado pro obvod, který je schopen šifrovat bloky zpráv pomocí algoritmu AES. Sekce reportu, kde využití daných zdrojů bylo nulové, jsou z reportu vynechány.

```
Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
-----------------------------------------------------------------------------
| Tool Version : Vivado v.2019.2 (win64) Build 2708876 Wed Nov 6
    21:40:23 MST 2019
| Date         : Tue May  4 05:23:17 2021
| Host         : PXDMLI running 64-bit major release  (build 9200)
| Command      : report_utilization -file C:/Users/Marek/Desktop/
    utilization_report_AES_enc.txt -name utilization_2
| Design       : design_1_wrapper
| Device       : 7z010clg225-1
| Design State : Synthesized
-----------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. IO and GT Specific
4. Clocking
5. Primitives
```

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	12715	0	17600	72.24
LUT as Logic	12619	0	17600	71.70
LUT as Memory	96	0	6000	1.60
LUT as Distributed RAM	0	0		
LUT as Shift Register	96	0		
Slice Registers	2083	0	35200	5.92
Register as Flip Flop	2083	0	35200	5.92
Register as Latch	0	0	35200	0.00
F7 Muxes	1878	0	8800	21.34
F8 Muxes	712	0	4400	16.18

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
0	Yes	-	Reset
13	Yes	Set	-
2070	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	8	0	60	13.33
RAMB36/FIFO*	8	0	60	13.33
RAMB36E1 only	8			
RAMB18	0	0	120	0.00

* Note: Each Block RAM Tile only has one FIFO logic available
and therefore can accommodate only one FIFO36E1 or one FIFO18E1.
However, if a FIFO18E1 occupies a Block RAM Tile, that tile
can still accommodate a RAMB18E1

3. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	0	0	54	0.00
Bonded IPADs	0	0	2	0.00
Bonded IOPADs	86	86	130	66.15
PHY_CONTROL	0	0	2	0.00
PHASER_REF	0	0	2	0.00
OUT_FIFO	0	0	8	0.00
IN_FIFO	0	0	8	0.00
IDELAYCTRL	0	0	2	0.00
IBUFDS	0	0	54	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	8	0.00
PHASER_IN/PHASER_IN_PHY	0	0	8	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	100	0.00
ILOGIC	0	0	54	0.00
OLOGIC	0	0	54	0.00

4. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	1	0	32	3.13
BUFIO	0	0	8	0.00
MMCME2_ADV	0	0	2	0.00
PLLE2_ADV	0	0	2	0.00
BUFMRCE	0	0	4	0.00
BUFHCE	0	0	48	0.00
BUFR	0	0	8	0.00

5. Primitives

Ref Name	Used	Functional Category
LUT6	9693	LUT
FDRE	2070	Flop & Latch
MUXF7	1878	MuxFx
LUT5	1095	LUT
LUT4	864	LUT
LUT2	750	LUT
MUXF8	712	MuxFx
LUT3	671	LUT
LUT1	139	LUT
SRLC32E	92	Distributed Memory
BIBUF	86	IO
FDSE	13	Flop & Latch
RAMB36E1	8	Block Memory
SRL16E	4	Distributed Memory
CARRY4	3	CarryLogic
PS7	1	Specialized Resource
BUFG	1	Clock

Příloha J

Výpis časování AES dešifrování

V příloze se nachází *Výpis časování* programu Vivado pro komponentu zajišťující proces dešifrování algoritmem AES. Obvod je kombinatorický a nevyužívá zřetězeného zpracování, proto se celá jeho cesta zpožďuje o 33,925 ns. Podrobnosti této cesty, včetně cest jí velmi podobných či nedůležitých, nejsou v reportu zaznamenány.

```
Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
-----------------------------------------------------------------------------

| Tool Version : Vivado v.2019.2 (win64) Build 2708876 Wed Nov 6 21:40:23
   MST 2019
| Date : Tue May 4 06:04:15 2021
| Host : PXDMLI running 64-bit major release (build 9200)
| Command : report_timing_summary -delay_type min_max -report_unconstrained
    -check_timing_verbose -max_paths 10 -input_pins -routable_nets -name
   timing_1 -file C:/Users/Marek/Desktop/timing_report._AES_dec.txt
| Design : design_1_wrapper
| Device : 7z010-clg225
| Speed File : -1 PRODUCTION 1.11 2014-09-11
-----------------------------------------------------------------------------


Timing Summary Report

-----------------------------------------------------------------------------

| Timer Settings
| --------------
-----------------------------------------------------------------------------


  Enable Multi Corner Analysis : Yes
  Enable Pessimism Removal : Yes
  Pessimism Removal Resolution : Nearest Common Node
  Enable Input Delay Default Clock : No
  Enable Preset / Clear Arcs : No
```

```
  Disable Flight Delays : No
  Ignore I/O Paths : No
  Timing Early Launch at Borrowing Latches : No
  Borrow Time for Max Delay Exceptions : Yes
  Merge Timing Exceptions : Yes

  Corner Analyze Analyze
  Name Max Paths Min Paths
  ------ --------- ---------
  Slow Yes Yes
  Fast Yes Yes

------------------------------------------------------------------------------

| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------


    WNS(ns) : -33.925
    TNS(ns) : -4309.732
    TNS Failing Endpoints : 128
    TNS Total Endpoints : 5682
    WHS(ns) : 0.007
    THS(ns) : 0.000
    THS Failing Endpoints : 0
    THS Total Endpoints : 5682
    WPWS(ns) : 9.020
    TPWS(ns) : 0.000
    TPWS Failing Endpoints : 0
    TPWS Total Endpoints : 2196

Timing constraints are not met.

------------------------------------------------------------------------------

| Clock Summary
| -------------
------------------------------------------------------------------------------


Clock : clk_fpga_0
Waveform(ns) : {0.000 10.000}
Period(ns) : 20.000
Frequency(MHz) : 50.000

------------------------------------------------------------------------------
```

```
| Intra Clock Table
| -----------------
------------------------------------------------------------------------------


Clock : clk_fpga_0
WNS(ns) : -33.925
TNS(ns) : -4309.732
TNS Failing Endpoints : 128
TNS Total Endpoints : 5682
WHS(ns) : 0.007
THS(ns) : 0.000
THS Failing Endpoints : 0
THS Total Endpoints : 5682
WPWS(ns) : 9.020
TPWS(ns) : 0.000
TPWS Failing Endpoints : 0
TPWS Total Endpoints : 2196

------------------------------------------------------------------------------

| Inter Clock Table
| -----------------
------------------------------------------------------------------------------


------------------------------------------------------------------------------

| Timing Details
| --------------
------------------------------------------------------------------------------



------------------------------------------------------------------------------

From Clock: clk_fpga_0
  To Clock: clk_fpga_0

Setup :
    128 Failing Endpoints
    Worst Slack -33.925ns
    Total Violation -4309.732ns
Hold :
    0 Failing Endpoints
    Worst Slack 0.007ns
    Total Violation 0.000ns
PW :
```

```
    0 Failing Endpoints
    Worst Slack 9.020ns
    Total Violation 0.000ns
----------------------------------------------------------------------------


Max Delay Paths
----------------------------------------------------------------------------

Slack (VIOLATED) : -33.925ns (required time - arrival time)
  Source: design_1_i/blk_mem_gen_read/U0/inst_blk_mem_gen/gnbram.
      gnative_mem_map_bmg.native_mem_map_blk_mem_gen/valid.cstr/ramloop[2].
      ram.r/prim_noinit.ram/DEVICE_7SERIES.WITH_BMM_INFO.TRUE_DP.
      SIMPLE_PRIM36.TDP_SP36_NO_ECC_ATTR.ram/CLKBWRCLK
                          (rising edge-triggered cell RAMB36E1 clocked by
                              clk_fpga_0 {rise@0.000ns fall@10.000ns period
                              =20.000ns})
  Destination: design_1_i/blk_mem_gen_write/U0/inst_blk_mem_gen/gnbram.
      gnative_mem_map_bmg.native_mem_map_blk_mem_gen/valid.cstr/ramloop[2].
      ram.r/prim_noinit.ram/DEVICE_7SERIES.WITH_BMM_INFO.TRUE_DP.
      SIMPLE_PRIM36.TDP_SP36_NO_ECC_ATTR.ram/DIBDI[24]
                          (rising edge-triggered cell RAMB36E1 clocked by
                              clk_fpga_0 {rise@0.000ns fall@10.000ns period
                              =20.000ns})
  Path Group: clk_fpga_0
  Path Type: Setup (Max at Slow Process Corner)
  Requirement: 20.000ns (clk_fpga_0 rise@20.000ns - clk_fpga_0 rise@0.000ns
      )
  Data Path Delay: 52.740ns (logic 10.404ns (19.727%) route 42.336ns
      (80.273%))
  Logic Levels: 64 (LUT2=17 LUT3=9 LUT4=4 LUT5=5 LUT6=29)
  Clock Path Skew: -0.145ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD): 1.505ns = ( 21.505 - 20.000 )
    Source Clock Delay (SCD): 1.700ns
    Clock Pessimism Removal (CPR): 0.050ns
  Clock Uncertainty: 0.302ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter (TSJ): 0.071ns
    Total Input Jitter (TIJ): 0.600ns
    Discrete Jitter (DJ): 0.000ns
    Phase Error (PE): 0.000ns

  -------------------------------------------------------------------
                       required time 20.516
                       arrival time -54.441
  -------------------------------------------------------------------
                       slack -33.925
```

Příloha K

Výpis využití AES dešifrování

Příloha obsahuje *výpis využití* programu Vivado obvodu pro dešifrování algoritmem DES. Sekce reportu, kde využití daných zdrojů bylo nulové, jsou z reportu vynechány.

```
Copyright 1986-2019 Xilinx, Inc. All Rights Reserved.
-----------------------------------------------------------------------------
| Tool Version : Vivado v.2019.2 (win64) Build 2708876 Wed Nov 6
    21:40:23 MST 2019
| Date         : Tue May  4 06:05:15 2021
| Host         : PXDMLI running 64-bit major release  (build 9200)
| Command      : report_utilization -file C:/Users/Marek/Desktop/
    utilization_report_AES_dec.txt -name utilization_1
| Design       : design_1_wrapper
| Device       : 7z010clg225-1
| Design State : Synthesized
-----------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. IO and GT Specific
4. Clocking
5. Primitives
```

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	14162	0	17600	80.47
LUT as Logic	14066	0	17600	79.92
LUT as Memory	96	0	6000	1.60
LUT as Distributed RAM	0	0		
LUT as Shift Register	96	0		
Slice Registers	2083	0	35200	5.92
Register as Flip Flop	2083	0	35200	5.92
Register as Latch	0	0	35200	0.00
F7 Muxes	884	0	8800	10.05
F8 Muxes	270	0	4400	6.14

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
0	Yes	-	Set
0	Yes	-	Reset
13	Yes	Set	-
2070	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	8	0	60	13.33
RAMB36/FIFO*	8	0	60	13.33
RAMB36E1 only	8			
RAMB18	0	0	120	0.00

* Note: Each Block RAM Tile only has one FIFO logic available
and therefore can accommodate only one FIFO36E1 or one FIFO18E1.
However, if a FIFO18E1 occupies a Block RAM Tile, that tile
can still accommodate a RAMB18E1

3. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	0	0	54	0.00
Bonded IPADs	0	0	2	0.00
Bonded IOPADs	86	86	130	66.15
PHY_CONTROL	0	0	2	0.00
PHASER_REF	0	0	2	0.00
OUT_FIFO	0	0	8	0.00
IN_FIFO	0	0	8	0.00
IDELAYCTRL	0	0	2	0.00
IBUFDS	0	0	54	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	8	0.00
PHASER_IN/PHASER_IN_PHY	0	0	8	0.00
IDELAYE2/IDELAYE2_FINEDELAY	0	0	100	0.00
ILOGIC	0	0	54	0.00
OLOGIC	0	0	54	0.00

4. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	1	0	32	3.13
BUFIO	0	0	8	0.00
MMCME2_ADV	0	0	2	0.00
PLLE2_ADV	0	0	2	0.00
BUFMRCE	0	0	4	0.00
BUFHCE	0	0	48	0.00
BUFR	0	0	8	0.00

5. Primitives

Ref Name	Used	Functional Category
LUT6	10670	LUT
FDRE	2070	Flop & Latch
LUT2	1271	LUT
LUT3	1008	LUT
LUT5	999	LUT
LUT4	907	LUT
MUXF7	884	MuxFx
MUXF8	270	MuxFx
LUT1	139	LUT
SRLC32E	92	Distributed Memory
BIBUF	86	IO
FDSE	13	Flop & Latch
RAMB36E1	8	Block Memory
SRL16E	4	Distributed Memory
CARRY4	3	CarryLogic
PS7	1	Specialized Resource
BUFG	1	Clock