VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

NÁVRH SPECIALIZOVANÝCH INSTRUKCÍ

SPECIALIZED INSTRUCTION DESIGN

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAN KOSCIELNIAK**
AUTHOR

VEDOUCÍ PRÁCE **Prof. Ing. TOMÁŠ HRUŠKA, CSc.**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/20405/2017/xkosci00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Koscielniak Jan**
Obor: Informační technologie
Téma: **Návrh specializovaných instrukcí**
Specialized Instruction Design
Kategorie: Návrh číslicových systémů

Pokyny:

1. Seznamte se s nástrojem Codasip Studio, jazykem CodAL a architekturou RISC-V.
2. Seznamte se s principy kryptografie v informatice a s možnostmi její akcelerace na úrovni hardware.
3. Vyberte vhodné kryptografické algoritmy a analyzujte možnosti jejich urychlení pomocí specializovaných instrukcí.
4. Navrhněte sadu instrukčních rozšíření pro architekturu RISC-V pro zvolené algoritmy.
5. Implementujte navržená instrukční rozšíření upravením instrukční sady RISC-V s využitím prostředí Codasip Studio.
6. Otestujte a zhodnoťte zvýšení efektivity pro nový instrukční repertoár, diskutujte případné další možné úpravy instrukční sady a architektury.

Literatura:

- Introduction to modern cryptography: KATZ, Jonathan a Yehuda LINDELL ; Second edition. Boca Raton, FL: CRC Press, 2015
- Practical Cryptography: Algorithms and Implementations Using C++. Azad, Saiful ; Pathan, Al - Sakib Khan ; Philadelphia, PA : CRC Press ; 2014
- Hardware Acceleration for Cryptography Algorithms by Hotspot Detection: Chang, Jed Kao-Tung and Liu, Chen and Gaudiot, Jean-Luc ;
- Grid and Pervasive Computing: 8th International Conference, GPC 2013 and Colocated Workshops, Seoul, Korea, May 9-11, 2013

Pro udělení zápočtu za první semestr je požadováno:

- Body 1 až 3 zadání

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hruška Tomáš, prof. Ing., CSc.**, UIFS FIT VUT
Konzultant: Šnobl Pavel, Ing., CODASIP
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

Tato práce se věnuje návrhu a implementaci specializovaných instrukcí pro architekturu instrukční sady RISC-V. Tato instrukční rozšíření slouží k akceleraci sady vybraných kryptografických algoritmů. Nové instrukce jsou implementovány v prostředí Codasip Studia na modelu 32bitového procesoru s instrukční sadou RV32IM. Byly zvoleny implementace kryptografických algoritmů s otevřeným zdrojovým kódem, který byl upraven, aby používal nové instrukce. Jednotlivé instrukce byly aplikovány na příslušné algoritmy, otestovány a profilovány. Výsledkem práce je rozšíření instrukční sady, které umožňuje až sedminásobné zrychlení v závislosti na vybraném algoritmu.

Abstract

The purpose of this thesis is to design and implement specialized instructions for RISC-V instruction set architecture. These instruction are used to accelerate a set of selected cryptographic algorithms. New instructions are implemented in Codasip Studio for 32bit processor model with RV32IM instruction set. Open source implementations were selected and edited to use new instructions. Instructions were used on respective algorithms, tested and profiled. The outcome of this thesis is instruction set extension, that enables up to seven times speed up, depending on used algorithm.

Klíčová slova

Codasip, RISC-V, RSA, AES, Blowfish, Twofish, 3DES, CodAL, Rozšíření instrukční sady

Keywords

Codasip, RISC-V, RSA, AES, Blowfish, Twofish, 3DES, CodAL, Instruction set extension

Citace

KOSCIELNIAK, Jan. *Návrh specializovaných instrukcí.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Prof. Ing. Tomáš Hruška, CSc.

Návrh specializovaných instrukcí

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc. Další informace mi poskytl Ing. Pavel Šnobl. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jan Koscielniak
12. května 2018

Poděkování

Rád bych poděkoval svému vedoucímu prof. Ing. Tomáši Hruškovi, CSc. a svému konzultantovi Ing. Pavlu Šnoblovi za odborné vedení, cenné rady a ochotu.

Obsah

Kapitola 1

Úvod

Cílem této práce je vytvoření instrukčních rozšíření pro sadu vybraných kryptografických algoritmů na architektuře RISC-V. Práce probíhá na modelu této architektury od společnosti Codasip a dalších nástrojích, které tato společnost poskytuje. Výsledkem práce by měla být rozšířená instrukční sada použitelná na IoT (Internet of Things) zařízeních. U těchto vestavěných zařízení jsou typicky omezené výpočetní zdroje a důraz je kladen i na nízkou spotřebu. IoT zařízení se mohou řadit mezi senzory, jako jsou kamery nebo teploměry, ale mohou také nabývat podobu například chytrých zámků nebo termostatů, a právě kvůli tomu, že mohou mít přímý vliv na život svých uživatelů, je nutné zamezit jejich zneužití použitím kryptografie. Tato práce si klade za cíl minimalizovat dopad využití kryptografie na rychlost aplikace optimalizováním těchto algoritmů pro co nejnižší počet cyklů procesoru.

Kryptografie je nezbytnou součástí aplikací, které nakládají s daty, která mohou být nějakým způsobem zneužita, a tím více aplikací, která tato data používají v komunikaci přes síť internet. Součástí této práce je i výběr vhodných algoritmů, které musí splňovat kritéria využitelnosti v IoT zařízeních, dobré kryptografické vlastnosti a možnost optimalizace pomocí instrukčních rozšíření.

Práce zahrnuje popis principu fungování jednotlivých algoritmů, informace o architektuře instrukční sady RISC-V, dále popis prostředí Codasip Studia a základy jazyka CodAL a popis jednotlivých přidaných instrukcí. Nakonec jsou uvedeny dosažené výsledky a zamyšlení nad možným pokračováním v podobě dalších instrukcí nebo jiných architektur.

Kapitola 2

Kryptografické algoritmy

Kryptografie je vědní obor zabývající se zabezpečenou komunikací a je nedílnou součástí moderních aplikací. Díky kryptografii je možná zabezpečená komunikace po internetu či zabezpečené ukládání dat. S požadavkem na bezpečnost také vzniká požadavek na co nejrychlejší a zároveň co nejbezpečnější šifry tak, aby zabezpečení aplikace mělo minimální vliv na její funkci a výkon. V následujících kapitolách jsou popsány některé obecné principy, na kterých jsou kryptografické algoritmy založeny, a také principy fungování některých vybraných algoritmů.

2.1 Proces výběru algoritmů pro optimalizaci

Jedním z důležitých úkolů této práce bylo vybrat vhodné algoritmy pro optimalizaci. Prvním z hlavních kritérií byla použitelnost ve vestavěných, respektive IoT (Internet of Things) zařízeních. Existuje několik situací, ve kterých by tato zařízení měla využívat kryptografii. Jedná se především o komunikaci se servery – zde nachází využití asymetrická kryptografie, pro kterou byl vybrán jeden algoritmus jako zástupce této kategorie. Dále je vhodné, aby zařízení šifrovalo data, která uchovává lokálně, jako jsou informace o použití, zálohy, apod. Zde je vhodné použití symetrické kryptografie, pro kterou bylo vybráno více algoritmů různé komplexnosti.

Druhým důležitým kritériem byla možnost optimalizace pomocí instrukčních rozšíření. Nové instrukce mohou pomoci v urychlení aplikace, ale jejich možnosti nejsou nekonečné. Velkou část kryptografických algoritmů typicky tvoří přístupy do paměti, kdy se nahrazují části vnitřního stavu z dvourozměrného pole. Přístupy do paměti lze optimalizovat na RISC architekturách jen velmi obtížně. Lze použít SIMD (Single Instruction Multiple Data) instrukce, ty však vyžadují změny v procesoru, jako je možnost souběžného přístupu do paměti. Příkladem algoritmu, který nelze optimalizovat je SHA-1, kde je většina cyklů spotřebována na přístupy do paměti nebo čtení ze souboru[1]. Naopak lze dobře optimalizovat části algoritmů, které se skládají ze sekvencí operací jako jsou rotace, posuny, případně logické operace jako XOR nebo AND.

2.2 Symetrické algoritmy

Symetrická kryptografie je založena na používání stejného klíče pro šifrování i dešifrování. Hlavními výhodami je rychlost, výpočetní nenáročnost těchto algoritmů a také fakt, že pro relativně malý klíč poskytují dobrou úroveň zabezpečení. Hlavní nevýhodou je pak nutnost

udržovat tajný klíč pro každé dva komunikující subjekty a obtížnost zabezpečeného předání tajného klíče. Tato podkapitola byla převzata z [2].

Blokové šifry

Blokové šifry se liší od proudových šifer ve způsobu zpracování vstupu. Proudové šifry zpracovávají vstupní text po jednom znaku, kdežto blokové šifry pracují po blocích předem definované délky n. Blokové šifry využívají dvou hlavních principů, které poprvé představil C. Shannon roku 1945 jako součást práce pro Bellovy laboratoře: zmatení a rozptyl.

Zmatení (Confusion)

Zmatení popisuje závislost výstupního (zašifrovaného) textu na klíči. V ideálním případě každý znak klíče ovlivňuje každý znak bloku šifrovaného textu. Zároveň je nutné zajistit, aby nebylo možné klíč zrekonstruovat na základě statistiky ze zašifrovaného textu. Dobré zmatení je tedy, pokud každý znak zašifrovaného textu závisí na několika částech klíče zdánlivě náhodně. K dosažení zmatení se využívá substituce.

Rozptyl (Diffusion)

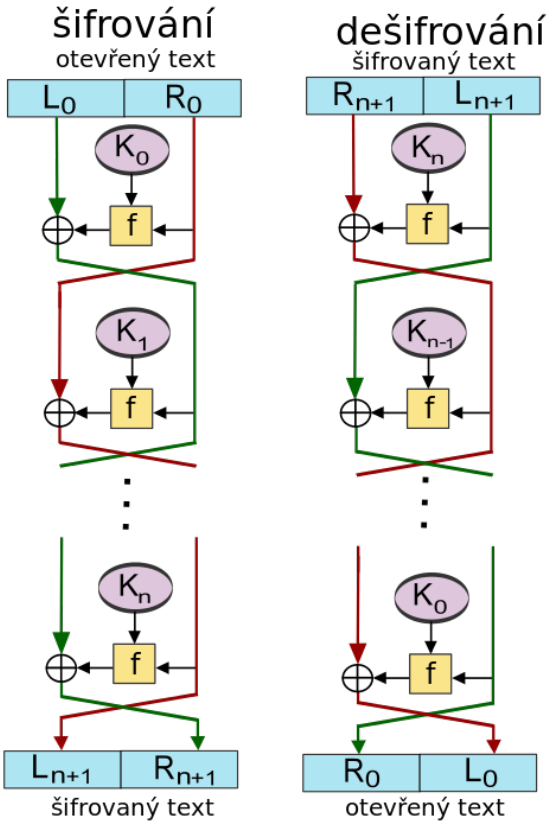
Rozptyl zavádí permutování (změnu pořadí) bitů, tedy šíření změny i v jediném bitu vstupního textu do celého bloku šifrovaného textu.

Substitučně-permutační sítě

Principy rozptylu a zmatení implementují substitučně-permutační sítě (SPN). Pracují po tzv. rounds neboli *rundách*, ve kterých se provádí zároveň permutace i substituce. Využívají se podklíče, které se odvozují z tzv. master klíče, které se často tvoří pouze spojováním různých částí tohoto klíče. V každé rundě je vstup XORován s daným podklíčem pro tuto rundu, výsledek této operace je poté vstupem pro substituci pomocí S-boxů, kdy každý S-box operuje nad částí vstupu. Výstupy z S-boxů jsou poté permutovány. Počet rund je závislý na kompromisu mezi rychlostí, potažmo výpočetní náročností a úrovní zabezpečení.

Feistelovy sítě

Další možnou implementací jsou Feistelovy sítě, implementované ve Feistelově šifře. Podstatnou výhodou oproti SPN je absence nutnosti invertovatelnosti funkcí použitých v síti. Feistelova síť totiž k inverzi rund při dešifrování používá převrácení pořadí rund. Operace v rundě samotné tak nemusí být invertovatelné, což umožňuje snadnější konstrukci těchto funkcí. Fungování šifry ilustruje obrázek 2.1. Vstupní text se rozdělí na dvě poloviny, provede se substituce na pravé části (R_0), ta se pak XORuje s levou částí (L_0) a produkuje tak R_1. R_0 se stává L_1. Feistelova šifra je základem pro mnoho dnešních blokových šifer.



Obrázek 2.1: Schéma procesu šifrování a dešifrování u Feistelovy šifry.[3]

2.2.1 DES

DES (Data Encryption Standard) je algoritmus vyvinutý firmou IBM ve spolupráci s Národní bezpečnostní agenturou (NSA), který byl ustanoven standardem v roce 1977. DES vychází z principů Feistelovy sítě. Algoritmus pracuje s bloky o velikosti 64 bitů a s klíči o efektivní velikosti 56 bitů. Zašifrování a dešifrování probíhá v 16 rundách, kdy pro každou rundu je použita jiná část klíče. Implementace použitá v této práci pochází z mbedTLS knihovny[4]. Následuje popis jednotlivých částí algoritmu.

Počáteční a závěrečná permutace

Prvním a posledním krokem algoritmu je permutace, tedy změna pořadí bitů. Ta probíhá pomocí tabulky 2.1. Forma tabulky je zvolena pro lepší přehlednost, ve skutečnosti se jedná o pole, kde index (indexováno od 1) značí původní pozici a hodnota na tomto indexu novou pozici. Pro závěrečnou permutaci je použita stejná tabulka, který je pouze invertovaná.

Rundová funkce *f*

Po počáteční permutaci se provádí 16 identických rund, které popisuje rundová funkce. Vstupem funkce je horních 32 bitů 64bitového bloku z předchozí operace a podklíč pro danou rundu. Nad výstupem této funkce a spodními 32 bity vstupního bloku je pak provedena

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

Tabulka 2.1: Tabulka pro počáteční permutaci

operace XOR. Funkce tedy mění pouze spodních 32 bitů bloku. Na závěr se spodních 32 bitů a horních 32 bitů zamění a výsledný blok je vstupem další operace.

Prvním krokem funkce rozšíření 32 bitového vstupu na 48 bitů. To probíhá pomocí duplikace některých bitů podle tabulky 2.2. Lze pozorovat, že první a poslední bit je zduplikován a umístěn na okraj a také že 4bitové skupiny jsou rozšířeny na 6bitové duplikací posledních 2 bitů předchozí skupiny.

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

Tabulka 2.2: Tabulka pro expanzi na 48 bitů

Dalším krokem je provedení operace XOR nad expandovaným vstupem a podklíčem pro danou rundu. Výsledek této operace je poté rozdělen po 6 bitech na osm bloků a ty jsou vstupy substitučních boxů (též S-boxy). S-boxy provádí mapování 6bitových bloků na 4bitové. První a poslední bit 6bitového bloku je použit k určení řádku tabulky a zbývající 4 bity k určení sloupce. Pro stručnost nejsou uvedeny tabulky pro všechny S-boxy, ale pouze pro první (tabulka 2.3).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	04	13	01	02	15	11	08	03	10	06	12	05	09	00	07
1	00	15	07	04	14	02	13	01	10	06	12	11	09	05	03	08
2	04	01	14	08	13	06	02	11	15	12	09	07	03	10	05	00
3	15	12	08	02	04	09	01	07	05	11	03	14	10	00	06	13

Tabulka 2.3: Tabulka pro S-box S_1

Na závěr se výstupy S-boxů opět permutují podle tabulky 2.4. Podobně jako u počáteční a závěrečné permutace je i tato tabulka ve skutečnosti pole.

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

Tabulka 2.4: Tabulka pro permutaci S-boxů

Odvození podklíčů

Součástí každé rundy je mimo jiné i daný podklíč specifický pro danou rundu. Těchto podklíčů je tedy stejně jako rund 16, každý o délce 48 bitů, každý odvozený z 56bitového klíče. Klíč bývá rozšířen na 64 bitů tak, že každý 8. bit je paritní. Tyto bity ale nezvyšují bezpečnost šifry, protože nejsou použity pro tvorbu podklíčů. Prvním krokem je tedy redukce na 56 bitů a zároveň je prováděna permutace PC-1 podle tabulky 2.5 (ve skutečnosti opět pole). Lze si povšimnout, že paritní bity 8, 16, 24, 32, 40, 48, 56 a 64 se zde nevyskytují.

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

Tabulka 2.5: Tabulka pro permutaci PC-1

Výsledný 56bitový blok je pak rozdělen na 28bitové poloviny, které jsou pak bitově rotovány doleva (v rundách 1, 2, 9 a 16 o jeden bit, v ostatních o dva bity). Celkový počet rotací je tedy 28, z čehož vyplývá, že bloky budou stejné před první rundou a po poslední (16.) rundě. Po rotaci je nad bloky provedena permutace PC-2 podle tabulky 2.6, jejíž výsledkem je 48bitový podklíč pro danou rundu.

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

Tabulka 2.6: Tabulka pro permutaci PC-2

Dešifrování

Dešifrování obecně v Feistelových sítích znamená pouze inverzi procesu zašifrování. Začíná se tedy poslední, šestnáctou rundou. Podklíč pro tuto rundu lze snadno odvodit z hlavního klíče, stačí ho pouze permutovat nejdřív permutací PC-1 a výsledný blok pak permutací PC-2. Další výchozí bloky pro podklíče se pak odvozují místo levé pravou rotací. Počty bitů rotace jsou určeny stejně jako při šifrování, pouze pro první rundu se rotace neprovádí.

Zhodnocení DES

Samotný algoritmus se již dnes téměř nepoužívá kvůli své nedostatečné bezpečnosti. Efektivní délka klíče 56 bitů umožňuje tedy 2^{56} možných klíčů. V 70. letech dvacátého století to ještě nebyl takový problém, protože náklady na postavení stroje, který by byl schopen úspěšně provést útok hrubou silou, tedy vyzkoušením všech možných klíčů, byly astronomické (podle optimistických propočtů přibližně 20 miliónů tehdejších amerických dolarů, v přepočtu s inflací dnešních 80 miliónů dolarů), ačkoliv už tehdy bylo jasné, že šifra s tak krátkým klíčem je prolomitelná s dostatečným výkonem. Tabulka 2.7 ilustruje realizované útoky, náklady těchto útoků a dobu nutnou na získání správného klíče. Z tabulky vyplývá, že v dnešní době už nelze považovat DES za bezpečnou šifru, sama o sobě je vhodná pouze na krátkodobé zabezpečení v řádu několika málo hodin.

Rok	Autor	Stroj	Doba prolomení		Náklady
			průměr	minimum	
1997	DESCHALL	síť počítačů	-	96 dní	-
1998	distributed.net	síť počítačů	-	39 dní	-
1998	EFF	Deep Crack	15 dní	56 hodin	-
1999	EFF + distributed.net	DC + síť PC	-	22 hodin	250 000 $
2006	Univerzita Bochum	COPACOBANA	7 dní	-	9000 €
2010	PICO computing	cluster 176 FPGA	11.5 hodiny	-	-

Tabulka 2.7: Historie brute-force útoků na DES

2.2.2 Triple DES

Triple DES (Data Encryption Standard) je vylepšením algoritmu DES, ve standard byl uveden v roce 1999. Triple DES se od původního DES liší pouze tím, že místo jednoho používá tři rozdílné klíče a provádí zašifrování třikrát, čímž velmi zvyšuje bezpečnost šifry. Problémem původního DES nebyla konkrétní vada v algoritmu, ale spíše malá délka klíče, který má pouze 56 efektivních bitů. Díky tomu se DES stal napadnutelným brute-force útokem. 3DES tento problém řeší použitím tří klíčů.

Varianty

Existuje několik variant algoritmu, všechny ale nějakým způsobem používají DES algoritmus třikrát. Již zmíněná varianta s trojitým zašifrováním se dá zapsat následovně:

$$y = DES_{k_3}(DES_{k_2}(DES_{k_1}(x)))$$

Další varianta pracuje s dešifrováním v druhém kroku:

$$y = DES_{k_3}(DES^{-1}_{k_2}(DES_{k_1}(x)))$$

Zhodnocení

Ačkoliv 3DES řeší některé nedostatky DES, jako malou délku klíče, i přesto zastarává. Kvůli malé velikosti vnitřního stavu se stává napadnutelný tzv. *birthday attacks*[5], které spočívají ve vyhledání kolizí analýzou velkého množství dat. Těmto útokům se dá předejít častými obměnami klíčů, které neumožní získání velkého množství dat k analýze.

2.2.3 AES

AES (Advanced Encryption Standard) je název kryptografického standardu, který byl zaveden z důvodu nedostatečnosti 3DES. Hlavními problémy 3DES byly především neefektivní softwarová implementace a malá velikost vnitřního stavu. Algoritmus pro tento standard byl vybrán ve veřejné soutěži, kterou pořádal americký NIST institut s těmito požadavky:

- Velikost bloku 128 bitů
- Šifra musí podporovat klíče délky 128, 192 a 256 bitů
- Efektivita v softwarové i hardwarové implementaci

V roce 2001 byl po ohodnocení přihlášených algoritmů vybrán a standardizován algoritmus Rijndael, který vytvořili Belgičané Joan Daemen a Vincent Rijmen. AES se pak začal používat v Amerických vládních institucích pro šifrování dokumentů až do stupně utajení přísně tajné. AES má 10,12 nebo 14 rund v závislosti na zvolené délce klíče (128/192/256). Níže následuje popis jednotlivých částí rundové funkce AES. Kombinování sloupců se neprovádí v poslední rundě. Implementací použitou v této práci je tiny-AES-c [6]. Tato podkapitola byla převzata z [7].

Bytová substituce

Na rozdíl od DES, AES pracuje s byty a ne s bity. Dalším rozdílem je, že používá pouze jeden substituční box pro všechny byty. Vstupní text je tedy rozdělen po jednotlivých bytech na 16 částí, a ta je každá vstupem do S-boxu. Adresování v tabulce je provedeno pomocí hexadecimální reprezentace bytu ve formátu xy, kde x je řádek tabulky a y je sloupec tabulky. Tedy pro byte $A7$ bude výsledek substituce $5C$.

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

Tabulka 2.8: Tabulka pro AES S-box

Posunutí řádků

Po substituci následuje cyklické posunutí jednotlivých řádků. Jak je možné si všimnout ve schématu 2.2, první řádek zůstává bez posunutí, druhý řádek je posunut o tři pozice doprava, třetí řádek o dvě a poslední o jednu pozici. Tato operace je součástí rozptylu, tedy snahy propagovat změny do celého bloku.

$$\begin{array}{|c|c|c|c|} \hline B_0 & B_4 & B_8 & B_{12} \\ \hline B_1 & B_5 & B_9 & B_{13} \\ \hline B_2 & B_6 & B_{10} & B_{14} \\ \hline B_3 & B_7 & B_{11} & B_{15} \\ \hline \end{array} \rightarrow \begin{array}{|c|c|c|c|} \hline B_0 & B_4 & B_8 & B_{12} \\ \hline B_5 & B_9 & B_{13} & B_1 \\ \hline B_{10} & B_{14} & B_2 & B_6 \\ \hline B_{15} & B_3 & B_7 & B_{11} \\ \hline \end{array}$$

Obrázek 2.2: Proces posunutí řádků

Kombinování sloupců

Kombinování sloupců je další nástroj propagace změn. Operace probíhá vždy nad čtveřicemi bytů, které jsou brány jako vektor. Ten je pak vynásoben předem definovanou maticí (rovnice 2.1 ukazuje výpočet první čtveřice bytů).

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix} \tag{2.1}$$

Násobení a sčítání členů při násobení matic neprobíhá podle pravidel klasické aritmetiky, ale podle aritmetiky tzv. *konečného tělesa* (též Galoisova tělesa, značeno $GF(x)$). V AES se jedná o $GF(2^8)$. Prvky tělesa jsou v tomto případě polynomy, které jsou reprezentovány bitovými vektory — např.:

$$2B \sim (00101011) \sim (x^5 + x^3 + x + 1)$$

Použitím operace modulo je pak zajištěno, že konečné výsledky budou také součástí tělesa. Sčítání je definováno jako $(A + B) \, mod \, 2$ (lze si všimnout, že je ekvivalentní operaci XOR), u násobení je to $(A \cdot B) \, mod \, P(x)$, kde $P(x) = x^8 + x^4 + x^3 + x + 1$ podle standardu AES.

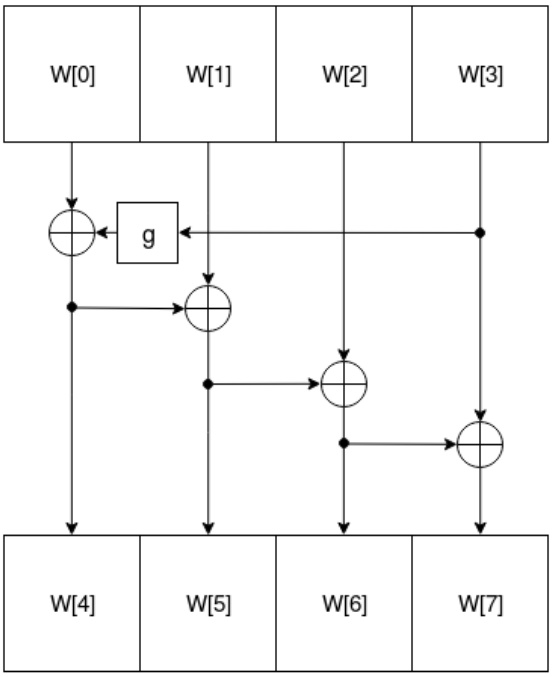
Přidání podklíče

Posledním krokem je přidání podklíče pro danou rundu. Podklíč je stejně jako vnitřní stav 128bitový. Přidání je realizováno pomocí operace XOR, která je ekvivalentní sčítání v konečném tělese.

Odvození podklíčů

Pro fungování algoritmu je potřeba $n + 1$ klíčů, kde n je počet rund. Nultý podklíč se totiž přidává už před první rundou, což zlepšuje bezpečnost šifry (tzv, *key whitening*). Podklíče jsou odvozovány rekurzivně, tedy nelze odvodit žádný podklíč kromě prvního, bez znalosti předchozího podklíče. Na rozdíl od rundové funkce, která operuje s byty (8 bitů), proces odvozování klíčů pracuje se slovy (32 bitů). Procesy odvození podklíčů se liší pro jednotlivé délky klíčů, pro stručnost bude odvození popsáno pouze pro 128bitový klíč.

Předpokládejme pole W, kam se podklíče budou ukládat. Vstupní klíč je rozdělen po slovech na 4 části $W[0] - W[3]$, které jsou zároveň nultým podklíčem. Obrázek 2.3 ilustruje proces odvození 1. podklíče.



Obrázek 2.3: Schéma procesu odvození 1. podklíče

Stejným způsobem se odvozují i všechny následující klíče. Při odvozování každého 4. slova (na schématu je to $W[4]$) se navíc ještě použije funkce $g()$. Ta vstupních 32 bitů nejdříve cyklicky rotuje o 8 bitů doleva, následně nahradí jednotlivé byty podle S-boxu (tabulka 2.8). Posledním krokem funkce je k prvnímu bytu pomocí operace XOR přidat prvek z $GF(2^8)$, který je na pozici 2^{i-1}, kde i je číslo rundy, pro kterou se podklíč počítá.

Dešifrování

Protože AES není šifra založená na Feistelových sítích, nestačí pouze invertovat pořadí rund, ale je nutné také invertovat všechny části rundové funkce. Začíná se rundou, která je poslední u zašifrování, neprovádí se zde tedy operace kombinování sloupců. Je také nutné odvodit podklíč pro tuto rundu, což v praxi znamená vypočítat i všechny podklíče před tímto, protože u zašifrování se jedná o podklíč pro poslední rundu. Za poslední rundou dešifrování se pak přidává pomocí operace XOR nultý podklíč. Následuje přehled způsobu inverze jednotlivých operací :

- **Kombinování sloupců**
  Tuto operaci lze invertovat pomocí inverze fixní matice a vynásobení vstupního vektoru touto maticí. Rovnice 2.2 ilustruje získání prvních 4 bytů, další se získávají obdobným způsobem.

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \cdot \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} \tag{2.2}$$

- **Posunutí řádků**
  Posunutí řádků se invertuje pouhou změnou směru posunutí, místo doprava se byty posunují cyklicky doleva, jak ukazuje schéma 2.4. První řádek se neposunuje, druhý o 3 pozice, třetí o 2 pozice a čtvrtý řádek o jednu pozici.

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}

$\rightarrow$

B_0	B_4	B_8	B_{12}
B_{13}	B_1	B_5	B_9
B_{10}	B_{14}	B_2	B_6
B_7	B_{11}	B_{15}	B_3

Obrázek 2.4: Proces posunutí řádků

- **Bytová substituce**
  Pro bytovou substituci se používá inverzní S-box, který zde není ve snaze o stručnost uveden.

- **Odvození podklíčů**
  Zde se nic nemění, pouze je potřeba odvodit kvůli rekurzivní povaze algoritmu všechny podklíče a následně je od posledního až po nultý postupně aplikovat.

Zhodnocení

Tento algoritmus byl vybrán kvůli své velké rozšířenosti a vysoké bezpečnosti. Dále je velmi dobře optimalizovatelný v hardware i software. Tvůrci AES navrhli variantu algoritmu pro implementaci v softwaru, která místo jednotlivých operací rundové funkce používá 4 velké vyhledávací tabulky (též T-boxy). Tato implementace je ale pro tuto bakalářskou práci zcela nevhodná, neboť se skládá většinou pouze z přístupů do paměti, které nelze efektivně optimalizovat. V současnosti neexistuje známý útok, který by byl efektivnější než brute-force útok, tedy vyzkoušení všech možných klíčů. V současnosti existuje instrukční rozšíření *AES-NI* pro architekturu x86, které využívají některé procesory od firem Intel[1] a AMD[2]. Tyto instrukce implementují rundy při procesech šifrování i dešifrování a také urychlují části generování podklíčů. Může se zde také uplatnit rozšíření *CLMUL* na stejné architektuře, které implementuje násobení nad konečným tělesem $GF(2^8)$.

2.2.4 Blowfish

Algoritmus Blowfish byl vytvořen jako alternativa ke stárnoucímu DES algoritmu v roce 1993 Brucem Schneierem. Záměrem bylo také nabídnout bezpečný algoritmus, který bude volně dostupný a nepatentovaný v době, kdy mnoho šifer bylo proprietárních, nebo neveřejných. Blowfish pracuje s vnitřním stavem o velikosti 64 bitů a má 16 rund. Velikost klíče je 32 až 448 bitů. Implementaci použitou v této práci vytvořil Brad Conte[8]. Tato podkapitola byla převzata z [9].

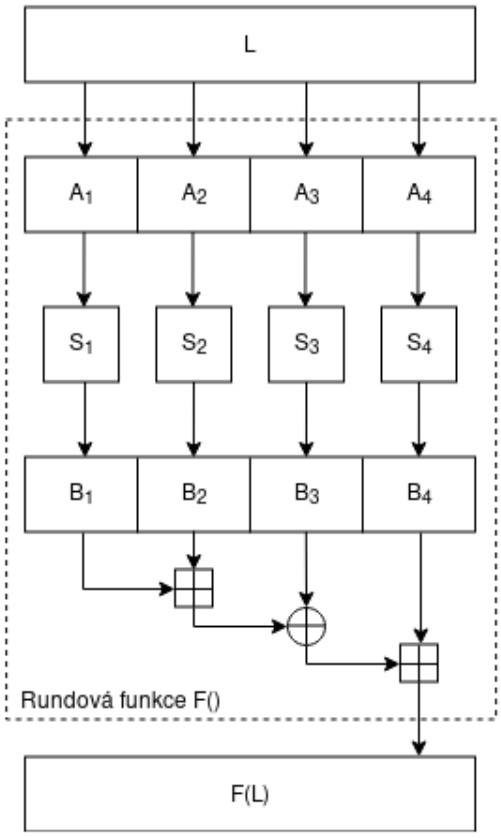
[1] https://ark.intel.com/Search/FeatureFilter?productType=processors&AESTech=true

[2] https://web.archive.org/web/20101126155830/http://blogs.amd.com/work/2010/11/22/following-instructions/

Rundová funkce F

Blowfish disponuje oproti ostatním algoritmům poměrně jednoduchou rundovou funkcí. Funkci ilustruje obrázek 2.5. Vstupem této funkce je 32 bitů, tedy levá půlka vnitřního stavu. Vstup je následně rozdělen po 8 bitech na 4 části. Každá část je pak vstupem jednoho ze čtyř S-boxů. Výstup prvního S-boxu je pak přičten k výstupu druhého S-boxu, nad výsledkem a výstupem třetího S-boxu je pak prováděna operace XOR a nakonec je výsledek této operace přičten k výstupu posledního, čtvrtého S-boxu. Nad výsledky sčítání je navíc prováděná operace modulo 256. V implementaci je ale výhodnější použít ekvivalentní logickou operaci AND 255.

$$F(L) = ((B_1 + B_2 \, mod \, 2^{32}) \oplus B_3) + B_4 \, mod \, 2^{32}$$



Obrázek 2.5: Schéma rundové funkce u Blowfish

Odvození podklíčů a S-boxů

Podklíčů je celkem 18, každý o 32 bitech, a jsou uloženy v poli P. 16 z nich jsou rundové podklíče a 2 z nich se po poslední rundě pomocí operace XOR použijí každý na jednu půlku 64bitového vnitřního stavu. S-boxy jsou celkem 4, každý o 256 hodnotách. Stejně jako S-boxy je i pole P inicializováno na hexadecimální reprezentace čísel následující za desetinou

čárkou u π. Tato čísla byla vybrána proto, aby nevzniklo podezření, že si tvůrce šifry vytváří "zadní vrátka", která by později mohl zneužít. Před samotným startem algoritmu je nutné přegenerovat podklíče a S-boxy. Algoritmus přegenerování je následující:

1. První prvek pole P (P_1) se zkombinuje s prvními 32 bity klíče. Taktéž P_2 s dalšími 32 bity klíče a tento proces pokračuje pro všechny bity klíče, dokud nejsou změněny všechny prvky pole. Protože ani nejdelší možný klíč (448 bitů) nepokryje celé pole P, klíč se zopakuje tolikrát, kolikrát je potřeba. Pro krátké klíče to znamená, že existují jejich delší ekvivalenty. Například pro 64bitový klíč existuje ekvivalentní 128bitový, 192bitový, atd.

2. Pomocí podklíčů vygenerovaných v prvním kroku se zašifruje 64 nulových bitů.

3. Zašifrovaným výsledkem se nahradí P_1 a P_2

4. Výstup kroku 2 se znovu zašifruje

5. Zašifrovaným výsledkem se nahradí P_3 a P_4

6. Proces se opakuje, dokud nejsou nahrazeny všechny prvky pole P a všechny S-boxy.

Celkem je k přegenerování potřeba 521 iterací, které dohromady vygenerují 4168 bytů.

Zhodnocení

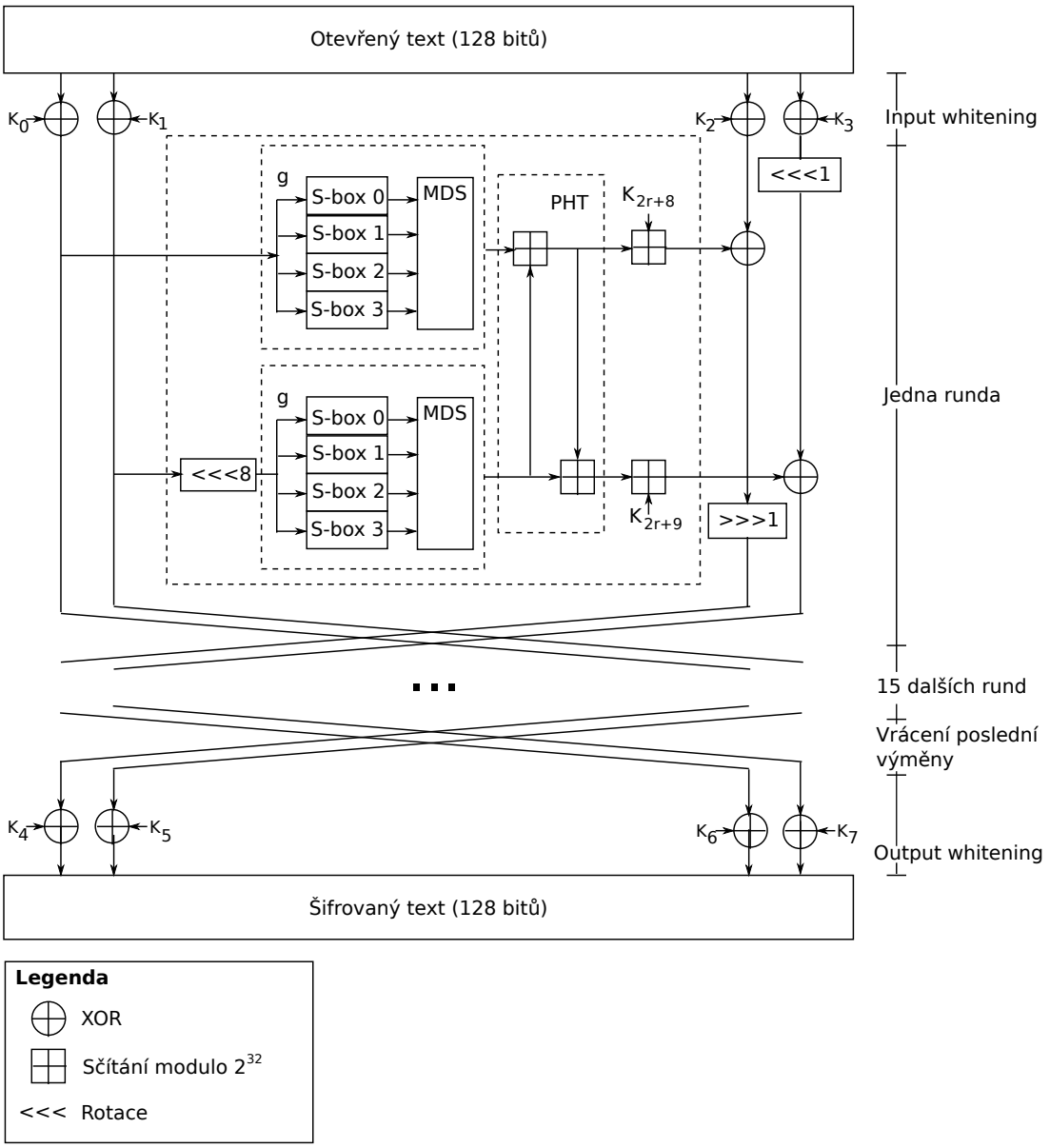
Hlavní nevýhodou tohoto algoritmu je velká výpočetní a prostorová náročnost při změně klíče, kde se při přípravě S-boxů a podklíčů musí zašifrovat více jak 4 kB textu. To nicméně neznemožňuje efektivní užití algoritmu. Právě kvůli této vlastnosti ho například využívá FreeBSD k šifrování hesel, protože ztěžuje brute-force útoky. Šifra jako taková je neprolomená, ale kvůli svému malému vnitřnímu stavu je zranitelná tzv. *birthday attacks*[5], podobně jako 3DES. Algoritmus je významný hlavně svojí publikací jako volné dílo v době, kdy mnoho šifrovacích algoritmů bylo proprietárních, nebo dokonce tajných.

2.2.5 Twofish

Twofish je jedním z finalistů veřejné soutěže o standardizaci na AES. Jedním z jeho tvůrců je Bob Schneier, který vytvořil předchůdce tohoto algoritmu, Blowfish. Algoritmus byl poprvé publikován v roce 1998, a přestože nezvítězil v soutěži o nový kryptografický standard, je zajímavý svým fungováním a faktem, že byl publikován jako volné dílo, stejně jako algoritmus Blowfish. Twofish je Feistelova síť s 16 rundami, klíči do velikosti až 256 bitů a 128bitovým vnitřním stavem. Implementace používaná v této práci je implementace, kterou používá GNU ZRTP[10]. Tato podkapitola byla převzata z [11].

Key whitening

Key whitening je proces zvyšování bezpečnosti šifry používáním operace XOR na části vnitřního stavu a některé klíče, typicky před první rundou a po poslední rundě. Tak je tomu i v případě Twofish, kde se před první rundou použijí čtyři 32bitové podklíče K_0 – K_3 a po poslední rundě K_4 – K_7.



Obrázek 2.6: Schéma algoritmu Twofish

Funkce g

Vstupem funkce g je prvních 32 bitů vnitřního stavu. Tento vstup je dále rozdělen na jednotlivé byty, které jsou pak substituovány pomocí čtyř S-boxů. Výstupy S-boxů jsou pak použity jako vektor a násobí se s MDS maticí (rovnice 2.3). Podobně jako u AES je zde využito tzv. *konečného tělesa*, konkrétně $GF(2^8)$. Pravidla pro operace platí stejná, s tím rozdílem, že u násobení (které je definováno jako $(A \cdot B)\, mod\, P(x)$) je polynom definován

jako $P(x) = x^8 + x^6 + x^5 + x^3 + 1$. Výstupem funkce je opět 32 bitů.

$$\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{pmatrix} = \begin{pmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{pmatrix} \cdot \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \tag{2.3}$$

Rundová funkce f

Jak je vidět na schématu 2.6, vstupem rundové funkce je celý 128bitový vnitřní stav. Ten je rozdělen na čtyři části po 32 bitech. První část (A_0) se stává vstupem funkce g. Druhá část (A_1) je rotována doleva o 8 bitů a také prochází funkcí g. Část ve schématu označená jako PHT značí pseudo-Hadamardovu transformaci, která poskytuje výpočetně nenáročný rozptyl. Výstupy rundové funkce jsou tedy:

$$C_0 = (B_0 + B_1 + K_{2r+8}) \, mod \, 2^{32}$$
$$C_1 = (B_0 + 2B_1 + K_{2r+9}) \, mod \, 2^{32}$$

Kde B_n jsou výstupy funkce g, K_n je n-tý podklíč a r je aktuální runda. Na závěr je provedeno:

$$A_2 = (A_2 \oplus C_0) \ggg 1$$
$$A_3 = (A_3 \lll 1) \oplus C_1$$

Poté je změněno pořadí jednotlivých částí vnitřního stavu a pokračuje se další rundou.

Odvození podklíčů a S-boxů

Twofish má podobně jako Blowfish S-boxy závislé na použitém klíči, a musí je tedy generovat spolu se 40 podklíči. Šifrovací klíč může být délky $N = 128$, $N = 192$ a nebo $N = 256$ bitů, jiné délky musí být prodlouženy nulami na nejbližší ze zmíněných. Definujme k jako $k = \frac{N}{64}$. Klíč je nejprve rozdělen po bytech na m_0 až m_{8k-1}. Z těchto bytů se pak vypočítají 32bitová slova podle následujícího vzorce:

$$M_i = \sum_{j=0}^{3} m_{4i+j} \cdot 2^{8j} \qquad i = 0, \ldots, 2k-1$$

Z těchto slov jsou složeny dva vektory, jeden ze sudých slov, jeden z lichých:

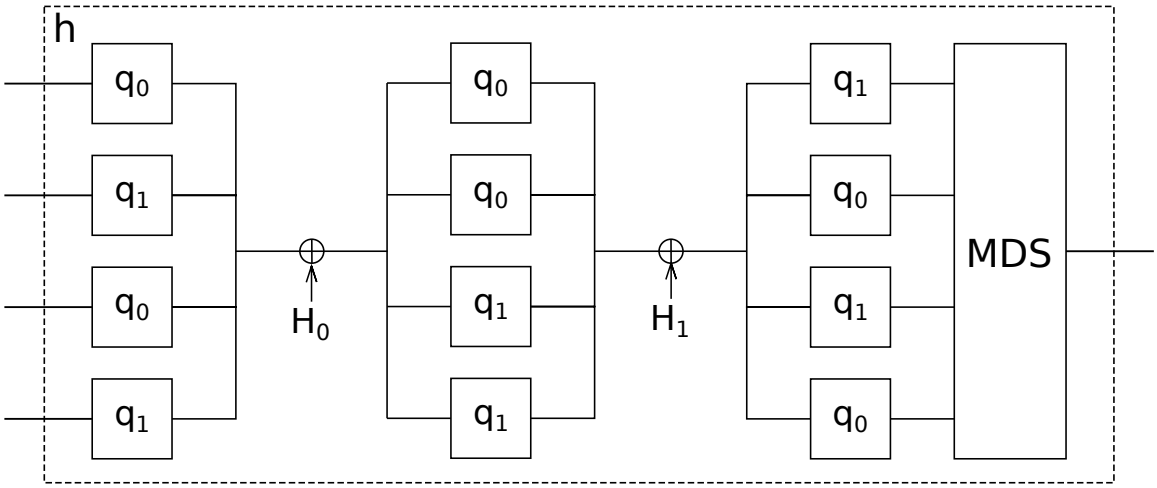
$$M_e = (M_0, M_2, ...)$$
$$M_o = (M_1, M_3, ...)$$

Třetí vektor S délky k je vypočítán pomocí následujícího vzorce:

$$\begin{pmatrix} s_{i,0} \\ s_{i,1} \\ s_{i,2} \\ s_{i,3} \end{pmatrix} = \begin{pmatrix} 01 & A4 & 55 & 87 & 5A & 58 & DB & 9E \\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5 \\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19 \\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03 \end{pmatrix} \cdot \begin{pmatrix} m_{8i} \\ m_{8i+1} \\ m_{8i+2} \\ m_{8i+3} \\ m_{8i+4} \\ m_{8i+5} \\ m_{8i+6} \\ m_{8i+7} \end{pmatrix}$$

$$S_i = \sum_{j=0}^{3} s_{i,j} \cdot 2^{8j} \qquad i = 0, \ldots, k-1$$

Jednotlivá slova jsou pak ve vektoru seřazena opačně: $S = (S_{k-1}, \ldots, S_0)$. Následuje popis procesu odvození podklíčů a S-boxů pro klíč o velikosti N=128 bitů. Pro delší klíče je proces obdobný, pouze prodloužený. Základem tohoto procesu je funkce h, kterou ilustruje schéma 2.7. Vstupem funkce je 32bitové slovo a v případě $k = 2$ dva další vstupy. Vstupní slovo je rozděleno po bytech a prochází fixními permutačními boxy q_0 a q_1, které zde nejsou do detailu popsány, ale provádí několik substitucí, rotací a XOR operací. Na konci schématu se pak násobí MDS maticí, stejně jako u funkce g.



Obrázek 2.7: Schéma funkce h algoritmu Twofish

Tato funkce se používá při výpočtu S-boxů i jednotlivých podklíčů. S-boxy se obvykle předpočítávají při inicializaci algoritmů a vychází ze vztahu:

$$g(X) = h(X, S)$$

Při výpočtu klíčů se operuje s konstantou $\rho = 2^{24} + 2^{16} + 2^8 + 2^0$, která se pro násobení hodnotami $0 \leq i \leq 255$ skládá ze čtyř identických bytů, které mají hodnotu i. Podklíče K_{2i} a K_{2i+1} lze pak odvodit takto:

$$\begin{aligned} A_i &= h(2i\rho, M_e) \\ B_i &= h((2i+1)\rho, M_o) \lll 8 \\ K_{2i} &= (A_i + B_i) \, mod \, 2^{32} \\ K_{2i+1} &= ((A_i + 2B_i) \, mod \, 2^{32}) \lll 9 \end{aligned}$$

Zhodnocení

Twofish sice nezvítězil v soutěži o standardizaci jako AES, přesto je k současnému AES dobrou alternativou. V současnosti není tak výhodné ho používat, protože pro AES existují na architektuře x86 standardizovaná instrukční rozšíření. Sami jeho tvůrci ho doporučují jako náhradu jeho předchůdce, algoritmu Blowfish. V současnosti neexistuje ani teoretický útok, který by byl schopen šifru napadnout. Twofish je stejně jako jeho předchůdce volným dílem.

2.3 Asymetrické algoritmy

Tato podkapitola byla převzata z [2].

Principy asymetrické kryptografie

Asymetrická kryptografie řeší nedostatky symetrické kryptografie, která se vyznačuje jedním sdíleným klíčem, který musí obě strany udržovat v tajnosti a musí si ho nejdříve sdělit nějakým nezabezpečeným způsobem - například osobně nebo přes důvěryhodnou třetí stranu. Asymetrická kryptografie tento neduh eliminuje zavedením dvou klíčů pro každého uživatele. Jeden klíč je tzv. privátní - tento klíč by měl zůstat tajný a měl by ho znát jen sám jeho majitel. Druhý klíč je tzv. veřejný - tento klíč může znát neomezeně mnoho uživatelů. Tyto klíče jsou navrženy tak, aby mezi sebou měly matematický vztah a společně se používají na šifrování a dešifrování dat a podepisování zpráv.

Šifrování a dešifrování dat

Mějme dva subjekty (A a B), které si chtějí bezpečným způsobem vyměnit důvěrná data (x). Každý z těchto subjektů má pár klíčů, veřejný a soukromý:

$$A = (a_{pub}, a_{priv})$$

$$B = (b_{pub}, b_{priv})$$

Subjekt A posílá data x subjektu B. Provede tedy zašifrování veřejně známým veřejným klíčem b_{pub} subjektu B.

$$y = encrypt(x, b_{pub})$$

Zašifrovaná data y pošle subjektu B. Ten provede dešifrování pomocí svého privátního klíče b_{priv} a získá tím opět data x.

$$x = decrypt(y, b_{priv})$$

Je tedy vyřešen problém zabezpečení dat. Stále ale existuje problém verifikace odesílatele, tedy že odesílatelem je skutečně subjekt A.

Digitální podpis

Uvažujme situaci z předchozí sekce. Je potřeba zajistit, aby subjekt B byl schopen verifikovat, že data skutečně odeslal subjekt A. Tento problém řeší koncept digitálního podpisu. V praxi to znamená dvojité zašifrování a následné dvojité dešifrování dat. Subjekt A kromě zašifrování veřejným klíčem b_{pub} subjektu B zašifruje data i svým privátním klíčem a_{priv}.

$$y = encrypt(encrypt(x, b_{pub}), a_{priv})$$

Subjekt B pak musí provést dešifrováním pomocí veřejného klíče a_{pub} subjektu A a svého privátního klíče.

$$x = decrypt(decrypt(y, a_{pub}), b_{priv})$$

Na pořadí šifrování a dešifrování nezáleží, je jen nutné, aby bylo provedeno v reverzním pořadí.

2.3.1 RSA

RSA je kryptografický algoritmus z roku 1977, který jako první implementoval principy asymetrické kryptografie, které byly v roce 1976 patentovány Whitfieldem Diffie a Martinem Hellmanem. Paralelně s jejich objevem byly objeveny tyto principy i v rámci práce pro britskou vládu v GCHQ, ale nebyly publikovány, protože podléhaly utajení. Implementace použitá v této práci pochází z mbedTLS knihovny[12].
Algoritmus zahrnuje následující kroky:

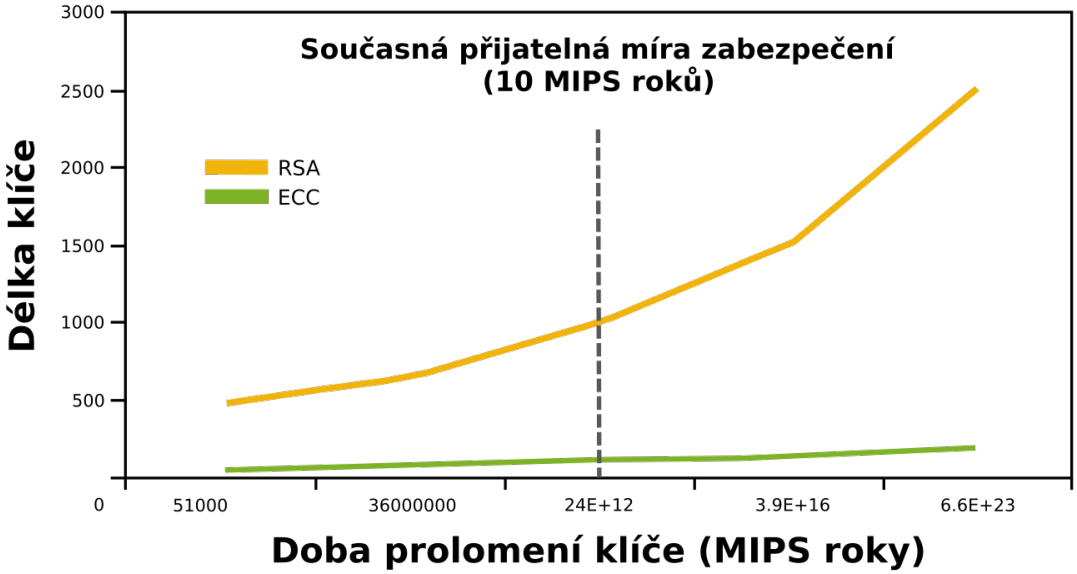
1. *Generování klíčů*
    1. Vyberou se dvě vysoká prvočísla p a q, která si nejsou navzájem rovna.
    2. Vypočítá se n jako $n = p * q$.
    3. Vypočítá se $\varphi(n) = \varphi(p) * \varphi(q) = (p-1) * (q-1)$ kde $\varphi(n)$ je Eulerova funkce[1].
    4. Vybere se celé číslo e, pro které platí $1 < e < \varphi(n)$ a zároveň platí, že největším společným dělitelem e a $\varphi(n)$ je 1 (tedy že čísla jsou nesoudělná)
    5. Vypočítá se d jako modulární inverz $e(\mathrm{mod}(\varphi(n)))$, tedy $d = e^{-1} \bmod \varphi(n)$
    6. Pár (e, n) se stává veřejným klíčem
    7. Pár (d, n) se stává privátním klíčem
2. *Zašifrování*
   Zašifrování probíhá pomocí veřejného klíče příjemce zprávy (pár (e, n)). Zprávu M lze zašifrovat následovně:
   $$C = M^e (\mathrm{mod}\, n)$$
   Pro příklad uvažujme veřejný klíč $(7, 143)$ a zprávu $M = 8$.
   $$C = 8^7 \bmod 143 = 57$$
3. *Dešifrování*
   Dešifrování probíhá stejně jako zašifrování, s tím rozdílem, že tentokrát se použije privátní klíč (pár (d, n)). Dešifrování šifrované zprávy C tedy probíhá následovně:
   $$M = C^d (\mathrm{mod}\, n)$$
   V našem příkladu uvažujme privátní klíč $(103, 143)$.
   $$M = 57^{103} \bmod 143 = 8$$

Zhodnocení algoritmu RSA

Algoritmus používá délku klíčů typicky od 1024 do 4096 bitů. Takto dlouhé klíče jsou nezbytné pro zachování neprolomitelnosti tzv. *brute-force* útokem, tedy vyzkoušením všech možných klíčů. Toto bohužel z RSA dělá velmi výpočetně náročný algoritmus a použitelným se stává spíše po kratší zprávy, nebo pro delší zprávy, které jsou rozdělené na menší úseky a po zašifrování spojené dohromady. Příjemce v tomto případě musí znát i způsob dělení původní zprávy. Problém velikosti klíče RSA řeší kryptografie eliptických křivek, která je

[1] Eulerova funkce $\varphi(n)$ je aritmetická funkce, která najde všechny nesoudělná čísla k n, která jsou menší nebo rovna n.

schopna zaručit téměř ekvivalentní bezpečnost jako u algoritmu RSA s klíčem délky 1024 bitů, a to s klíčem délky pouze 163 bitů. Toto výrazné zvýšení bezpečnosti při kratších klíčích lze pozorovat na grafu 2.8, kde je označena současná hranice neprolomitelnosti. Pro RSA existují na architektuře x86 instrukce[2], které usnadňují aritmetiku velkých čísel zavedením instrukcí pro operace které tento algoritmus typicky využívá. Tento algoritmus byl vybrán pro své velké rozšíření, používá se například v implementaci protokolu SSH OpenSSH, nebo v populárním kryptografickém software GPG.



Obrázek 2.8: Porovnání prolomitelnosti algoritmů ECC a RSA pro různé délky klíčů.[13]

[2] https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/large-integer-squaring-ia-paper.pdf

Kapitola 3

Architektura RISC-V

RISC-V je otevřená architektura instrukční sady (též ISA), která byla vytvořena na University of California, Berkeley a nyní ji spravuje a rozvíjí RISC-V Foundation. Umožňuje komukoliv tuto architekturu používat a prodávat pod licencí. Byla navržena s ohledem na škálovatelnost, ve své finální podobě by měla být použitelná na různých zařízeních, od mobilních a vestavěných zařízení až po cloudové počítače. Instrukční sada je navržena tak, aby bylo možné podporovat 32bitové, 64bitové a v budoucnosti i 128bitové varianty.

Tato architektura byla pro tuto práci vybrána z důvodu své aktuálnosti a kvůli svému návrhu, který přímo počítá s rozšiřitelností. Tato kapitola čerpá ze zdroje [14].

3.1 Instrukční sada

Základní sada

RISC-V definuje základní instrukční sadu pro 32bitové procesory (**RV32I**) a pro 64bitové procesory (**RV64I**). Zároveň jsou v současné době ve vývoji základní sady pro 128bitové procesory (**RV128I**) a pro vestavěná (embedded) zařízení (**RV32E**). Základní instrukční sada musí být přítomna v každé implementaci. Je to nezbytné minimum pro použití obecného kompilátoru. Obsahuje 47 instrukcí, které umožňují aritmetické operace, přístupy do paměti, větvení, skoky a systémové operace. Základní sada umožňuje pouze celočíselné instrukce. Tabulka 3.1 popisuje instrukční sadu **RV32I**. Jsou tu zastoupeny aritmetické instrukce, jako jsou logické bitové operace, sčítání a odčítání. Tyto instrukce jsou definovány pro tři registry a také pro dva registry a přímý operand. Dále jsou zde instrukce pro větvení programu, podmíněné i nepodmíněné skoky. Sada také obsahuje instrukce pro přístup do paměti, jsou zde instrukce pro čtení různých velikostí dat z paměti s možností načítání unsigned čísel (bez znaménka). Dále je zde instrukce `AUIPC`, která počítá offset z `pc` registru a konstanty. Instrukce pro ukládání do paměti jsou pouze tři, pro byte (8b), půlslovo (16b) a slovo (32b). Sada poskytuje také přístup k čítačům cyklů procesoru a k reálnému času. Dalšími instrukcemi jsou `FENCE`, které se používají synchronizaci tzv. *harts* (Hardware threads) při přístupech do paměti. Nakonec jsou zde ještě dvě instrukce pro implementaci systémových volání a podporu ladění. Nejsou zde pro stručnost zmíněny CSR (Control and Status Register) instrukce pro ovládání stavových a kontrolních registrů, kterých je pro neprivilegovaný přístup dostupných jen několik a to pouze ke čtení.

ALU instrukce		**Skoky**		**LOAD/STORE**		**Čítače**	**Ostatní**
3 registry	**konstanta**	**bez podmínky**	**s podmínkou**	**LOAD**	**STORE**		
ADD	ADDI	JAL	BEQ	LW	SW	RDCYCLE	FENCE
SUB		JALR	BNE	LH	SH	RDCYCLEH	FENCE.I
AND	ANDI		BLT	LHU		RDTIME	ECALL
OR	ORI		BLTU	LB	SB	RDTIMEH	EBREAK
XOR	XORI		BGE	LBU		RDINSTRET	
SLL	SLLI		BGEU	LUI		RDINSTRETH	
SRL	SRLI			AUIPC			
SRA	SRAI						
SLT	SLTI						
SLTU	SLTIU						

Tabulka 3.1: Seznam instrukcí sady **RV32I**

Instrukční sada **RV32E** pro vestavěná zařízení používá stejnou instrukční sadu jako **RV32I**, ale má pouze 16 registrů. Instrukční sada **RV64I** rozšiřuje 32bitové registry na 64bitové, přidává instrukce pro manipulaci s hodnotami této velikosti a upravuje některé instrukce z **RV32I**.

Rozšiřující sady

Rozšiřující sady se dělí na standardizované a nestandardizované. Standardizovaná rozšíření jsou spíše obecného rázu a rozšíření nejsou mezi sebou konfliktní, zatímco nestandardizovaná rozšíření by měla být vysoce specializovaná a mohou u nich nastávat konflikty a to i se standardizovanými rozšířeními. Tabulka 3.2 poskytuje stručný přehled standardních rozšíření. Tučně označená rozšíření spolu se základní sadou (**IMAFD**) tvoří dohromady obecnou instrukční sadu, též označovanou jako **G**. Nutno poznamenat, že v současné době jsou některá z těchto rozšíření teprve ve fázi návrhu.

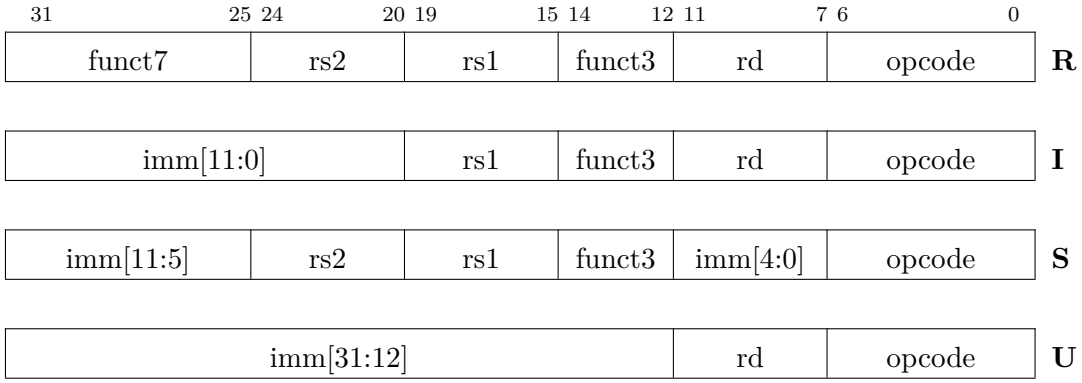
Pro tuto práci byl použit 32bitový model Codix Berkelium, který poskytuje společnost Codasip, a který odpovídá instrukční sadě **RV32IM**.

Název	Popis
M	Instrukce pro násobení a dělení
A	Atomické instrukce pro podporu vícejádrových procesorů
F	Podpora čísel s plovoucí řádovou čárkou s jednoduchou přesností
D	Podpora čísel s plovoucí řádovou čárkou s dvojitou přesností
Q	Podpora čísel s plovoucí řádovou čárkou s čtyřnásobnou přesností
L	Instrukce pro decimální aritmetiku
C	Komprimované instrukce
B	Instrukce pro bitové operace
J	Rozšíření pro dynamicky překládané jazyky
T	Instrukce pro operace s transakční pamětí
P	Podpora pro SIMD instrukce
V	Instrukce pro vektorové operace
N	Podpora uživatelských přerušení

Tabulka 3.2: Seznam standardních rozšíření pro RISC-V

3.2 Formát instrukcí

Tato podkapitola se zabývá formátem instrukcí v **RV32I**. Ty mají pevně daný formát a délku 32 bitů, ale při návrhu rozšíření lze využít i delší, či kratší délky instrukcí, pokud je jejich délka zarovnaná na násobky 16 – příkladem může být rozšíření **C**, které používá komprimované 16bitové instrukce. RISC-V umí efektivně rozlišovat délku instrukce, protože zavádí konvenci jejich kódování na nejnižších bitech instrukce – 16bitové instrukce mají nejnižší bity 00, 01 nebo 10, 32bitové instrukce začínají 11 a následující tři bity nesmí být 111. Konvence podobně definuje i další délky. Následuje přehled formátů kódování instrukcí pro **RV32I** a detailní vizualizace na schématu 3.1



Obrázek 3.1: Formáty instrukcí pro RV32I

- **R** – nejjednodušší formát, který využívá tři registry, dva zdrojové a jeden cílový
- **I** – formát, který používá 12bitový přímý operand a dva registry, zdrojový a cílový
- **S** – formát se dvěma zdrojovými registry a rozděleným 12bitovým přímým operandem, kvůli zjednodušení dekodéru
- **U** – formát s pouze jedním zdrojovým registrem a 20bitovým přímým operandem

Registry

RISC-V definuje celkem 31 uživatelsky dostupných obecných registrů a k tomu nultý registr, který obsahuje konstantní nulu. Velikost registru je pak dána základní instrukční sadou, tedy 32 nebo 64 bitů, případně 128 bitů. Výjimkou je sada **RV32E**, která má celkem pouze 16 registrů. Dále je pak přístupný registr `pc`, který ukládá adresu aktuální instrukce. Další registry mohou být dostupné v rámci jednotlivých rozšíření, např. floating-point registry pro rozšíření F, nebo vektorové registry pro rozšíření V.

3.3 Příklad strojového kódu

V této sekci bude na příkladu jednoduchého programu v jazyce C ukázán neoptimalizovaný strojový kód, který vznikne přeložením tohoto programu.

```
1 int main(){
2   int a = 3;
3   int b = 5;
4   int c = a + b;
5   return c;
6 }
```

Zdrojový kód 3.1: Kód v C

```
1  main:
2    sw x0, 20 ( sp )
3    add x3, x0, 3
4    sw x3, 16 ( sp )
5    add x3, x0, 5
6    sw x3, 12 ( sp )
7    lw x3, 16 ( sp )
8    lw x4, 12 ( sp )
9    add x3, x4, x3
10   sw x3, 8 ( sp )
11   lw x10, 8 ( sp )
12   add sp, sp, 32
```

Zdrojový kód 3.2: Strojový kód pro RISC-V

Jedná se pouze o definici dvou proměnných, přiřazení jejich hodnot, následné sečtení, uložení výsledku do třetí proměnné a vrácení této hodnoty. Vedle je pak vidět vygenerovaný strojový kód pro RISC-V (respektive jeho relevantní část). Lze si povšimnout, že zde není žádná `mov` instrukce. Není totiž potřeba, protože instrukci `mov x3 , 3` lze nahradit za `add x3, x0, 3`, protože registr `x0` je zapojen jako konstantní nula. Kromě instrukce `add`, která je zde použita jak pro přesun hodnot do registrů, tak i pro sčítání mezi registry, jsou zde už jen instrukce `lw` a `sw` pro čtení a zápis 32bitových hodnot do paměti, konkrétně na zásobník.

Kapitola 4

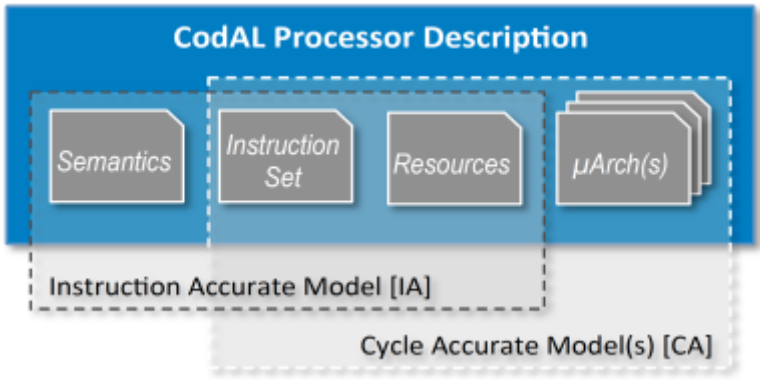
Codasip Studio a jazyk CodAL

4.1 Codasip Studio

V této kapitole je čerpáno ze zdroje [15]. Codasip Studio je integrované vývojové prostředí založené na Eclipse zaměřené na vývoj a návrh procesorů. Obsahuje nástroje na generování SDK pro daný model procesoru, poskytuje také možnost generování hardwarové reprezentace modelu v jazycích VHDL nebo Verilog. Stavebním kamenem Codasip Studia je nástroj *Codasip CommandLine*, který lze používat i samostatně v příkazové řádce. Následuje přehled částí generovaného SDK:

- **Assembler** – Překlad strojových instrukcí na objektové binární soubory.
- **Disassembler** – Překlad z objektových souborů zpět na strojové instrukce, které jsou platným vstupem pro Assembler.
- **Linker** – Nástroj pro spojování objektových souborů, nezávislý na cílové architektuře.
- **C/C++ překladač** – Je založen na frameworku LLVM, který umožnuje lepší optimalizaci překladače. Obsahuje standardní knihovnu jazyka C Newlib, která je zaměřená na vestavěné systémy. Umožňuje překlad zdrojových kódů do strojového kódu.
- **Simulátor** – Umožňuje testování a ladění modelů procesorů.
- **Debugger** – Umožňuje spustit simulaci s laděním na různých úrovních, od zdrojového kódu, přes strojový kód až po popis procesoru v jazyce CodAL.
- **Profiler** – Umožňuje profilovat aplikaci pro daný model procesoru a to až na úroveň využití jednotlivých instrukcí a anotaci zdrojových kódů

Codasip Studio umožňuje definovat dva druhy popisu procesoru. Prvním z nich je *Instruction Accurate* model, který definuje instrukční sadu bez hardwarové implementace. Je vhodný na rychlé prototypování instrukční sady a generují se z něj nástroje, jako je překladač, simulátor apod. Pak přichází na řadu *Cycle Accurate* model, který popisuje hardwarovou implementaci instrukční sady a je z něj možné generovat RTL reprezentaci modelu. Rozdíly mezi oběma přístupy ilustruje obrázek 4.1. Lze na něm vidět, že *Instruction Accurate* popis zahrnuje sémantiku, instrukční sadu a zdroje, jako jsou např. paměť. Oproti tomu *Cycle Accurate* popis nezahrnuje sémantiku, ale zato přidává popis mikroarchitektur. V této práci je použit pouze Instruction Accurate model, protože cílem práce je pouze navrhnout rozšíření instrukční sady.



Obrázek 4.1: Rozdíly mezi typy modelů – zdroj [16]

4.2 CodAL

V této kapitole je čerpáno ze zdroje [16]. CodAL (Codasip Architecture Language) je jazyk pro popis architektury procesorů, který umožňuje současný návrh software i hardware. Syntaxe vychází z jazyka C a VHDL. Stavebním kamenem modelu procesoru je `element`. Pomocí tohoto konstruktu lze mimo jiné definovat nové a upravit stávající instrukce. Dále ho lze použít k definici operandů, registrů a dalších součástí modelu. `element` má následující sekce:

- `assembler` – definuje textovou reprezentaci pro assembler
- `binary` – definuje formát binární reprezentace elementu pro nástroje v SDK
- `return` – definuje návratovou hodnotu pro další elementy v modelu
- `semantics` – definuje chování elementu

Příkladem instrukce definované v tomto jazyce může být instrukce `XOR` definovaná jako typ **R** pro architekturu RISC-V.

```
1  element i_xor
2  {
3      use opc_xor as opc;
4      use reg_any as dst, src1, src2;
5      assembler { opc dst "," src1 "," src2 };
6      binary {opc[16..10] src2 src1 opc[9..7] dst opc[6..0]};
7      semantics
8      {
9          uint32 result;
10
11         result = rf_gpr_read(src1) ^ rf_gpr_read(src2);
12         rf_gpr_write(result, dst);
13     };
14 };
```

Zdrojový kód 4.1: Instrukce XOR v jazyce CodAL

Na úvod je použito několik `use` výrazů, které deklarují použití dalších elementů v této instrukci, konkrétně opkód, zdrojové operandy a cílový operand. Následuje `assembler` sekce,

která definuje podobu instrukce ve strojovém kódu. Sekce `binary` definuje binární podobu instrukce. Lze si povšimnout, že 16bitový opkód je podle formátu **R** rozdělen na tři části. Sekce `semantics` pak definuje samotné chování instrukce, tedy provedení operace XOR a uložení výsledku.

Kapitola 5

Instrukční rozšíření

5.1 Akcelerace na úrovni hardware

Akcelerace kryptografických algoritmů pomocí hardware je opačný přístup k urychlení algoritmů, než kterým se ubírá tato práce. Tento přístup umožňuje dosahovat velmi dobrých výsledků využitím hardware, který je navržen přímo pro danou aplikaci, a urychluje některé její části[1]. Tento přístup má však i svoje nevýhody, jako je větší režie, nutnost přenášet data z/do paměti akcelerátoru, nebo nemožnost měnit algoritmus, pro který je akcelerátor vytvořen. Rozšíření instrukční sady oproti tomu umožňuje větší flexibilitu. U instrukcí nelze často dosáhnout tak masivního urychlení jako u akcelerátorů, ale vývoj nových instrukcí je podstatně jednodušší, než vývoj nového hardware.

5.2 Proces tvorby nových instrukcí

5.2.1 Spuštění pod `GCC`

Nejdříve je potřeba ověřit fungování samotné aplikace pod běžným překladačem jako je například `GCC`. Tento krok je nutný pro eliminaci chyb, které jsou způsobeny špatným způsobem překladu nebo chybami v samotné aplikaci. Je vhodné využívat přepínače `--Wall` pro upozornění na většinu možných chyb v aplikaci, `-O3` pro maximální optimalizaci. Pokud je překlad úspěšný a aplikace nekončí chybovým kódem, lze se přesunout k návrhu verifikačního systému.

5.2.2 Verifikační systém

Verifikační systém je podstatnou součástí návrhu nových instrukcí. Zajišťuje ověření správnosti nově přidaných instrukcí, tedy pokud aplikaci rozbíjí, měl by na to upozornit. Obecně jde o vytvoření testovací sady. Pokud se zaměříme přímo na kryptografické algoritmy, je zde verifikace o něco jednodušší, neboť k algoritmům bývají poskytovány tzv. *testovací vektory*, tedy dvojice otevřený text – zašifrovaný text, které jsou určeny pro verifikace implementací těchto algoritmů. V některých případech implementace už obsahuje test sama sebe a pro verifikaci tedy stačí pouze volat tuto funkci. Verifikaci je nutné provádět při každé úpravě instrukční sady nebo jiných parametrů modelu procesoru.

5.2.3 Překlad, simulace a profilace

Překlad

Při prvním spuštění nové aplikace je dobré ji nejprve přeložit s nástroji vygenerovanými z neupraveného modelu procesoru. Aplikaci je nutné překládat s přepínači `-lsim`, který nahrává knihovnu, která implementuje systémová volání a `-Wl,--defsym,_HEAP_SIZE=0x80000`, který zvyšuje velikost hromady, protože výchozí hodnota je nulová. Dále je nutný přepínač `-O3`, který povoluje maximální optimalizace překladače, protože nemá cenu vytvářet nové instrukce pro části programu, které je překladač sám schopen optimalizovat. Posledním přepínačem je `--save-temps`, který ukládá mezivýstupy překladače, jako je strojový kód, který je vhodný pro pozdější analýzu.

Simulace

Pokud je přeložení úspěšné, lze aplikaci simulovat na daném modelu. Vždy by se měla nejdříve odsimulovat testovací sada, která je ustanovena při tvorbě verifikačního systému, a teprve v případě úspěchu samotná aplikace. Simulátor generuje soubor s návratovým kódem aplikace, přes který je možné ověřovat úspěšnost simulace.

Profilace

Poté přichází na řadu profilace. Simulátor mimo jiné generuje soubor `codasip.prof`, který umožňuje profilaci. Profiler je pro účely této práce vhodné spouštět s přepínači `--source-code-coverage`, který umožňuje pokrytí kódu `--annotate-asm`, který povoluje anotování strojového kódu a `--annotate-src`, který umožňuje anotovat zdrojový kód. Profiler poté vygeneruje soubor `profiler.txt`, který obsahuje informace o pokrytí kódu a o zatížení, které způsobují jednotlivé funkce. Dále je pak vytvořena složka s anotovanými zdrojovými kódy.

5.2.4 Analýza

Analýza probíhá nad soubory se strojovým kódem, jak těmi vygenerovanými překladačem, tak i anotovanými zdrojovým kódem. Anotace poskytují informace o počtu cyklů strávených na dané instrukci a procentuální podíl na celkovém počtu cyklů. Cílem analýzy je objevit místa, která lze akcelerovat pomocí nových instrukcí, případně verifikovat použití nové instrukce překladačem. Vhodnými místy pro nové instrukce jsou sekvence logických operací nebo místa, kde překladač musí instrukce emulovat. Naopak např. přístupy do paměti lze optimalizovat jen obtížně, a to obvykle za cenu změn v HW, jako je např. víceportová paměť. Níže je uvedena zkrácená ukázka z implementace algoritmu AES (zdrojový kód 5.1). Jedná se o funkci `xtime()`, která je součástí fáze kombinování sloupců v rundové funkci tohoto algoritmu.

```
1  static uint8_t xtime(uint8_t x)
2  {
3          return ((x<<1) ^ (((x>>7) & 1) * 0x1b));
4          /* 0x015e0 31230 31.409% */
5          /*
6          0x015e0    117  0.118%  sw x20 , 32 ( sp )
7          0x015e4    117  0.118%  and x22 , x3 , 255
8          0x015e8    117  0.118%  and x3 , x4 , 255
9          0x015ec    117  0.118%  sw x3 , 28 ( sp )
10         0x015f0    117  0.118%  and x3 , x17 , 255
11         0x015f4    117  0.118%  sw x3 , 164 ( sp )
12         0x015f8    117  0.118%  and x31 , x6 , 255
13         0x015fc    117  0.118%  and x3 , x23 , 255
14         0x01600    117  0.118%  sw x3 , 168 ( sp )
15         0x01604    117  0.118%  and x9 , x9 , 255
16         0x01608    117  0.118%  sw x9 , 36 ( sp )
17         ...
18         */
```

Zdrojový kód 5.1: Ukázka anotovaného zdrojového kódu

Profiler k této funkci přiřadil strojový kód spolu s počtem cyklů procesoru strávených na každé instrukci. Podstatná informace pro analýzu je první řádek anotace, který je sumou cyklů strávených na tento řádek kódu, včetně procentuálního vytížení. Tato funkce je tedy ideálním kandidátem na optimalizaci, neboť se jedná pouze o logické operace s pouze jedním přístupem do paměti. Konkrétní instrukce, která tuto funkci implementuje, je popsána v další sekci.

Místa podobná této funkci byla pomocí profileru zkoumána i v dalších algoritmech. Vhodná místa se typicky nachází v rundových funkcích, neboť to je část algoritmu, která se neustále opakuje při šifrování i dešifrování. Výjimku tvoří např. algoritmus Twofish, kde kvůli svému návrhu způsobuje značné zatížení procesoru inicializace, proto se vyplatilo tuto část optimalizovat.

Dalším typickým místem k optimalizaci jsou manipulace s vnitřním stavem těchto algoritmů. Tento přístup se dá uplatnit především u algoritmů, které manipulují s jednotlivými byty, protože místo čtyř 8bitových přístupů lze použít jeden 32bitový, a to jak při čtení, tak i při zápisu.

5.2.5 Aplikování instrukcí

Po přidání nových instrukcí je nutné zjistit, zda je překladač aplikoval na zamýšlené místo. Obecně se toto dá předpokládat u jednodušších instrukcí, jako jsou například rotace, nebo vícenásobné logické operace. U složitějších instrukcí je nutné explicitně deklarovat jejich použití pomocí vkládaného assembleru. Ten používá poměrně jednoduchou syntaxi, je však nutné deklarovat, které registry jsou vstupní, které výstupní a nebo zda je měněna paměť. Níže je uveden příklad použití instrukce ADD pomocí vkládaného assembleru v jazyce C (zdrojový kód 5.2).

```
1 int x,y,res;
2 x = 15;
3 y = 8;
4 __asm__("add %[RES],%[X],%[Y]"
5          :[RES] "=r" (res)
6          :[X] "r" (x), [Y] "r" (y));
7 //res = 23
```

Zdrojový kód 5.2: Ukázka použití vkládaného assembleru

Nejdříve je zapsána instrukce a její operandy jako řetězec, na dalším řádku jsou uvedeny výstupy, zde proměnná `res` s aliasem `RES` jako cílový registr. Na dalším řádku jsou stejným způsobem uvedeny proměnné `x` a `y` jako vstupní operandy.

5.3 Nové instrukce

V této podkapitole se nachází popis jednotlivých instrukcí členěných podle algoritmu, pro který jsou určené. Pokud není uvedeno jinak, byl pro jejich použití v algoritmech využit vkládaný assembler.

5.3.1 3DES

DES_IP

Instrukce DES_IP implementuje počáteční permutaci algoritmu 3DES. Jedná se převážně o logické operace, jako jsou bitové posuny, operace XOR nebo AND. Instrukce má dva vstupní operandy, spodních 32 bitů vnitřního stavu `src1` a horních 32 bitů vnitřního stavu `src2`. Poté, co jsou provedeny potřebné operace, jsou oba operandy zapsány zpět do registrů, ze kterých byly načteny. Instrukce používá 17bitový operační kód. Zdrojový kód instrukce se nachází v příloze A.6.

DES_FP

Tato instrukce je velmi podobná předchozí instrukci DES_IP, protože se jedná o inverzní operaci, tedy o závěrečnou permutaci v algoritmu 3DES. Vstupní operandy jsou stejné, dohromady tvoří vnitřní stav algoritmu. Stejně jako v příbuzné instrukci se i tady nakonec výsledky zapíší zpět do vstupních registrů. Instrukce používá 17bitový operační kód a její zdrojový kód se nachází v příloze A.7.

DES_R

Tato instrukce implementuje rundovou funkci algoritmu 3DES. Jedná se ve skutečnosti o dvě instrukce (resp. dva operační kódy), které jsou sloučeny v jednu instrukci. Toto je realizováno pomocí zadefinování dvou operačních kódů se stejnou skupinou, popis instrukce v CodALu je pak přiřazen k této skupině. Důvodem použití tohoto přístupu bylo téměř identické chování v obou částech rundové funkce. Druhá část se od první liší pouze rotací vnitřního stavu a substitucí jinými S-boxy. Pro efektivní využití této instrukce bylo nutno změnit implementaci algoritmu, konkrétně sloučit jednotlivá pole S-boxů do jednoho dvourozměrného pole. Tato úprava umožňuje zachovat délku instrukce 32 bitů, protože ukazatel na všechny S-boxy lze předat jako jeden argument. Instrukce používá 10bitový

operační kód, kvůli nutnosti použít čtyři operandy. Prvním operandem (`dst`) je 32bitová pravá polovina vnitřního stavu. Druhým operandem (`src1`) je 32bitová polovina podklíče, který je v celku 48bitový, tedy při druhé části rundové funkce je využito jenom spodních 16 bitů. Třetím operandem (`src2`) je též 32bitová levá polovina vnitřního stavu. Posledním operandem (`src3`) je ukazatel na pole S-boxů. Výstupem této instrukce je modifikovaná pravá polovina vnitřního stavu. Zdrojový kód instrukce lze najít v příloze A.5.

DES_SK1

Tato instrukce je spolu s instrukcí DES_SK2 součástí procesu odvozování podklíčů v algoritmu DES. Instrukci tvoří pouze bitové posuny a operace AND a OR. Prvním operandem (`dst`) je ukazatel do paměti na položku v poli podklíčů, kam se výsledek operace ukládá. Další operandy (`src1` a `src2`) jsou už permutované části podklíče po permutaci PC-1. Instrukce používá 17bitový operační kód.

```
1  element i_des_sk1
2  {
3    use opc_des_sk1 as opc;
4    use reg_any as dst,src1,src2;
5    assembler { opc dst "," src1 "," src2 };
6    binary { opc[OPC_FRAG2] src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0] };
7    semantics
8    {
9      uXlen X,Y, Z;
10     X = rf_gpr_read(src1);
11     Y = rf_gpr_read(src2);
12     Z = ((X << 4) & 0x24000000) | ((X << 28) & 0x10000000)
13     | ((X << 14) & 0x08000000) | ((X << 18) & 0x02080000)
14     | ((X << 6) & 0x01000000) | ((X << 9) & 0x00200000)
15     | ((X >> 1) & 0x00100000) | ((X << 10) & 0x00040000)
16     | ((X << 2) & 0x00020000) | ((X >> 10) & 0x00010000)
17     | ((Y >> 13) & 0x00002000) | ((Y >> 4) & 0x00001000)
18     | ((Y << 6) & 0x00000800) | ((Y >> 1) & 0x00000400)
19     | ((Y >> 14) & 0x00000200) | ((Y ) & 0x00000100)
20     | ((Y >> 5) & 0x00000020) | ((Y >> 10) & 0x00000010)
21     | ((Y >> 3) & 0x00000008) | ((Y >> 18) & 0x00000004)
22     | ((Y >> 26) & 0x00000002) | ((Y >> 24) & 0x00000001);
23     rf_gpr_write(dst, Z);
24   };
25 };
```

Zdrojový kód 5.3: Instrukce DES_SK1

DES_SK2

Tato instrukce je strukturou i účelem shodná s předchozí instrukcí DES_SK1. Jediným rozdílem jsou jiné logické operace, které jsou též součástí procesu tvorby podklíčů v algoritmu DES. Instrukce je pro úplnost uvedena v příloze A.8.

5.3.2 AES

AES_MUL

Instrukce AES_MUL implementuje násobení v konečném tělese $GF(2^8)$ v algoritmu AES. Tato konkrétní instrukce se používá ve fázi kombinování sloupců při dešifrování. Prvním operandem (`dst`) je registr, kam se má výsledek násobení uložit. Druhým operandem (`src1`) je první z čísel k násobení a poslední operand (`src2`) je druhé číslo k násobení, v tomto případě je kvůli povaze algoritmu použit přímý 5bitový operand, protože nejvyšší číslo, kterým se násobí je 14. Instrukce používá 17bitový operační kód.

```
1  element i_aes_mul
2  {
3    use opc_aes_mul as opc;
4    use reg_any as src1, src2, dst;
5    assembler { opc dst "," src1 "," src2};
6    binary { opc[OPC_FRAG2] src1 src2 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
7    semantics
8    {
9      uint8 x, y, tmp0, tmp1, tmp2, tmp3;
10     x = rf_gpr_read(src1);
11     y = rf_gpr_read(src2);
12     tmp0 = ((y & 1) * x);
13     x = (((uint8)x<<1) ^ (((x>>7) & 1) * 0x1b));
14     tmp1 = ((y>>1 & 1) * x);
15     x = (((uint8)x<<1) ^ (((x>>7) & 1) * 0x1b));
16     tmp2 = ((y>>2 & 1) * x);
17     x = (((uint8)x<<1) ^ (((x>>7) & 1) * 0x1b));
18     tmp3 = ((y>>3 & 1) * x);
19     rf_gpr_write(dst, tmp0 ^ tmp1 ^ tmp2 ^ tmp3);
20   };
21 };
```

Zdrojový kód 5.4: Instrukce AES_MUL

AES_SB

Instrukce `AES_SB` implementuje jeden krok bytové substituce, která je součástí rundové funkce algoritmu AES. Prvním operandem (`dst`) je ukazatel pole, které reprezentuje vnitřní stav algoritmu. Druhý operand (`src1`) je ukazatel na pole, které obsahuje S-box. Poslední operand (`src2`) je 5bitový přímý operand indikující číslo řádku, pro který se substituce provádí. Jeden řádek má velikost 32 bitů, je tedy možné ho načíst celý najednou. Následně probíhá načtení 8bitových hodnot z S-boxu podle jednotlivých bytů řádku vnitřního stavu a nakonec jsou tyto hodnoty opět spojeny a jedním přístupem do paměti zapsány do vnitřního stavu. Pro úplné pokrytí fáze bytové substituce je nutné tuto instrukci použít čtyřikrát, s hodnotami třetího operandu 0-3. Instrukce používá 17bitový operační kód.

```
element i_aes_sb
{
  use opc_aes_sb as opc;
  use reg_any as src1, dst;
  use imm5 as src2;
  assembler { opc dst "," src1 "," src2};
  binary { opc[OPC_FRAG2] src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
  semantics
  {
    uXlen state, sbox, state_val;
    uint8 i, j,tmp0,tmp1,tmp2,tmp3;
    uint5 row;
    row = src2;
    state = rf_gpr_read(dst);
    sbox = rf_gpr_read(src1);
    state_val = load_val(OPC_I_LW, state + row*4);
    tmp0 = load_val(OPC_I_LBU, sbox + state_val[7..0]);
    tmp1 = load_val(OPC_I_LBU, sbox + state_val[15..8]);
    tmp2 = load_val(OPC_I_LBU, sbox + state_val[23..16]);
    tmp3 = load_val(OPC_I_LBU, sbox + state_val[31..24]);
    write_val(OPC_S_SW, state + row*4, tmp3 :: tmp2 :: tmp1 :: tmp0);
  };
};
```

Zdrojový kód 5.5: Instrukce AES_SB

AES_SR

Tato instrukce implementuje jeden krok fáze posunutí, která je součástí rundové funkce algoritmu AES. Kvůli této instrukci a její inverzní variantě bylo nutné pozměnit reprezentaci vnitřního stavu v implementaci algoritmu tak, aby byl v paměti uložen po řádcích, a ne po sloupcích. To umožňuje efektivní načtení celého řádku stavu na jeden přístup do paměti, jeho cyklické posunutí a zapsání zpět do paměti. Prvním operandem instrukce je `dst`, který obsahuje ukazatel na pole, kterým je v algoritmu reprezentován vnitřní stav. Druhý operand (`src1`) je 5bitový přímý operand, který značí řádek, pro který se má posunutí provést. Na základě tohoto operandu se také určuje míra posunutí. Instrukce používá 17bitový operační kód.

```
element i_aes_sr
{
use opc_aes_sr as opc;
use reg_any as dst;
use imm5 as src1;
assembler { opc dst "," src1};
binary { opc[OPC_FRAG2] UNUSED(5) src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
semantics
{
  uXlen state,rdata;
  uint5 row;
  state = rf_gpr_read(dst);
  row = src1;
  rdata = load_val(OPC_I_LW, state + row*4);
  switch(row){
    case 1:
    rdata = rdata[7..0] :: rdata[31..24] :: rdata[23..16] :: rdata[15..8];
    break;
    case 2:
    rdata = rdata[15..8] :: rdata[7..0] :: rdata[31..24] :: rdata[23..16];
    break;
    case 3:
    rdata = rdata[23..16] :: rdata[15..8] :: rdata[7..0] :: rdata[31..24];
    break;
  }
  write_val(OPC_S_SW, state + row*4, rdata);
  };
};
```

Zdrojový kód 5.6: Instrukce AES_SR

AES_ISR

Tato instrukce je inverzní k instrukci AES_SR, implementuje tedy fázi posunutí při dešifrování v algoritmu AES. Od předchozí instrukce se liší pouze v tom, o kolik se jednotlivé řádky stavu posunují. Instrukce je pro úplnost uvedena v příloze A.9.

AES_XTM

Instrukce `AES_XTM` implementuje funkci `xtime()` z implementace algoritmu AES. Jedná se o pomocnou funkci k násobení v konečném tělese $GF(2^8)$, která je součástí fáze kombinování sloupců rundové funkce. Zajišťuje modulární redukci `x` tak, aby výsledek násobení náležel do konečného tělesa $GF(2^8)$. Instrukce používá pouze dva operandy, cílový a zdrojový. Operační kód instrukce je 17bitový.

```
element i_aes_xtm
{
  use opc_aes_xtm as opc;
  use reg_any as dst, src;
  assembler { opc dst "," src};
  binary {opc[OPC_FRAG2] UNUSED(5) src opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
  semantics
  {
    uint8 x;
    x = rf_gpr_read(src);
    rf_gpr_write(dst, ((uint8) x<<1) ^ ((((uint8) x>>7) & 1) * 0x1b));
  };
};
```

Zdrojový kód 5.7: Instrukce AES_XTM

AES_ADD

Tato instrukce implementuje přidávání podklíčů v rundové funkci algoritmu AES. Skládá se z načtení příslušného řádku vnitřního stavu, načtení odpovídajících částí podklíče pro danou rundu, provedení operace XOR nad těmito čísly a ze zapsání zpět do paměti. Operandy jsou celkem 4, první z nich (`dst`) je ukazatel na pole reprezentující vnitřní stav, druhý operand (`src1`) je ukazatelem na pole obsahující podklíče, třetí operand (`src2`) označuje číslo probíhající rundy a poslední operand (`src3`) určuje řádek vnitřního stavu, pro který se instrukce provádí. Instrukce tak v důsledku použití většího počtu operandů používá 10bitový operační kód. Zdrojový kód instrukce je uveden v příloze A.10.

5.3.3 Blowfish

BF_F

Tato instrukce je navržena pro algoritmus Blowfish. Vychází z implementace jeho rundové funkce, která spočívá v rozdělení 32bitové půlky vnitřního stavu po bytech, nahrazení jednotlivých hodnot jejich protějšky z S-boxů a následného zkombinování pomocí sčítání a operace XOR. Funkce je snadno implementovatelná, neboť vstupními operandy jsou pouze 32bitová hodnota poloviny vnitřního stavu a ukazatel na pole S-boxů. Zrychlení je umožněno možností načíst celý stav najednou, místo po jednotlivých bytech, a stejně tak ho i uložit. U instrukce je použit 17bitový operační kód, instrukce má R formát.

```
element i_bf_f
{
  use opc_bf_f as opc;
  use reg_any as src1, src2, src3;
  assembler { opc src1 "," src2 "," src3};
  binary {opc[OPC_FRAG2] src1 src2 opc[OPC_FRAG1] src3 opc[OPC_FRAG0]};
  semantics
  {
    uXlen x, t, s;
    x = rf_gpr_read(src1);
    t = rf_gpr_read(src2);
    s = rf_gpr_read(src3);
    t = load_val(OPC_I_LW, s + 0 + (x >> 24)*4);
    t += load_val(OPC_I_LW, s + 256*4*1 + ((x >> 16)& 0xff)*4);
    t ^= load_val(OPC_I_LW, s + 256*4*2 + ((x >> 8)& 0xff)*4);
    t += load_val(OPC_I_LW, s + 256*4*3 + (x & 0xff)*4);
    rf_gpr_write(src2, t);
  };
};
```

Zdrojový kód 5.8: Instrukce BF_H

5.3.4 Twofish

TF_H

Tato instrukce implementuje funkci `h()` algoritmu Twofish. Jedná se o jednu z nejkomplikovanějších instrukcí v celé sadě. Používá pouze 10bitový operační kód a má pět operandů. Tak velký počet operandů je umožněn tím, že jeden z operandů je pouze 2bitový. Prvním z operandů je `dst`, který je použit pro uložení výsledku a zároveň slouží k předání vstupní hodnoty funkce. Druhý operand (`src1`) je ukazatel na pole reprezentující matici MDS, která slouží k získání výsledků násobení touto maticí jejich substitucí z pole. Třetí operand (`src2`) slouží k předání ukazatele na pole, kde jsou uloženy jednotlivé byty klíče. Čtvrtý operand (`src3`) je ukazatel na pole, ve kterém jsou uloženy oba permutační boxy q_0 a q_1, každý o 256 položkách. Posledním operandem (`k`) je přímý 2bitový operand, reprezentující proměnnou `k`, která v algoritmu reprezentuje použitou délku klíče a může nabývat hodnot $k \in \{2, 3, 4\}$. Proměnná se kvůli velikosti operandu předává jako $k - 2$. Zdrojový kód instrukce je uveden v příloze A.2.

TF_FS

Instrukce TF_FS je obdobou předchozí instrukce TF_H a používá se k vypočítání hodnot, kterými se má naplnit na klíči závislý S-box v algoritmu Twofish. Má stejné operandy, které mají i stejný význam v instrukci, až na operand `dst`, který tentokrát nepředává proměnnou, ale ukazatel do 8bitového pole, kam se ukládají výsledné byty. Ty jsou vždy čtyři a jsou poté použity jako indexy do MDS pole. Substituované hodnoty jsou poté uloženy do S-boxu. Zdrojový kód instrukce lze najít v příloze A.1.

TF_G

Tato instrukce implementuje obě varianty funkce $g()$, které jsou součástí algoritmu Twofish. Tyto varianty se liší pouze v rotaci vstupního slova o 8 bitů doleva. Prvním operandem instrukce je `dst`, který určuje registr pro uložení výsledku. Další operand (`src1`) je ukazatelem na pole reprezentující S-box. Poslední operand (`src2`) je vstupní slovo. Po nahrazení hodnot z S-boxu se výsledná čísla XORují a jsou uložena do cílového registru. Instrukce používá 17bitový operační kód. Zdrojový kód instrukce je uveden v příloze A.3.

TF_RS

Instrukce TF_RS je určena k usnadnění výpočtu podklíčů, konkrétně části, kde je vypočítán vektor S v algoritmu Twofish. Zvláštností tohoto výpočtu je, že výsledný vektor musí být v opačném pořadí a pole jsou tedy procházena pozpátku. V důsledku toho bylo nutné načítat jednotlivá čísla po bytech, místo po celých slovech. Instrukce má pouze jeden operand (`src`), kterým je ukazatel do pole koeficientů polynomu, nad kterými se výpočet provádí. Instrukce používá 17bitový operační kód a její zdrojový kód je uveden v příloze A.4.

TF_ROR4BY1

Tato instrukce je implementací stejnojmenného pomocného makra v algoritmu Twofish. Účel tohoto makra je rotovat spodní 4 bity zadaného čísla. Prvním operandem (`src`) je tedy vstup tohoto makra a druhým operandem (`dst`) je registr, kam se má výsledek uložit. Instrukce používá 17bitový operační kód.

```
1  element i_tf_ror4by1
2  {
3    use opc_tf_ror4by1 as opc;
4    use reg_any as src, dst;
5    assembler { opc src "," dst};
6    binary { opc[OPC_FRAG2] UNUSED(5) src opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
7    semantics
8    {
9      uXlen x;
10     x = rf_gpr_read(src);
11     x = (((x)>>1) | (((x)<<3) & 0x8) ) ;
12     rf_gpr_write(dst, x);
13   };
14 };
```

Zdrojový kód 5.9: Instrukce TF_ROR4BY1

5.3.5 RSA

RSA_MC

Instrukce RSA_MC implementuje pomocné makro při násobení velkých čísel, které je nutné pro výpočet dostatečně bezpečných klíčů v algoritmu RSA. Tato instrukce má celkem čtyři operandy, tedy používá pouze 10bitový operační kód. Prvním operandem je `dst1` a reprezentuje registr, který ukládá tzv. *carry*, neboli přenos, který může při násobení vzniknout. Operand `dst2` značí registr, kam má být výsledek násobení uložen. Zbylé dva operandy

(`src1` a `src2`) jsou čísla, která mají být vynásobena. Jedná se o násobení dvou 32bitových čísel. Zdrojový kód instrukce lze nalézt v příloze A.11.

Lze si povšimnout dvou funkcí v popisu instrukce – `codasip_compiler_builtin()` a `codasip_compiler_undefined()`. Tyto funkce bylo nutné použít, protože generovaný překladač nebyl schopný instrukci přijmout. Tyto funkce vynucují zahrnutí instrukce jako built-in funkce překladače.

RSA_SHR

Tato instrukce implementuje část funkce používané pro bitový posun velkých čísel z implementace algoritmu RSA. První operand (`dst`) je 32bitová část čísla, která má být posunuta. Druhý operand (`src1`) je míra posunutí a poslední operand (`src2`) je proměnná, která se používá v průběhu výpočtu a kam se pak ukládá mezivýsledek. Instrukce používá 17bitový operační kód. Stejně jako předchozí instrukce zahrnuje i tato funkce, které vynucují použití instrukce jako built-in funkce překladače.

```
1  element i_rsa_shr
2  {
3    use opc_rsa_shr as opc;
4    use reg_any as src1, src2, dst;
5    assembler { opc dst "," src1 "," src2};
6    binary { opc[OPC_FRAG2] src1 src2 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
7    semantics
8    {
9      uXlen Xp, v1, r0, r1;
10
11     codasip_compiler_builtin();
12     codasip_compiler_undefined();
13
14     Xp = rf_gpr_read(dst);
15     v1 = rf_gpr_read(src1);
16     r0 = rf_gpr_read(src2);
17     r1 = Xp << (32-v1);
18     Xp = (Xp >> v1) | r0;
19     rf_gpr_write(dst, Xp);
20     rf_gpr_write(src2, r1);
21   };
22 };
```

Zdrojový kód 5.10: Instrukce RSA_SHR

5.3.6 Obecné instrukce

ROTL32

ROTL32 je první ze čtyř instrukcí, které byly navrženy pro obecné užití ve všech algoritmech, protože se jedná o v kryptografii běžně používané operace. Jedná se levou bitovou rotaci 32bitového čísla. Instrukce má tři operandy, rotované číslo (`src1`), míru rotace (`src2`) a cílový registr (`dst`). Instrukce tedy používá 17bitový operand. Zajímavostí této instrukce je, že je přenositelná i na 64bitovou nebo 128bitovou architekturu, protože používá makro

XLEN, které obsahuje počet bitů použité architektury, pro tento model je to tedy 32. Tato instrukce byla použita překladačem bez nutnosti využití vkládaného assembleru.

```
element i_rotl32
{
  use opc_rotl32 as opc;
  use reg_any as src1, src2, dst;
  assembler { opc dst "," src1 "," src2};
  binary { opc[OPC_FRAG2] src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
  semantics
  {
    uXlen x,y;
    x = rf_gpr_read(src1);
    y = rf_gpr_read(src2);
    rf_gpr_write(dst, x << y | x >> (XLEN - y));
  };
};
```

Zdrojový kód 5.11: Instrukce ROTL32

ROTR32

Instrukce ROTR32 je obdobou předchozí instrukce ROTL32, jedná se také o rotaci, tentokrát doprava. Používá také 17bitový operační kód a tři operandy se stejným názvem i významem, jako u předchozí instrukce. Ani pro tuto instrukci nebylo nutné vynucovat použití v algoritmu pomocí vkládaného assembleru.

```
element i_rotr32
{
  use opc_rotr32 as opc;
  use reg_any as src1, src2, dst;
  assembler { opc dst "," src1 "," src2};
  binary { opc[OPC_FRAG2] src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
  semantics
  {
    uXlen x,y;
    x = rf_gpr_read(src1);
    y = rf_gpr_read(src2);
    rf_gpr_write(dst, x >> y | x << (XLEN - y));
  };
};
```

Zdrojový kód 5.12: Instrukce ROTR32

XOR3

XOR3 je další z obecně využitelných instrukcí. Instrukční sada RV32IM obsahuje instrukci XOR, která provádí operaci XOR nad dvěma operandy a výsledek ukládá do třetího operandu. XOR3 je rozšířením původní instrukce o další operand. Instrukce byla navržena kvůli skutečnosti, že v kryptografii je operace XOR např. nad vnitřním stavem hojně užívána. Prvním operandem je dst, který je zároveň zdrojem prvního čísla a cílovým reg-

istrem. Zbylá dvě čísla se nachází v operandech `src1` a `src2`. Instrukce používá 17bitový operand. XOR3 je jedna z několika mála instrukcí, kterou překladač použil sám bez nutnosti použití vkládaného assembleru.

```
element i_xor3
{
  use opc_xor3 as opc;
  use reg_any as dst,src1,src2;
  assembler { opc dst "," src1 "," src2 };
  binary { opc[OPC_FRAG2] src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0] };
  semantics
  {
    uXlen X,Y,Z;
    X = rf_gpr_read(dst) ^ rf_gpr_read(src1);
    rf_gpr_write(dst, X ^ rf_gpr_read(src2));
  };
};
```

Zdrojový kód 5.13: Instrukce XOR3

XOR4

XOR4 je obdobou předchozí instrukce XOR3, pouze rozšířenou o další operand. Tato instrukce tedy provádí operaci XOR nad čtyřmi operandy a do jednoho z nich (`dst`) výsledek uloží. Instrukce kvůli velkému počtu operandů používá pouze 10bitový operační kód. I tato instrukce byla překladačem použita automaticky.

```
element i_xor4
{
  use opc_xor4 as opc;
  use reg_any as dst,src1,src2,src3;
  assembler { opc dst "," src1 "," src2 "," src3 };
  binary { UNUSED(2) src3 src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0] };
  semantics
  {
    uXlen X,Y,Z;
    X = rf_gpr_read(dst) ^ rf_gpr_read(src1);
    X ^= rf_gpr_read(src2);
    rf_gpr_write(dst, X ^ rf_gpr_read(src3));
  };
};
```

Zdrojový kód 5.14: Instrukce XOR4

GET32

Instrukce GET32 implementuje pomocnou operaci, která slouží pro manipulaci s vnitřním stavem v algoritmech 3DES a Blowfish. Pomocí této instrukce lze efektivně načíst čtyři 8bitové hodnoty z pole a zapsat je do registru. Vstupní operandy jsou celkem tři, jedná se o cílový registr (`dst`), kterým je jedna z polovin vnitřního stavu, dále pak ukazatel na 8bitové pole obsahující klíč nebo otevřený text (`src1`) a přímý operand značící odsazení

od začátku pole v bytech (`imm`). Pro tento operand byla zvolena šířka 5 bitů, protože se v algoritmu nevyskytují vyšší odsazení než o 4. Pořadí bytů je invertováno, protože implementace vyžaduje načtení jako big-endian. Instrukce používá 17bitový operační kód.

```
1   element i_get32
2   {
3   use opc_get32 as opc;
4   use reg_any as dst,src1;
5   use imm5 as imm;
6   assembler { opc dst "," src1 "," imm };
7   binary { opc[OPC_FRAG2] imm src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0] };
8   semantics
9   {
10  uXlen base, n;
11  base = rf_gpr_read(src1) + imm;
12  n = load_val(OPC_I_LW, base);
13  rf_gpr_write(dst, n[7..0] :: n[15..8] :: n[23..16] :: n[31..24]);
14  };
15  };
```

Zdrojový kód 5.15: Instrukce GET32

PUT32

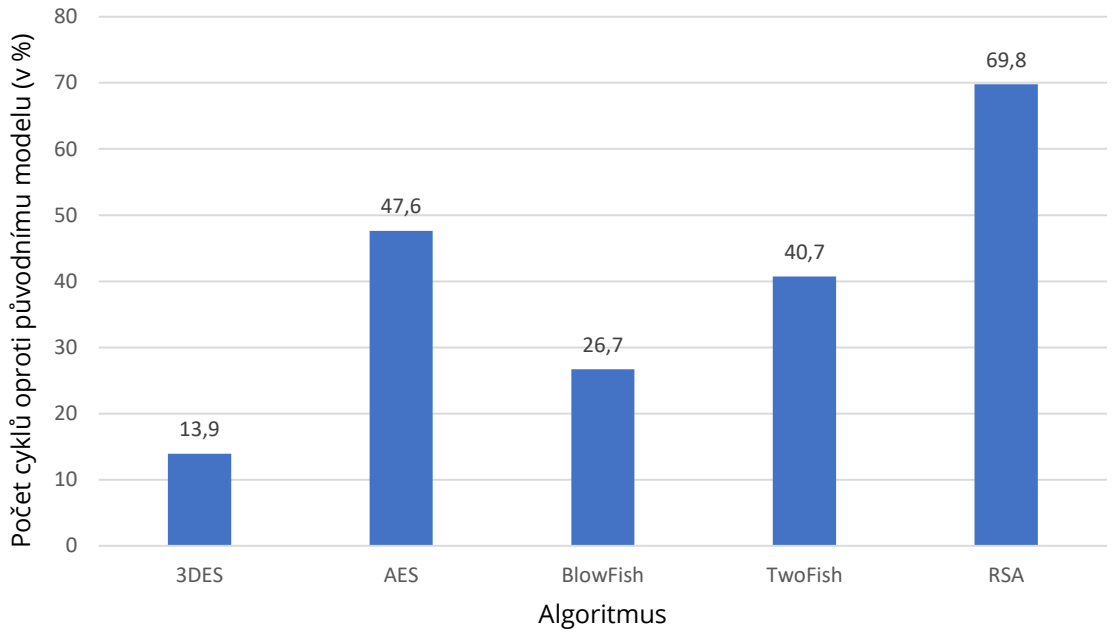
Tato instrukce je obdobou GET32 pro zápis 32bitové hodnoty do paměti v algoritmech 3DES a Blowfish. Instrukce se používá k zápisu do osmibitového pole, kde místo čtyř 8bitových hodnot je zapsáno jednou 32 bitů. Vstupní operandy jsou stejné jako u zmíněné instrukce GET32, `dst` je ukazatel do 8bitové pole, `src1` je 32bitová hodnota k zápisu a `imm` odsazení v poli v bytech. Stejně i zde je hodnota před zápisem invertována tak, aby byla převedena z big-endian zpět na little-endian. I tato instrukce používá 17bitový operační kód.

```
1   element i_put32
2   {
3   use opc_put32 as opc;
4   use reg_any as dst,src1;
5   use imm5 as imm;
6   assembler { opc dst "," src1 "," imm };
7   binary { opc[OPC_FRAG2] imm src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0] };
8   semantics
9   {
10  uXlen base, n;
11  n = rf_gpr_read(src1);
12  write_val(OPC_S_SW, rf_gpr_read(dst) + imm, n[7..0] :: n[15..8] :: n
        [23..16] :: n[31..24]);
13  };
14  };
```

Zdrojový kód 5.16: Instrukce PUT32

5.4 Výsledky

Nových instrukcí je celkem 27. Všechny algoritmy se povedlo do určité míry urychlit, jak ilustruje graf 5.1. Nejvíce urychleným algoritmem je 3DES, kde lze pozorovat snížení počtu cyklů procesoru na 13, 9% původní hodnoty. Naopak nejméně urychleným algoritmem je pak RSA, kde zrychlení činí pouze 30%. Celkově se tedy všechny algoritmy povedlo urychlit alespoň o třetinu. Průměrně jsou algoritmy urychleny o 60%. Efekty přidání nových instrukcí a jejich vliv na počet cyklů u konkrétních algoritmů jsou popsány v následujících podkapitolách.



Obrázek 5.1: Graf dosaženého urychlení u jednotlivých algoritmů

5.4.1 3DES

3DES je nejvíce urychleným algoritmem z celé sady. Bylo pro něj také vytvořeno nejvíce instrukcí. Téměř všechny měly poměrně malý vliv v řádu jednotek procent, až na dvojici instrukcí DES_R1 a DES_R2, které umožnily masivní urychlení celého algoritmu, protože dohromady optimalizují celou rundovou funkci. Toto zrychlení bylo umožněno především vhodnou velikostí bloku, která je 64bitů, ale operuje se vždy s jeho polovinou. Dalším důvodem je také celkově jednoduchá struktura algoritmu, kterou je Feistelova síť.

Instrukce	Počet cyklů	Zrychlení
-	118362222	-
DES_FP + DES_IP	112562222	1,051527057
XOR3 + XOR4	106962222	1,106579686
DES_SK1 + DES_SK2	106936086	1,106850142
GET32 + PUT32	105135870	1,125802469
ROTL32 + ROTR32	100875870	1,173345241
DES_R1 + DES_R2	16495870	**7,17526399**

Tabulka 5.1: Shrnutí výsledků pro 3DES

5.4.2 AES

U algoritmu AES se podařilo dosáhnout více než dvojnásobného urychlení. Instrukce pokrývají jednotlivé části rundové funkce algoritmu. Největší vliv zde měla instrukce AES_XTM, která implementuje část procesu násobení v konečném tělese $GF(2^8)$. Další významnou instrukcí je zde AES_ADD, která optimalizuje přidávání podklíčů k vnitřnímu stavu pomocí operace XOR. Nabízí se zde porovnání s již existujícím instrukčním rozšířením AES-NI pro architektury x86, ARM apod. Porovnání ztěžuje fakt, že např. Intel ve svých materiálech používá paralelní implementaci[1] s vícejádrovými procesory. Vliv jednotlivých instrukcí je popsán v tabulce 5.2.

Instrukce	Počet cyklů	Zrychlení
-	99430	-
AES_MUL	92319	1,077026398
AES_SB	87543	1,1357847
AES_SR + AES_ISR	84164	1,181383965
AES_XTM	59126	1,681662889
AES_ADD	47343	**2,100204888**

Tabulka 5.2: Shrnutí výsledků pro AES

5.4.3 Blowfish

Blowfish je druhým nejvíce urychleným algoritmem (přibližně $3,7$krát, jak lze pozorovat v tabulce 5.3). Zrychlení zde bylo umožněno jeho jednoduchou strukturou podobně jako u 3DES. Nejzatíženějším místem algoritmu je funkce $f()$, kterou optimalizuje instrukce BF_F. Dále zde našla užití dvojice instrukcí PUT32 a GET32, které se používají na efektivní manipulaci s vnitřním stavem.

[1] https://software.intel.com/sites/default/files/m/d/4/1/d/8/10TB24_Breakthrough_AES_Performance_with_Intel_AES_New_Instructions.final.secure.pdf

Instrukce	Počet cyklů	Zrychlení
-	695503	-
XOR3 + XOR4 + BF_F	262459	2,649949135
PUT32 + GET32	185683	**3,745647151**

Tabulka 5.3: Shrnutí výsledků pro Blowfish

5.4.4 Twofish

U algoritmu Twofish byla kritickým místem funkce $h()$, která se v implementaci objevuje v rundové funkci a při výpočtu S-boxů, které jsou závislé na klíči. Oba tyto výskyty implementují dohromady instrukce TF_H a TF_FS, které tvoří jádro urychlení algoritmu. Další zrychlení poskytují instrukce TF_G, která implementuje funkci $g()$, která je součástí rundové funkce, a TF_RS, která implementuje část výpočtu podklíčů. Ostatní instrukce mají pouze minimální vliv v jednotkách procent. Přesné hodnoty zrychlení ukazuje tabulka 5.4.

Instrukce	Počet cyklů	Zrychlení
-	5192156	-
XOR3 + XOR4	5169304	1,004420711
TF_ROR4BY1	5166232	1,005017971
ROTL32 + ROTR32	5142835	1,009590236
TF_H	4448502	1,167169533
TF_FS	2363293	2,197000541
TF_G	2219767	2,339054504
TF_RS	2110913	**2,448847489**

Tabulka 5.4: Shrnutí výsledků pro Twofish

5.4.5 RSA

RSA je nejméně urychleným algoritmem ze všech vybraných. Tabulka 5.5 ukazuje, že se povedlo dosáhnout zrychlení $1,4$krát. Nejúspěšnější instrukcí je zde RSA_MC, která implementuje násobení s libovolnou přesností. Zmíněná operace je jádrem celého algoritmu, proto se tedy vyplatilo ji optimalizovat. Přináší zrychlení o přibližně 28%. Další instrukcí je RSA_SHR, která implementuje část funkce, která bitově posunuje čísla s libovolnou přesností. Přináší podstatně menší zrychlení (2%), nicméně se stále vyplatí.

Hlavním důvodem, proč se tento algoritmus nepodařilo více optimalizovat, je skutečnost, že ke svému efektivnímu fungování potřebuje operovat s velkými čísly, který jsou v případě 32bitového modelu, který byl použit pro tuto práci, uloženy ve strukturách po 32 bitech. Optimalizace tohoto algoritmu by lépe probíhala na 64bitové verzi modelu, případně dokonce na 128bitové verzi.

Instrukce	Počet cyklů	Zrychlení
-	24758483	-
RSA_MC	17789380	1,391756374
RSA_SHR	17284188	**1,432435414**

Tabulka 5.5: Shrnutí výsledků pro RSA

5.5 Další možná rozšíření

Ačkoliv výsledky této práce jsou u většiny algoritmů poměrně uspokojivé, lze předpokládat, že by se dalo docílit ještě větších optimalizací změnou některých skutečností. Například u algoritmu RSA se nabízí použití vícebitové architektury, tedy např. 64bitové nebo 128bitové. Obecně užitečné by mohlo být instrukční rozšíření pokrývající knihovnu pro výpočty s libovolnou přesností, kterou algoritmus RSA využívá, které by ale vyžadovalo alespoň 64bitové registry. Tato architektura by naopak neměla příliš smysl pro jednodušší algoritmy, jako jsou 3DES nebo Blowfish, pro které postačuje 32bitová architektura.

Další možnou změnou by mohlo být vyzkoušení jiných implementací vybraných algoritmů.

Kapitola 6

Závěr

Tato práce vychází z mé předchozí činnosti v předmětu projektová praxe, kde jsem řešil také instrukční rozšíření v kryptografii, a jejíž výsledkem byl optimalizovaný algoritmus SHA-3. Tato práce je rozšířením cílů z projektové praxe na populární a v IoT zařízeních užitečné kryptografické algoritmy.

V rámci této práce jsem nejdříve nastudoval obecné principy kryptografie a na základě těchto znalostí jsem vybral několik algoritmů k optimalizaci a u nich nastudoval podrobný princip jejich fungování. Dále jsem se seznámil s prostředím Codasip Studia a principy jazyka CodAL a také s architekturou instrukční sady RISC-V. Následně jsem implementoval testy těchto algoritmů, které následně umožnily rychlé prototypování nových instrukcí. Na základě znalostí o vybraných kryptografických algoritmech jsem navrhl a implementoval celkem 27 nových instrukcí, které poskytují průměrné urychlení algoritmů o 60% oproti původnímu profilování na neupraveném modelu procesoru. Součástí této práce jsou i upravené implementace kryptografických algoritmů, protože u složitějších instrukcí nebyl překladač schopen sám instrukce použít a bylo nutno si jejich využití vynutit pomocí vkládaného assembleru.

Moje řešení v podobě sady instrukčních rozšíření může minimalizovat výkonovou penalizaci při použití silnějších kryptografických algoritmů a zároveň může přispět k snadnější adopci architektury RISC-V, ať už samotnými rozšířeními pro kryptografii, nebo demonstrací snadné a efektivní rozšiřitelnosti instrukční sady této architektury.

Dalším možným pokračováním této práce by mohlo být rozšíření sady algoritmů na další populární algoritmy v kryptografii nebo zaměření se na jednu z populárních kryptografických knihoven, jako je Libgcrypt nebo OpenSSL, a implementování části algoritmů v této knihovně.

Literatura

[1] Chang, J. K.-T. – Liu, C. – Gaudiot, J.-L. Hardware Acceleration for Cryptography Algorithms by Hotspot Detection. In Park, J. J. et al. (Ed.) *International Conference on Grid and Pervasive Computing*, s. 472–481, Berlin, Heideberg, 2013. Springer. ISBN 978-3-642-38026-6.

[2] Katz, J. – Lindell, Y. *Introduction to modern cryptography.* Boca Raton, FL, USA : CRC Press, 2 edition, 2015. ISBN 978-1-4665-7026-9.

[3] Amirki. *Feistel cipher diagram*, September 2011. Dostupné z: <https://commons.wikimedia.org/wiki/File:Feistel_cipher_diagram_en.svg>. [Online; navštíveno 22.10.2017].

[4] *DES source code* [online]. [Online; navštíveno 22.10.2017]. Dostupné z: <https://tls.mbed.org/des-source-code>.

[5] Bhargavan, K. – Leurent, G. On the Practical (In-)Security of 64-bit Block Ciphers. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, s. 456–467, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4139-4.

[6] *kokke/tiny-AES-c: Small portable AES128/192/256 in C* [online]. [Online; navštíveno 22.10.2017]. Dostupné z: <https://github.com/kokke/tiny-AES-c>.

[7] Paar, C. – Pelzl, J. *Understanding cryptography: a textbook for students and practitioners.* Heidelberg, Německo : Springer, 2010. ISBN 978-3-642-04100-6.

[8] *B-Con/crypto-algorithms: Basic implementations of standard cryptography algorithms, like AES and SHA-1.* [online]. [Online; navštíveno 22.10.2017]. Dostupné z: <https://github.com/B-Con/crypto-algorithms>.

[9] Schneier, B. Description of a new variable-length key, 64-bit block cipher (Blowfish). In Anderson, R. (Ed.) *Fast Software Encryption*, s. 191–204, Berlin, Heideberg, 1994. Springer. ISBN 978-3-540-58108-6.

[10] *wernerd/ZRTPCPP: C++ Implementation of ZRTP protocol - GNU ZRTP C++* [online]. [Online; navštíveno 22.10.2017]. Dostupné z: <https://github.com/wernerd/ZRTPCPP>.

[11] Schneier, B. et al. *Twofish: A 128-Bit Block Cipher* [online]. 1998. [Online; navštíveno 08.04.2018]. Dostupné z: <https://www.schneier.com/academic/paperfiles/paper-twofish-paper.pdf>.

[12] *RSA source code* [online]. [Online; navštíveno 22.10.2017]. Dostupné z: <https://tls.mbed.org/rsa-source-code>.

[13] Maletsky, K. RSA vs ECC Comparison for Embedded Systems. White paper, Atmel Corporation, 1600 Technology Drive, San Jose, CA 95110 USA, 2015. Dostupné z: <http://www.atmel.com/Images/Atmel-8951-CryptoAuth-RSA-ECC-Comparison-Embedded-Systems-WhitePaper.pdf>[Online; navštíveno 08.04.2018].

[14] Waterman, A. – Asanović, K. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2"*. RISC-V Foundation, 2017. Dostupné z: <https://riscv.org/specifications/>. [Online; navštíveno 08.04.2018].

[15] *Codasip Studio User Guide*. Codasip Ltd, 7.0.4 edition, 2018.

[16] *CodAL Language Reference Manual*. Codasip Ltd, 7.0.4 edition, 2018.

[17] Azad, S. – Pathan, A.-S. K. (Ed.). *Practical crypthography: algorithms and implementations using C++*. Boca Raton, FL, USA : CRC Press, 2015. ISBN 978-1-4822-2889-2.

Přílohy

Příloha A

Instrukční rozšíření

A.1 TF_FS

```
1  element i_tf_fs
2  {
3    use opc_tf_fs as opc;
4    use reg_any as dst, src1, src2, src3;
5    use imm2 as k;
6    assembler { opc dst "," src1 "," src2 "," src3 "," k};
7    binary { k src3 src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
8    semantics
9    {
10     uXlen mds, q0, q1, l, tmp0, tmp1, tmp2, tmp3, lword1, lword2,
11     y, q0_value10, q1_value10, q0_value11, q1_value11, q0_value2,
12     q1_value2,q0_value3, q1_value3, q0_val, q1_val, lword3, lword4, K;
13     mds = rf_gpr_read(src1);
14     l = rf_gpr_read(src2);
15     q0 = rf_gpr_read(src3);
16     q1 = q0 + 256;
17     y = load_val(OPC_I_LBU,rf_gpr_read(dst));
18
19     q0_val = load_val(OPC_I_LBU, q0 + y);
20     q1_val = load_val(OPC_I_LBU, q1 + y);
21
22     q0_value10 = q0_val;
23     q1_value10 = q1_val;
24     q0_value11 = q0_val;
25     q1_value11 = q1_val;
26     K = (uint32)k;
27     switch(K)
28     {
29       //kCycles = 3
30       case 1:
31       lword3 = load_val(OPC_I_LW, l + 16);
32       q0_value10 = load_val(OPC_I_LBU, q0 + (q1_val ^ lword3[7..0]));
```

```
33        q1_value10 = load_val(OPC_I_LBU, q1 + (q1_val ^ lword3[15..8]));
34        q0_value11 = load_val(OPC_I_LBU, q0 + (q0_val ^ lword3[23..16]));
35        q1_value11 = load_val(OPC_I_LBU, q1 + (q0_val ^ lword3[31..24]));
36        break;
37        //kCycles = 4
38        case 2:
39        lword3 = load_val(OPC_I_LW, l + 16);
40        lword4 = load_val(OPC_I_LW, l + 24);
41        q0_value10 = load_val(OPC_I_LBU, q1 + (q1_val ^ lword4[7..0])) ^
              lword3[7..0];
42        q1_value10 = load_val(OPC_I_LBU, q1 + (q0_val ^ lword4[15..8])) ^
              lword3[15..8];
43        q0_value11 = load_val(OPC_I_LBU, q0 + (q0_val ^ lword4[23..16])) ^
              lword3[23..16];
44        q1_value11 = load_val(OPC_I_LBU, q0 + (q1_val ^ lword4[31..24])) ^
              lword3[31..24];
45
46        q0_value10 = load_val(OPC_I_LBU, q0 + q0_value10);
47        q1_value10 = load_val(OPC_I_LBU, q1 + q1_value10);
48        q0_value11 = load_val(OPC_I_LBU, q0 + q0_value11);
49        q1_value11 = load_val(OPC_I_LBU, q1 + q1_value11);
50        break;
51      }
52
53      lword1 = load_val(OPC_I_LW, l);
54      lword2 = load_val(OPC_I_LW,l + 8);
55
56      q0_value2 = load_val(OPC_I_LBU, q0 + (q0_value10 ^ lword2[7..0]));
57      q0_value3 = load_val(OPC_I_LBU, q0 + (q1_value10 ^ lword2[15..8]));
58      q1_value2 = load_val(OPC_I_LBU, q1 + (q0_value11 ^ lword2[23..16]));
59      q1_value3 = load_val(OPC_I_LBU, q1 + (q1_value11 ^ lword2[31..24]));
60
61      tmp0 = q0_value2 ^ lword1[7..0];
62      tmp1 = q0_value3 ^ lword1[15..8];
63      tmp2 = q1_value2 ^ lword1[23..16];
64      tmp3 = q1_value3 ^ lword1[31..24];
65
66      write_val(OPC_S_SW, rf_gpr_read(dst), ((uint8) tmp3) :: ((uint8)tmp2)
            :: ((uint8)tmp1) :: ((uint8)tmp0));
67    };
68  };
```

A.2 TF_H

```
element i_tf_h
{
  use opc_tf_h as opc;
  use reg_any as dst, src1, src2, src3;
  use imm2 as k;
  assembler { opc dst "," src1 "," src2 "," src3 "," k};
  binary { k src3 src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
  semantics
  {
    uXlen mds, q0, q1, l, tmp0, tmp1, tmp2, tmp3, lword1, lword2,
    y, q0_value10, q1_value10, q0_value11, q1_value11, q0_value2,
    q1_value2,q0_value3, q1_value3, q0_val, q1_val, lword3, lword4, K;
    mds = rf_gpr_read(src1);
    l = rf_gpr_read(src2);
    q0 = rf_gpr_read(src3);
    q1 = q0 + 256;
    y = rf_gpr_read(dst);

    q0_val = load_val(OPC_I_LBU, q0 + y);
    q1_val = load_val(OPC_I_LBU, q1 + y);

    q0_value10 = q0_val;
    q1_value10 = q1_val;
    q0_value11 = q0_val;
    q1_value11 = q1_val;
    K = (uint32)k;
    switch(K)
    {
      //kCycles = 3
      case 1:
      lword3 = load_val(OPC_I_LW, l + 16);
      q0_value10 = load_val(OPC_I_LBU, q0 + (q1_val ^ lword3[7..0]));
      q1_value10 = load_val(OPC_I_LBU, q1 + (q1_val ^ lword3[15..8]));
      q0_value11 = load_val(OPC_I_LBU, q0 + (q0_val ^ lword3[23..16]));
      q1_value11 = load_val(OPC_I_LBU, q1 + (q0_val ^ lword3[31..24]));
      break;
      //kCycles = 4
      case 2:
      lword3 = load_val(OPC_I_LW, l + 16);
      lword4 = load_val(OPC_I_LW, l + 24);
      q0_value10 = load_val(OPC_I_LBU, q1 + (q1_val ^ lword4[7..0])) ^
          lword3[7..0];
      q1_value10 = load_val(OPC_I_LBU, q1 + (q0_val ^ lword4[15..8])) ^
          lword3[15..8];
      q0_value11 = load_val(OPC_I_LBU, q0 + (q0_val ^ lword4[23..16])) ^
          lword3[23..16];
```

```
44        q1_value11 = load_val(OPC_I_LBU, q0 + (q1_val ^ lword4[31..24])) ^
              lword3[31..24];
45
46        q0_value10 = load_val(OPC_I_LBU, q0 + q0_value10);
47        q1_value10 = load_val(OPC_I_LBU, q1 + q1_value10);
48        q0_value11 = load_val(OPC_I_LBU, q0 + q0_value11);
49        q1_value11 = load_val(OPC_I_LBU, q1 + q1_value11);
50        break;
51      }
52
53      lword1 = load_val(OPC_I_LW, l);
54      lword2 = load_val(OPC_I_LW,l + 8);
55
56      q0_value2 = load_val(OPC_I_LBU, q0 + (q0_value10 ^ lword2[7..0]));
57      q0_value3 = load_val(OPC_I_LBU, q0 + (q1_value10 ^ lword2[15..8]));
58      q1_value2 = load_val(OPC_I_LBU, q1 + (q0_value11 ^ lword2[23..16]));
59      q1_value3 = load_val(OPC_I_LBU, q1 + (q1_value11 ^ lword2[31..24]));
60
61      tmp0 = load_val(OPC_I_LW,mds + 0*4*256 + 4*(q0_value2 ^ lword1[7..0])
            );
62      tmp1 = load_val(OPC_I_LW,mds + 1*4*256 + 4*(q0_value3 ^ lword1
            [15..8]));
63      tmp2 = load_val(OPC_I_LW,mds + 2*4*256 + 4*(q1_value2 ^ lword1
            [23..16]));
64      tmp3 = load_val(OPC_I_LW,mds + 3*4*256 + 4*(q1_value3 ^ lword1
            [31..24]));
65
66      rf_gpr_write(dst, tmp0 ^ tmp1 ^ tmp2 ^ tmp3);
67    };
68  };
```

A.3 TF_G

```
element i_tf_g
{
  use opc_tf_g as opc;
  use reg_any as dst, src1, src2;
  assembler { opc dst "," src1 "," src2};
  binary { opc[OPC_FRAG2] src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
  semantics
  {
    uXlen S, X, tmp0, tmp1, tmp2, tmp3;
    S = rf_gpr_read(src1);
    X = rf_gpr_read(src2);

    if(opc == OPC_CR_TF_G0)
    {
      tmp0 = load_val(OPC_I_LW, S + 0*4*256 + X[7..0]*4);
      tmp1 = load_val(OPC_I_LW, S + 1*4*256 + X[15..8]*4);
      tmp2 = load_val(OPC_I_LW, S + 2*4*256 + X[23..16]*4);
      tmp3 = load_val(OPC_I_LW, S + 3*4*256 + X[31..24]*4);
    }
    else {
      tmp0 = load_val(OPC_I_LW, S + 0*4*256 + X[31..24]*4);
      tmp1 = load_val(OPC_I_LW, S + 1*4*256 + X[7..0]*4);
      tmp2 = load_val(OPC_I_LW, S + 2*4*256 + X[15..8]*4);
      tmp3 = load_val(OPC_I_LW, S + 3*4*256 + X[23..16]*4);
    }
    rf_gpr_write(dst, tmp0 ^ tmp1 ^ tmp2 ^ tmp3);
  };
};
```

A.4 TF_RS

```
element i_tf_rs
{
  use opc_tf_rs as opc;
  use reg_any as src;
  assembler { opc src};
  binary { opc[OPC_FRAG2] UNUSED(10) opc[OPC_FRAG1] src opc[OPC_FRAG0]};
  semantics
  {
    uint8 t, bx, bxx, tmp0, tmp1, tmp2, tmp3;

    t = load_val(OPC_I_LBU, rf_gpr_read(src));
    tmp0 = load_val(OPC_I_LBU, rf_gpr_read(src) - 1);
    tmp1 = load_val(OPC_I_LBU, rf_gpr_read(src) - 2);
    tmp2 = load_val(OPC_I_LBU, rf_gpr_read(src) - 3);
    tmp3 = load_val(OPC_I_LBU, rf_gpr_read(src) - 4);
    bx = ((uint8)t << 1) ^ ((t[7..7]) ? 0x14d : 0);
    bxx = ((uint8)t >> 1) ^ ((t[0..0]) ? 0xa6 : 0) ^ bx;
    write_val(OPC_S_SB, rf_gpr_read(src) -1,tmp0^bxx);
    write_val(OPC_S_SB, rf_gpr_read(src) -2,tmp1^bx);
    write_val(OPC_S_SB, rf_gpr_read(src) -3,tmp2^bxx);
    write_val(OPC_S_SB, rf_gpr_read(src) -4,tmp3^t);

  };
};
```

A.5 DES_R

```
element i_des_r
{
  use opc_des_round as opc;
  use reg_any as dst, src1, src2, src3;
  assembler { opc dst "," src1 "," src2 "," src3};
  binary { UNUSED(2) src3 src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
  semantics
  {
    uXlen T,SK,X,Y,SB,sval0,sval1,sval2,sval3,
    addr0, addr1, addr2, addr3;
    uint8 tmp0, tmp1, tmp2, tmp3;
    SK = rf_gpr_read(src1);
    X = rf_gpr_read(src2);
    Y = rf_gpr_read(dst);
    SB = rf_gpr_read(src3);
    if(opc == OPC_CR_DES_R2)
    {
      X = ((X << 28) | (X >> 4));
```

```
19      }
20      T = SK ^ X;
21      tmp0 = T[5..0];
22      tmp1 = T[13..8];
23      tmp2 = T[21..16];
24      tmp3 = T[29..24];
25      if(opc == OPC_CR_DES_R1)
26      {
27        addr0 = SB + 7*64*4 + tmp0*4;
28        addr1 = SB + 5*64*4 + tmp1*4;
29        addr2 = SB + 3*64*4 + tmp2*4;
30        addr3 = SB + 1*64*4 + tmp3*4;
31      }
32      else
33      {
34        addr0 = SB + 6*64*4 + tmp0*4;
35        addr1 = SB + 4*64*4 + tmp1*4;
36        addr2 = SB + 2*64*4 + tmp2*4;
37        addr3 = SB + 0*64*4 + tmp3*4;
38      }
39      sval0 = load_val(OPC_I_LW, addr0);
40      sval1 = load_val(OPC_I_LW, addr1);
41      sval2 = load_val(OPC_I_LW, addr2);
42      sval3 = load_val(OPC_I_LW, addr3);
43
44      rf_gpr_write(dst, Y ^ sval0 ^ sval1 ^ sval2 ^ sval3);
45    };
46  };
```

A.6 DES_IP

```
1   element i_des_ip
2   {
3   use opc_des_ip as opc;
4   use reg_any as src1, src2;
5   assembler { opc src1 "," src2};
6   binary { opc[OPC_FRAG2] UNUSED(5) src2 opc[OPC_FRAG1] src1 opc[
        OPC_FRAG0]};
7   semantics
8   {
9   uXlen T,X,Y;
10  X = rf_gpr_read(src1);
11  Y = rf_gpr_read(src2);
12  T = ((X >> 4) ^ Y) & 0x0F0F0F0Fu;
13  Y ^= T;
14  X ^= (T << 4);
15  T = ((X >> 16) ^ Y) & 0x0000FFFFu;
16  Y ^= T;
17  X ^= (T << 16);
18  T = ((Y >> 2) ^ X) & 0x33333333u;
19  X ^= T; Y ^= (T << 2);
20  T = ((Y >> 8) ^ X) & 0x00FF00FFu;
21  X ^= T; Y ^= (T << 8);
22  Y = ((Y << 1) | (Y >> 31)) & 0xFFFFFFFFu;
23  T = (X ^ Y) & 0xAAAAAAAAu;
24  Y ^= T;
25  X ^= T;
26  X = ((X << 1) | (X >> 31)) & 0xFFFFFFFFu;
27  rf_gpr_write(src1, X);
28  rf_gpr_write(src2, Y);
29  };
30 };
```

A.7 DES_FP

```
element i_des_fp
{
  use opc_des_fp as opc;
  use reg_any as src1, src2;
  assembler { opc src1 "," src2};
  binary { opc[OPC_FRAG2] UNUSED(5) src2 opc[OPC_FRAG1] src1 opc[
      OPC_FRAG0]};
  semantics
  {
    uXlen T,X,Y;
    X = rf_gpr_read(src1);
    Y = rf_gpr_read(src2);
    X = ((X << 31) | (X >> 1)) & 0xFFFFFFFFu;
    T = (X ^ Y) & 0xAAAAAAAAu;
    X ^= T; Y ^= T;
    Y = ((Y << 31) | (Y >> 1)) & 0xFFFFFFFFu;
    T = ((Y >> 8) ^ X) & 0x00FF00FFu;
    X ^= T;
    Y ^= (T << 8);
    T = ((Y >> 2) ^ X) & 0x33333333u;
    X ^= T;
    Y ^= (T << 2);
    T = ((X >> 16) ^ Y) & 0x0000FFFFu;
    Y ^= T;
    X ^= (T << 16);
    T = ((X >> 4) ^ Y) & 0x0F0F0F0Fu;
    Y ^= T;
    X ^= (T << 4);
    rf_gpr_write(src1, X);
    rf_gpr_write(src2, Y);
  };
};
```

A.8 DES_SK2

```
element i_des_sk2
  {
  use opc_des_sk2 as opc;
  use reg_any as dst,src1,src2;
  assembler { opc dst "," src1 "," src2 };
  binary { opc[OPC_FRAG2] src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0] };
  semantics
  {
    uXlen X,Y, Z;
    X = rf_gpr_read(src1);
    Y = rf_gpr_read(src2);
    Z = ((X << 15) & 0x20000000) | ((X << 17) & 0x10000000)
    | ((X << 10) & 0x08000000) | ((X << 22) & 0x04000000)
    | ((X >> 2) & 0x02000000) | ((X << 1) & 0x01000000)
    | ((X << 16) & 0x00200000) | ((X << 11) & 0x00100000)
    | ((X << 3) & 0x00080000) | ((X >> 6) & 0x00040000)
    | ((X << 15) & 0x00020000) | ((X >> 4) & 0x00010000)
    | ((Y >> 2) & 0x00002000) | ((Y << 8) & 0x00001000)
    | ((Y >> 14) & 0x00000808) | ((Y >> 9) & 0x00000400)
    | ((Y ) & 0x00000200) | ((Y << 7) & 0x00000100)
    | ((Y >> 7) & 0x00000020) | ((Y >> 3) & 0x00000011)
    | ((Y << 2) & 0x00000004) | ((Y >> 21) & 0x00000002);
    rf_gpr_write(dst, Z);
  };
};
```

A.9 AES_ISR

```
element i_aes_isr
{
  use opc_aes_isr as opc;
  use reg_any as dst;
  use imm5 as src1;
  assembler { opc dst "," src1};
  binary { opc[OPC_FRAG2] UNUSED(5) src1 opc[OPC_FRAG1] dst opc[
      OPC_FRAG0]};
  semantics
  {
    uXlen state,rdata;
    uint5 row;
    state = rf_gpr_read(dst);
    row = src1;
    rdata = load_val(OPC_I_LW, state + row*4);
    switch(row){
      case 1:
      rdata = rdata[23..16] :: rdata[15..8] :: rdata[7..0] :: rdata
          [31..24];
      break;
      case 2:
      rdata = rdata[15..8] :: rdata[7..0] :: rdata[31..24] :: rdata
          [23..16];
      break;
      case 3:
      rdata = rdata[7..0] :: rdata[31..24]:: rdata[23..16] :: rdata
          [15..8];
      break;
    }
    write_val(OPC_S_SW, state + row*4, rdata);
  };
};
```

A.10 AES_ADD

```
element i_aes_add
{
  use opc_aes_add as opc;
  use reg_any as dst, src1, src2;
  use imm5 as src3;
  assembler { opc dst "," src1 "," src2 "," src3};
  binary { UNUSED(2) src3 src2 src1 opc[OPC_FRAG1] dst opc[OPC_FRAG0]};
  semantics
  {
    uXlen state, rkey, round, state_val;
    uint8 tmp0, tmp1, tmp2, tmp3;
    state = rf_gpr_read(dst);
    rkey = rf_gpr_read(src1);
    round = rf_gpr_read(src2);
    state_val = load_val(OPC_I_LW, state + 4*src3);
    tmp0 = load_val(OPC_I_LBU, rkey + round * 4 * 4 + 0*4 + src3) ^
       state_val[7..0];
    tmp1 = load_val(OPC_I_LBU, rkey + round * 4 * 4 + 1*4 + src3) ^
       state_val[15..8];
    tmp2 = load_val(OPC_I_LBU, rkey + round * 4 * 4 + 2*4 + src3) ^
       state_val[23..16];
    tmp3 = load_val(OPC_I_LBU, rkey + round * 4 * 4 + 3*4 + src3) ^
       state_val[31..24];
    write_val(OPC_S_SW, state + 4*src3, tmp3 :: tmp2 :: tmp1 :: tmp0);

  };
};
```

A.11 RSA_MC

```
element i_rsa_mc
{
  use opc_rsa_mc as opc;
  use reg_any as dst1, dst2, src1, src2;
  assembler { opc dst1 "," dst2 "," src1 "," src2};
  binary { UNUSED(2) src2 src1 dst2 opc[OPC_FRAG1] dst1 opc[OPC_FRAG0]};
  semantics
  {
    uXlen c,d,s,b, r0, r1;
    uint64 r;
    c = rf_gpr_read(dst1);
    d = rf_gpr_read(dst2);
    s = rf_gpr_read(src1);
    b = rf_gpr_read(src2);

    codasip_compiler_builtin();
    codasip_compiler_undefined();

    r = s* ((uint64) b);
    r0 = (uint32) r;
    r1 = (uint32)( r >> 32 );
    r0 += c;
    r1 += (r0 < c);
    r0 = r0 + d;
    r1 += (r0 < d);
    rf_gpr_write(dst1, r1);
    rf_gpr_write(dst2, r0);

  };
};
```

Příloha B

Obsah CD

Přiložené CD obsahuje následující soubory a adresáře:

- `xkosci00.pdf` – tato práce ve formátu PDF
- `tex/` – zdrojové texty této práce v jazyce LaTeX
- `crypto/` – zdrojové kódy implementací kryptografických algoritmů
- `model/` – zdrojové kódy instrukčního rozšíření a návod na jejich použití v modelu