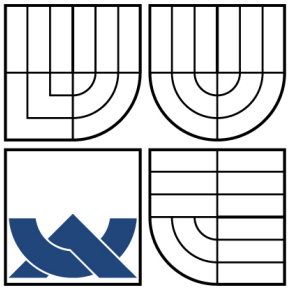
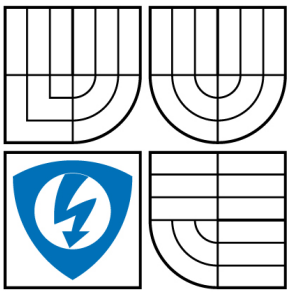


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

NÁVRH HARDWAROVÉHO ŠIFROVACÍHO MODULU

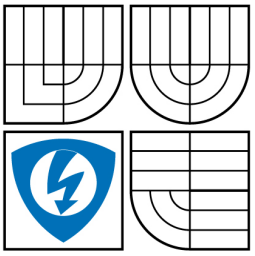
DESIGN OF HARDWARE CIPHER MODULE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. TOMÁŠ BAYER
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ SOBOTKA
SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student: Bc. Tomáš Bayer ***ID:*** 83937
Ročník: 2 ***Akademický rok:*** 2008/2009

NÁZEV TÉMATU:

Návrh hardwarového šifrovacího modulu

POKYNY PRO VYPRACOVÁNÍ:

Proveďte rozbor šifrovacích algoritmů vhodných pro implementaci do hardwarových kryptografických modulů. Na základě provedeného rozboru navrhněte koncepci jednoduchého hardwarového šifrovacího modulu, který umožní implementaci zvoleného šifrovacího algoritmu. Při návrhu šifrovacího modulu se zaměřte především na možnosti implementace kryptografických služeb určených pro šifrování dat.

DOPORUČENÁ LITERATURA:

[1] Schneier, B.: Applied Cryptography. John Wiley, N. York 1996. ISBN 0-471-11709-9
[2] Stallings, W.: Cryptography and Network Security: Principles and Practice. Prentice Hall, Englewood Cliffs 2003. ISBN: 0-13-091429-0

Termín zadání: 9.2.2009 ***Termín odevzdání:*** 26.5.2009

Vedoucí práce: Ing. Jiří Sobotka

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Anotace

Tato diplomová práce pojednává o problematice kryptografických systémů a šifrovacích algoritmů, u nichž je rozebráno, jak fungují, kde se využívají a jak se implementují v praxi. V první kapitole jsou uvedeny základní kryptografické pojmy, rozdělení algoritmů na symetrické a asymetrické a zhodnocení jejich použití a spolehlivostí. Následující kapitoly popisují substituční a transpoziční šifry, blokové a proudové šifry, z nichž je většina šifrovacích algoritmů odvozena. V neposlední řadě jsou jmenovány a popsány režimy, v nichž šifry pracují.

Ve čtvrté kapitole jsou popsány principy některých konkrétních šifrovacích algoritmů. Cílem je přiblížit podstatu fungování jednotlivých algoritmů. U těch složitějších algoritmů jako DES a GOST jsou pro lepší představu přiložena bloková schémata popisující průběh a pořadí prováděných operací. V závěru každého algoritmu je uveden příklad jeho použití v praxi.

Následující pátá kapitola pojednává o tématu hardwarové implementace šifer. V této kapitole je porovnána hardwarová implementace se softwarovou a to hlavně z praktického úhlu pohledu. Jsou popsány různé prostředky návrhu implementace a různé programovací jazyky pro návrh hardwarové implementace algoritmů. U programovacích jazyků jsou uvedeny jejich vývoj, výhody a nevýhody.

Kapitola šestá pojednává o samotném návrhu vybraných šifrovacích algoritmů. Konkrétně se jedná o návrh hardwarové implementace proudové šifry s generátorem pseudonáhodné posloupnosti založeným na LFSR navrhnuté v jazyku VHDL a také v programu Matlab. Jako druhý návrh hardwarové implementace byla zvolena bloková šifra GOST. Tato byla navržena v jazyce VHDL. Funkce obou návrhů implementací šifrovacích algoritmů byly otestovány a výsledky zhodnoceny.

Klíčová slova: kryptografie, kryptoanalýza, kryptologie, šifrování, dešifrování, symetrické a asymetrické šifry, substituční a transpoziční šifry, rotorové stroje, Vigenèrova šifra, Enigma, bloková šifra, režim Electronic Codebook, režim Cipher Block Chaining, inicializační vektor, proudová šifra, generátor reálné náhodné posloupnosti, generátor pseudonáhodné posloupnosti, režim Cipher-Feedback, režim Output-Feedback, režim čítače, Data Encryption Standard (DES), S-box substituce, P-box permutace, International Data Encryption Algorithm (IDEA), GOST 28147-89, A5, Linear Feedback Shift Register (LFSR), RC4, hardwarové šifrování, softwarové šifrování, programovací jazyky pro popis hardware, hardwarová implementace algoritmu, HDL, VHDL, Verilog, SystemVerilog, Handel-C, SystemC, SystemCrafter, návrh hardwarové implementace proudové šifry, návrh hardwarové implementace šifry GOST.

Abstract

This diploma's thesis discourses the cryptographic systems and ciphers, whose function, usage and practical implementation are analysed. In the first chapter basic cryptographic terms, symmetric and asymetric cryptographic algorithms and are mentioned. Also usage and reliability are analysed. Following chapters mention substitution, transposition, block and stream ciphers, which are elementary for most cryptographic algorithms. There are also mentioned the modes, which the ciphers work in.

In the fourth chapter are described the principles of some chosen cryptographic algorithms. The objective is to make clear the essence of the algorithms' behavior. When describing some more difficult algorithms the block scheme is added. At the end of each algorithm's description the example of practical usage is written.

The chapter no. five discusses the hardware implementation. Hardware and software implementation is compared from the practical point of view. Several design instruments are described and different hardware design programming languages with their progress, advantages and disadvantages are mentioned.

Chapter six discourses the hardware implementation design of chosen ciphers. Concretely the design of stream cipher with pseudo-random sequence generator is designed in VHDL and also in Matlab. As the second design was chosen the block cipher GOST, which was designed in VHDL too. Both designs were tested and verified and then the results were summarized.

Keywords: cryptography, cryptoanalysis, cryptology, enciphering, deciphering, symetric and asymetric ciphers, substitution and transposition ciphers, rotor machines, Vigenèr cipher, Enigma, block cipher, Electronic Codebook mode, Cipher Block Chaining mode, initialization vector, stream cipher, Pseudo-Random-Sequence Generator, real Random-Sequence Generator, Cipher-Feedback mode, Output-Feedback mode, Counter mode, Data Encryption Standard (DES), S-box substitution, P-box permutation, International Data Encryption Algorithm (IDEA), GOST 28147-89, A5, Linear Feedback Shift Register (LFSR), RC4, hardware enciphering, software enciphering, hardware design programming languages, hardware implementation of algorithm, HDL, VHDL, Verilog, SystemVerilog, Handel-C, SystemC, SystemCrafter, hardware implementation design of stream cipher, hardware implementation design of GOST cipher.

BAYER, T. *Návrh hardwarového šifrovacího modulu.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. 62 s. Vedoucí diplomové práce Ing. Jiří Sobotka.

Prohlášení o původnosti práce

Prohlašuji, že svou diplomovou práci na téma "Návrh hardwarového šifrovacího modulu" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.“

V Brně dne …… ……..

(podpis autora)

Poděkování

Děkuji vedoucímu diplomové práce Ing. Jiřímu Sobotkovi za užitečnou metodickou pomoc, odborné vedení a rady při zpracování diplomové práce. Rovněž nemohu opomenout poděkovat kolegovi Bc. Jaroslavu Švédovi za cenné rady, které zpracování praktické části diplomové práce značně urychlily a ušetřily tápání.

V Brně dne …… ……..

(podpis autora)

Obsah

Úvod

Můžeme rozlišit dva druhy kryptografie: ta první znemožní, aby si vaše soubory prohlížel malý sourozenec, druhá zabrání v prohlížení vašich souborů osobami, které by tyto soubory dokázali zneužít ve váš neprospěch. Předpokládám, že malý sourozenec nezneužije něčí osobní údaje k nějaké trestné činnosti. Může se jednat o jakákoliv data, například emailová korespondence, fotky z dovolené, technická dokumentace k vašemu novému vynálezu, ale také osobní údaje, informace o bankovním kontu, atd. Ať už by se jednalo o jakkoli důvěrná či choulostivá data, jejich majiteli by se jistě nelíbilo, kdyby se tyto dostaly do nepovolaných rukou a způsobily mu například finanční škodu nebo poškodily jeho dobré jméno apod. Jiný stupeň ochrany bude vyžadovat dokument obsahující výrobní či obchodní tajemství a jiný stupeň zase komunikační kanál spojeneckých vojsk ve válce. Ale oba tyto případy spojuje fakt, že oprávněný uživatel může dokument prohlížet, nebo komunikovat se spojenci, pouze pomocí klíče, který by měl zůstat pro veškeré ostatní osoby nedostupný. Takovým klíčem může být buď znalost hesla, nebo čipová karta, či otisk prstu.

Tato práce si klade za cíl popsat problematiku kryptografických systémů a šifrovacích algoritmů, u nichž bude rozebráno, jak fungují, případně jak se vyvíjely, dále kde se využívají a jak jsou implementovány v praxi. V první kapitole se seznámíte se základními pojmy používanými v kryptografii, se základním rozdělením algoritmů a jejich spolehlivostí. Následují kapitoly popisující základní šifry, blokové a proudové šifry, z nichž je většina současných šifrovacích algoritmů odvozena, a také budou jmenovány režimy, v nichž šifry pracují. V kapitole 4 se potom seznámíte s principy některých konkrétních algoritmů. Následující kapitola pojednává o již tématu hardwarové implementace šifer a rozebírá možné prostředky návrhu implementace jako například programovací jazyk VHDL a další. Kapitola 6 pojednává o samotném návrhu dvou vybraných šifrovacích algoritmů.

1 Základní pojmy

Účelem a cílem kryptografie je ochránit důvěrná data úplně nebo jejich získání pokud možno do vysoké míry omezit a ztížit, aby se útočník o jejich získání přestal pokoušet. U všech kryptografických ochran je totiž pro umožnění přístupu k datům potřeba vyřešit nějaký matematický problém. Tyto matematické problémy jsou voleny tak, aby je útočník nemohl být schopen vyřešit v reálném čase. A pouze uživatel mající tajný klíč, který napomůže matematický problém vyřešit úspěšně a v reálném čase, získá přístup k chráněným datům [1].

Kryptografickou ochranu tedy můžeme definovat jako technickou ochranu, která je založena na obtížnosti řešení matematických problémů [2]. Problematikou konstrukcí kryptografických ochran se zabývá věda nazývaná **kryptografie** [2]. Na druhou stranu věda zkoumající prolomení kryptografických ochran se nazývá **kryptoanalýza**. Obě předchozí vědy sdružuje matematický obor kryptologie, jejíž odborníci zvaní kryptologové musejí mít velmi dobré znalosti z teoretické matematiky.

1.1 Šifrování, dešifrování

Zpráva, kterou hodláme zašifrovat, se označuje M (z angl. message) nebo někdy také P (z angl. plaintext). Můžou jí být posloupnost bitů, bitová mapa, textový soubor, prostý text, záznam hlasu, video, atd. jednoduše řečeno jde o binární data. Zprávu můžeme chtít buď uložit nebo odeslat, ale v obou případech to provedeme bezpečně a zprávu pomocí některé z kryptografických ochran zašifrujeme. Šifrování je tedy proces maskování zprávy ve smyslu ukrytí jejího obsahu, tzn. že případný útočník odhalí pouze fakt, že nějaká zprávy byla odeslána, nedozví se však její obsah. Takováto zašifrovaná zpráva se nazývá kryptogram a značí se písmenem C (z angl. ciphertext). Kryptogram jsou také binární data, jejichž velikost je někdy stejná jako zpráva M, ale někdy i větší. Proces, který přemění kryptogram zpět na původní zprávu se nazývá dešifrování. Šifrovací funkce značená E (z angl. encryption) pracuje se zprávou M, aby vytvořila kryptogram C, neboli matematicky vyjádřeno:

$$E(M) = C . \tag{1.1}$$

Inverzní funkce, tedy dešifrovací, značená D (z angl. decryption) operuje s C a výsledkem je původní zpráva M, matematicky:

$$D(C) = M . \tag{1.2}$$

Když do výše uvedené rovnice dosadíme za C z rovnice předešlé, dostaneme vztah popisující zašifrování zprávy a následné dešifrování jejího kryptogramu:

$$D(E(M)) = M . \tag{1.3}$$

1.2 Algoritmy a klíče

Pro šifrování a dešifrování se používají vzájemně závislé matematické funkce zvané kryptografické algoritmy neboli šifry. Obě operace používají klíč, což je množství symbolů, které může být a nemusí pro operaci šifrování stejné jako pro operaci dešifrování. Klíč se značí *K* (z angl. key) a rozsah jeho možných hodnot, které se při šifrování mohou použít, se nazývá množina klíčů [1]. Podle toho, zda se pro šifrování a dešifrování používají stejné či odlišné klíče, se rozlišují dva základní druhy algoritmů: symetrické (viz.kapitola 1.3) a asymetrické (viz.kapitola 1.4).

1.3 Symetrické algoritmy

U symetrických algoritmů, též zvaných algoritmy s tajným klíčem, se šifrovací a dešifrovací klíč shodují nebo se z šifrovacího klíče dá snadno výpočetně odvodit klíč dešifrovací. Odesílatel a příjemce zprávy tudíž musí klíče držet v tajnosti, protože při vyzrazení klíče by mohl kdokoliv dešifrovat jejich zprávy a zjistit tak jejich obsah anebo by mohl zprávy dešifrovat, pozměnit jejich obsah a opět zašifrovat a příjemce by tak obdržel znehodnocenou zprávu. Tyto algoritmy jsou na jednu stranu velmi rychlé, ale „problémem je bezpečná distribuce klíčů od zdroje klíčů k odesílateli a příjemci“ [2]. Pokud označíme šifrovací a dešifrovací funkce za použití klíče *K* jako E_K a D_K, budou tyto funkce matematicky vyjádřeny následovně:

$$E_K(M) = C, \tag{1.4}$$

$$D_K(C) = M. \tag{1.5}$$

Při dosazení z první rovnice do druhé, dostaneme opět vztah pro zašifrování zprávy a následné dešifrování jejího kryptogramu:

$$D_K(E_K(M)) = M. \tag{1.6}$$

1.4 Asymetrické algoritmy

Algoritmy s veřejným klíčem, jak se dají asymetrické algoritmy také nazývat, používají odlišný klíč pro šifrování (označený K1) a pro dešifrování (označený K2). Navíc se dešifrovací klíč nedá v reálném čase ani odvodit z toho šifrovacího. Algoritmy s veřejným klíčem se tak nazývají proto, že jeden z klíčů je veřejně známý, kdežto ten druhý je tajný. Takže například když bude zveřejněn šifrovací klíč, může ním zprávu zašifrovat kdokoli, ale pouze specifická osoba vlastnící tajný klíč může tuto zprávu dešifrovat a přečíst. Tím je zajištěna důvěrnost a integrita přenesené zprávy [2]. Existuje i opačný případ, kdy je tajným klíčem šifrovací klíč a veřejným je klíč dešifrovací. To znamená, že zpráva může být zašifrována pouze osobou s tajným klíčem, ale dešifrovat ji může kdokoli. Tak je zaručena autentičnost přenesené zprávy (tzv. digitální podpis) [2]. Šifrovací a dešifrovací funkce budou matematicky vyjádřeny takto:

$$E_{K1}(M) = C, \tag{1.7}$$

$$D_{K2}(C) = M, \; D_{K2}(E_{K1}(M)) = M. \tag{1.8}$$

1.5 Kryptoanalýza

Účelem kryptografie je zabránit naslouchajícímu útočníkovi v přístupu k chráněným datům nebo k dešifrovacímu klíči. Předpokládá se, že takový útočník má neomezený přístup ke komunikaci mezi odesílatelem a příjemcem [1].

Kryptoanalýza je věda zabývající se získáním (rozluštěním) zprávy M z kryptogramu C bez znalosti klíče K. V případě úspěšné kryptoanalýzy je možné získat přímo původní zprávu či klíč nebo v analyzovaném kryptosystému nalézt skulinu, která by eventuálně vedla ke stejným výsledkům, tj. k nalezení zprávy či klíče. [1]

1.6 Spolehlivost algoritmů

Stupeň spolehlivosti algoritmu závisí na obtížnosti jeho prolomení. Data budou pravděpodobně v bezpečí, pokud náklady na prolomení ochrany převyšují hodnotu dat. Když čas potřebný k prolomení algoritmu je delší než doba, po kterou data musejí zůstat utajena, jsou data také pravděpodobně v bezpečí. I v případě, kdy množství dat potřebných k dešifrování je větší než množství zašifrovaných dat, budou data pravděpodobně v bezpečí. Slovo „pravděpodobně“ bylo použito z toho důvodu, že vždy existuje šance, že dojde k nějakému zásadnímu průlomu v oblasti kryptoanalýzy a data už v bezpečí nebudou.

Lars Knudsen rozdělil způsoby prolomení algoritmu v [3] takto:

1. Totální průnik. Kryptoanalytik zjistí klíč K takový, že platí:

$$D_K(C) = M . \tag{1.9}$$

2. Celkové (komplexní) odvození. Kryptoanalytik použije alternativní algoritmus, ekvivalentní k $D_K(C)$, bez znalosti klíče K.
3. Částečné odvození. Kryptoanalytik objeví rozšifrovaný text k zachycenému kryptogramu.
4. Odvození informací. Kryptoanalytik získá nějaké informace o klíči nebo zprávě (například několik málo bitů klíče, forma, velikost zprávy, atd.).

Algoritmus je bezpodmínečně bezpečný (zajišťuje tedy dokonalé utajení) v případě, že nezáleží na tom, kolik kryptogramů má kryptoanalytik k dispozici, stejně nemá dostatek informací k dešifrování zprávy. Po vyzkoušení všech možných klíčů zjistí, že luštěný kryptogram mohl vzniknout zašifrováním libovolné ze všech možných zpráv [2]. Kryptografie se více zabývá kryptosystémy, které jsou výpočetně neprolomitelné. Algoritmus je považován za výpočetně bezpečný (odolný), pokud nemůže být prolomen dostupnými prostředky současnými nebo budoucími [1].

2 Substituční a transpoziční šifry

V dobách před výpočetní technikou sestávala kryptografie z algoritmů založených na písmenech. Různé druhy algoritmů pracovaly tak, že zaměňovaly písmena za jiná písmena (substituce) anebo písmena ve zprávě rozmístily na jiné pozice (transpozice). A některé z těch lepších algoritmů uměly provádět oboje. V současné době je to vše komplikovanější, ale filozofie zůstává stejná [1]. Mnoho dobrých algoritmů stále používá prvky substituce či transpozice. Základní změna oproti minulosti tkví v tom, že namísto 26-ti písmen algoritmy pracují se dvěma prvky, bitovou 1 a bitovou 0.

2.1 Substituční šifry

V tomto algoritmu se každý znak (písmeno) vstupní zprávy substituuje jiným písmenem. Známá je Caesarova šifra, kde je písmeno zprávy zaměněno za písmeno ležící v abecedě o 3 místa vpravo. Takže například písmeno A je nahrazeno písmenem D, písmeno W je zastoupeno písmenem Z, X je substituováno za A atd. Existují čtyři typy substitučních algoritmů:

- **Jednoduché substituční šifry,** kde je každý znak vstupní zprávy vyměněn odpovídajícím znakem kryptogramu.
- **Homofonní substituční šifry** jsou podobně těm předešlým s tím rozdílem, že jednomu znaku zprávy může odpovídat více znakům v kryptogramu (například písmenu B může odpovídat 7, 19, 31 či 42).
- **Polygamní substituční šifry** šifrují znaky po skupinách. Například zpráva „ABA“ může být zašifrována jako „RTQ“.
- **Polyalfabetické substituční šifry** se skládají z několika jednoduchých šifrátorů. Takže může být použito například pět různých jednoduchých šifrátorů, kde každý z nich šifruje jiné písmeno. První šifruje první písmeno zprávy, druhý šifrátor šifruje druhé písmeno zprávy atd. [1]

2.2 Transpoziční šifry

Obsah zprávy se rovná obsahu kryptogramu u této šifry, jen písmena jsou zamíchána. U jednoduché sloupcové transpoziční šifry je zpráva psána horizontálně na papír s pevně stanovenou šířkou a kryptogram vznikne přečtením tohoto papíru vertikálně a zapsáním na jiný papír horizontálně. Dešifrování potom probíhá tak, že se kryptogram zapíše vertikálně na papír o dané šířce a potom se přečte horizontálně. Z historie je známá například německá šifra ADFGVX používaná za První světové války. Je kombinací transpoziční a substituční šifry a na svou dobu byla velice složitým algoritmem, ale byla rozluštěna francouzským kryptoanalytikem Georgesem Painvinem. [1]

2.3 Rotorové stroje

Ve dvacátých letech 20.století byly vynalezeny různé druhy mechanických šifrovacích zařízení, která měla zautomatizovat šifrovací proces. Většina z nich byla založena na principu mechanicky poháněného kola (rotoru), které vykonávalo substituci, přičemž se vlastně jednalo o verzi Vigenèrovy šifry [1]. Takový rotorový stroj měl klávesnici a několik řad rotorů, z nichž každý byl libovolnou permutací abecedy, měl 26 pozic a vykovával jednoduchou substituci [1]. Výstup rotoru byl vždy připojen na vstup rotoru následujícího. Takže například u stroje se čtyřmi rotory první rotor substituoval A za F, druhý rotor zaměňoval F za Y, třetí provedl substituci Y za E, čtvrtý rotor substituoval E za C a výstupním kryptogram by tak v tomto případě bylo C. Poté některé rotory změní svá nastavení a při dalším šifrování bude substituce odlišná. Právě změna nastavení jednotlivých rotorů probíhající u každého z rotorů o jiný počet písmen zajišťovala bezpečnost stroje [1]. Perioda, za kterou se bude opakovat již jednou použité nastavení rotorů, je u n-rotorového stroje 26^n [1].

Nejznámějším rotorovým strojem je Enigma, používaná Němci za Druhé světové války. Pro zajímavost ještě uvádím, že byla vynalezena Arthurem Scherbiusem, který ji nechal patentovat ve Spojených státech, a v průběhu Druhé světové války došlo k jejímu prolomení, které měli na svědomí polští kryptografové [1].

3 Druhy a režimy algoritmů

Symetrické algoritmy se dají rozdělit na dva základní typy: proudové a blokové šifry. Proudové šifry pracují se zprávou bit po bitu, kdežto blokové algoritmy šifrují zprávu po skupinách bitů (blocích) a výsledný kryptogram vznikne napojením zašifrovaných bloků za sebe.

Kryptografický režim v sobě spojuje základní šifrovací algoritmus, nějaký druh zpětné vazby a nějaké prosté operace. Prosté operace proto, aby šifrovací režim nesnížil bezpečnost základního algoritmu. Dále by mělo být zajištěno, aby vstup do šifrátoru byl volen náhodně, aby vnášení chyb do kryptogramu pomocí manipulace se zprávou bylo velmi obtížné a aby bylo možné pomocí stejného klíče šifrovat více než jednu zprávu. Kromě toho by režim neměl mít výrazně nižší účinnost než samotný algoritmus. Je také velmi důležité, aby dešifrovací proces byl odolný proti chybám v kryptogramu nebo byl schopen rekonstruovat zprávu při ztrátě či naopak přebytku nějakého bitu v kryptogramu. Rozdílné režimy se v těchto vlastnostech liší. [1]

3.1 Blokové šifry

Délka jednotlivých bloků zprávy, respektive kryptogramu bývá 64, 128 nebo 256 bitů. Základem blokové šifry jsou tzv. blokové operace, které jednomu nebo několika blokům bitů přiřazují jeden výstupní blok bitů. Blokové operace se dělí na permutační, substituční a aritmetické. Následuje přehled nejpoužívanějších provozních režimů blokových šifer.

3.2 Režim Electronic Codebook (ECB)

ECB je nejsnáze pochopitelná bloková šifra. Zašifrováním totožného bloku zprávy vznikne vždy totožný blok kryptogramu. Teoreticky je tedy možné vytvořit seznamy kódů (code book), které by obsahovaly zprávy a k nim odpovídající kryptogramy.

Každý blok zprávy může být zašifrován nezávisle, není tedy třeba šifrovat bloky postupně od prvního po poslední, můžou se zašifrovat nejprve bloky ze středu, potom bloky na konci a v závěru bloky ze začátku. Toto je výhodné například u databází, kde je potřeba k zašifrovaným souborům přistupovat nezávisle v libovolném pořadí. Každý záznam tak může být zapsán, smazán, šifrován či dešifrován nezávisle na ostatních záznamech. Zpracování je možné paralelizovat tak, že několik šifrovacích procesorů šifruje či dešifruje různé bloky najednou bez toho, aniž by se vzájemně ovlivňovali.

Problém u ECB režimu je, že pokud kryptoanalytik zachytí několik zpráv a k nim odpovídající kryptogramy, může si začít tvořit code book (kódovou knihu) bez znalosti klíče. U zpráv tvořených počítačem, například emaily, je velká míra opakujících se částí, z nichž může kryptoanalytik získat velké množství informací. Takové zprávy totiž mají přesně

definovaná záhlaví a zápatí obsahující informace o odesílateli, příjemci, atd., které může případný útočník zneužít.

Kladnou vlastností může být fakt, že není nebezpečné šifrovat více zpráv pomocí stejného klíče. Další plus je, že v případě výskytu chyb v některém bloku kryptogramu bude špatně dešifrován pouze ten některý blok zprávy, ostatní bloky budou dešifrovány správně. Na druhé straně, pokud náhodou dojde ke ztrátě či přidání bitu v kryptogramu, všechny následující bity kryptogramu budou dešifrovány nesprávně.

3.2.1 Vyplňování

ECB vyžaduje bloky o velikosti 64 bitů. Většina zpráv se ale nerozdělí úhledně do 64-bitových bloků, na konci bývá blok o menší velikosti. Tento problém řeší právě **vyplnění** posledního bloku nějakými znaky, většinou nulami nebo jedničkami, aby jeho velikost dosáhla 64 bitů. Pokud je potřeba tyto znaky po dešifrování vymazat, do posledního bytu takto vyplněného bloku se uloží počet vyplňujících bitů. [1]

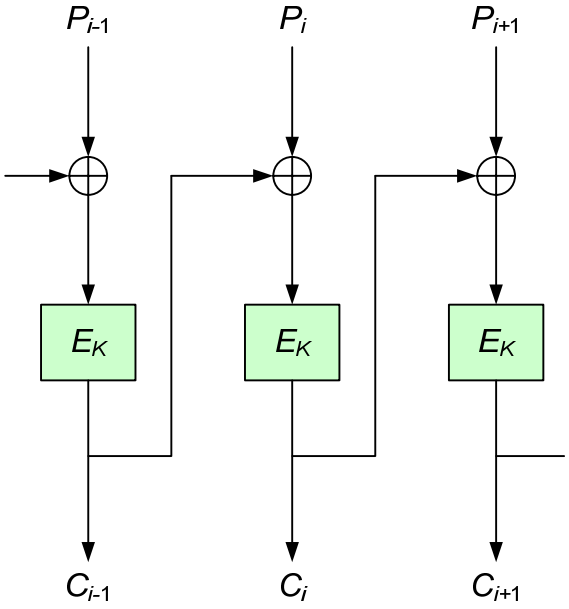
3.3 Režim Cipher Block Chaining (CBC)

Tento režim přidává k blokovému šifrovacímu algoritmu zpětnou vazbu, která způsobí, že výsledek šifrování předchozího bloku se pošle zpět na začátek a šifruje se spolu s novým blokem. Každý blok tedy ovlivňuje šifrování následujícího bloku a každý blok kryptogramu je závislý na všech předešlých blocích zprávy.

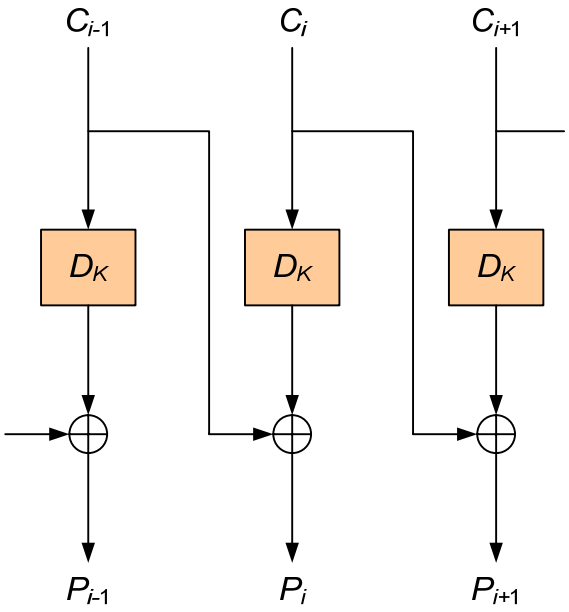
Jak funguje CBC. Blok zprávy je xorován s předchozím blokem kryptogramu a výsledek této operace se zašifruje. Přesně řečeno poté, co je zpráva zašifrována, je výsledek C_{i-1} uložen v registru zpětné vazby. Než dojde k šifrování následujícího bloku P_i, tento blok se xoruje s předchozím blokem kryptogramu C_{i-1} a výsledek je poté šifrován. Vznikne další blok kryptogramu C_i, který je opět uložen do registru zpětné vazby, poté xorován s dalším blokem zprávy a zašifrován atd. až do konce zprávy. Při dešifrování se blok kryptogramu C_{i-1} dešifruje, zároveň se jako nedešifrovaný uloží v registru zpětné vazby a po dešifrování dalšího bloku C_i je blok C_{i-1} z registru xorován s výsledkem dešifrování bloku C_i a vznikne tak blok zprávy P_i. Procesy šifrování a dešifrování jsou popsány v obrázcích Obr. 3.1 a Obr. 3.2 a popisují je také následující matematické rovnice:

$$C_i = E_K\left(P_i \bullet C_{i-1}\right), \tag{3.1}$$

$$P_i = C_{i-1} \bullet D_K\left(C_i\right). \text{ [1]} \tag{3.2}$$



Obr.3.1: Šifrování v režimu Cipher Block Chaining.



Obr.3.2: Dešifrování v režimu Cipher Block Chaining.

3.3.1 Inicializační vektor

V CBC režimu se dvě totožné zprávy zašifrují do totožných kryptogramů, co hůř, dvě zprávy mající shodné počáteční bloky se zašifrují do stejné podoby až po první rozdíl mezi zprávami. Jak již bylo uvedeno u ECB, některé zprávy mají shodná záhlaví, takže případný útočník by mohl získat cenné informace. Tomuto se předchází tak, že se namísto počátečního bloku zprávy zašifruje blok obsahující náhodná data. Tento blok náhodných dat se nazývá inicializační vektor (nebo taky inicializační proměnná). Inicializační vektor tak

promění dvě zprávy se shodným začátkem na dvě rozdílné unikátní zprávy. A taktéž shodné zprávy budou při použití inicializačního vektoru zašifrovány do odlišných kryptogramů a to vše při použití stejného klíče.

3.3.2 Vyplňování

Vyplňování posledních bloků zpráv nulami či jedničkami tak, aby dosáhli požadované velikosti, pracuje stejně jako u ECB režimu. V některých aplikacích však musí být velikost kryptogramu stejná jako velikost zprávy.

3.3.3 Množení chyb

Chyba v jednom bitu bloku zprávy ovlivní následující blok kryptogramu a také všechny další bloky kryptogramu. To ale není podstatné, protože dešifrováním, jakožto inverzní operací, se toto otočí a v dešifrované zprávě bude pouze ta jedna chyba v jednom bitu.

Horší to je v případě, kdy vznikne chyba v kryptogramu, například za přítomnosti rušení v přenosovém kanále nebo selháním úložného média. V režimu CBC taková chyba ovlivní celý blok a jeden bit dalšího bloku dešifrované zprávy. Blok obsahující chybu bude zcela nepoužitelný a následující blok bude mít pouze chybu v jednom bitu na stejné pozici, jako byla chyba v kryptogramu. Toto tzv. rozšíření chyby je hlavním neduhem režimu CBC.

Na druhou stranu ale nejsou kromě postiženého bloku a jednoho bitu bloku druhého ovlivněna či znehodnocena kterákoliv další data zprávy. Dá se tedy říci, že CBC režim je samozotavující se. Chybou jsou ovlivněny pouze dva bloky, ale systém se zotaví a dál pokračuje v práci s ostatními bloky korektně.

Zatímco se ale CBC zotaví z bitové chyby, při ztrátě nebo přidání bitu v kryptogramu jsou všechny následné bloky o ten jeden bit posunuty a tím pádem naprosto znehodnoceny. Při používání CBC se teda musí požadovat a zajistit neporušenost struktury bloků.

3.3.4 Problémy s bezpečností

V případě, že útočník by na konec zašifrované zprávy přidal libovolné bloky, nebude tato skutečnost odhalena. Dešifrování zprávy tak bude mít za výsledek nějaký nesmysl, což je nepřípustné. Takže v případě používání CBC by měli uživatelé vědět, kde má zpráva své hranice, a odhalit tak případné přidané bloky.

Paradox vytvoření říká, že budou vytvořeny dva identické bloky po uplynutí $2^{m/2}$ bloků, kde m je velikost bloku. Takže například při velikosti bloku 64 bitů to činí 34gigabytů. Zpráva by tedy musely být moc dlouhá, aby k tomuto problému vůbec došlo. [1]

3.4 Proudové šifry

Jak již bylo řečeno v úvodu kapitoly 3, proudové šifry pracují s šifrovanou zprávou bit po bitu. Základní princip proudové šifry je zobrazen ve schématu v Obr. 3.3. Proudová šifra tedy funguje tak, že proud bitů hesla k_i se xoruje s proudem bitů vstupní zprávy p_i a vzniká proud bitů kryptogramu c_i. Hesla jsou na základě klíče K tvořena pomocí generátorů hesel, které pracují synchronně. Takže při dešifrování se generátoru hesel zadá totožný klíč K jako

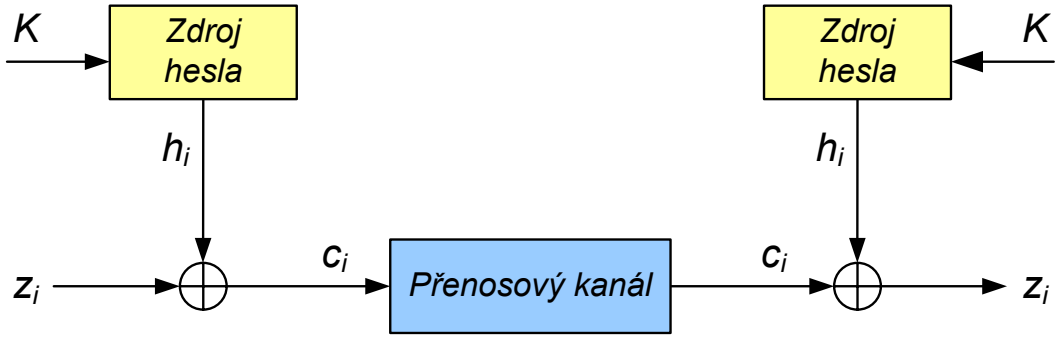
byl zadán při šifrování a je vygenerováno stejné heslo jako na straně šifrátoru. Při dešifrování jsou bity kryptogramu c_i xorovány s bity hesla k_i (totožného jako při šifrování) a vznikne původní zpráva p_i. Princip šifrování a dešifrování je popsán následujícími rovnicemi:

$$c_i = p_i \bullet k_1, \tag{3.3}$$

$$p_i = c_i \bullet k_1. \tag{3.4}$$

Dosazením z první rovnice do druhé za c_i je zřejmé, že platí následující:

$$p_i \bullet k_i \bullet k_1 = p_i. \tag{3.5}$$



Obr. 3.3: Schéma proudové šifry.

Bezpečnost systému s proudovou šifrou závisí na způsobu, jakým generátor hesel hesla vytváří. Pokud by vytvářel jen proud samých nul, tak kryptogram bude mít shodnou podobu se vstupní zprávou a celé šifrování přijde nazmar [1]. Stejně tak když bude generovat opakující se 16-bitové vzorky hesel, bezpečnost bude jen zanedbatelná. Anebo když po každém zapnutí vygeneruje shodnou posloupnost bitů, bude velmi snadné šifru prolomit. Generátor by tedy měl vychrlit nekonečnou opravdu náhodnou posloupnost bitů, aby byla zajištěna dokonalá bezpečnost. Realita je ale někde mezi, výstup generátoru se pouze zdá být zcela náhodný. Čím více je generované heslo náhodné, tím delší čas zabere útočníkovi prolomit tuto šifru. [1]

3.4.1 Generátor reálné náhodné posloupnosti

Tyto generátory využívají náhodnou posloupnost, která je odvozena od některého náhodného fyzikálního děje jako například rozpad atomů nebo teplotní šum apod. V případě dvou fyzikálně shodných takových generátorů nebude nikdy produkována stejná posloupnost. Z tohoto důvodu se musí náhodná posloupnost zapsat na dvě paměťová média, jedno bude jako heslo pro šifrátor a druhé pro dešifrátor.

Speciálním případem šifrátoru s generátorem náhodné posloupnosti je dokonalá šifra. Jedná se o jedinou šifru poskytující dokonalé utajení, což znamená, „že ji nelze bez znalosti klíče vyluštit a to i v případě použití neomezených výpočetních a paměťových zdrojů“ [2]. Dokonalá šifra je založena na schématu podle Obr. 3.3. Zdrojem hesla je posloupnost z paměťového média. Šifrovacím a dešifrovacím klíčem K je samotná heslová posloupnost [2]. Aby bylo dosáhnuto dokonalého utajení, musí být klíč stejně dlouhý jako

zpráva, klíč musí být zcela náhodný a každá zpráva musí být zašifrována jiným klíčem. Problémem u dokonalé šifry je bezpečná distribuce klíče, bez níž by se potom nedalo mluvit o dokonalém utajení. Proto se v současné době uvedený typ šifrátoru používá jen pro šifrování zpráv s nejvyšším stupněm utajení [2]. S vývojem paměťové technologie je však použití dokonalé šifry možné i v jiných aplikacích.

3.4.2 Generátor pseudonáhodné posloupnosti

Generátory pseudonáhodné posloupnosti (též označované PNP) produkují posloupnost na základě předem daného výchozího stavu. Takto vzniklá posloupnost se na základě statistických testů jeví jako náhodná. U šifrátoru a dešifrátoru se tedy klíčem K nastaví výchozí stav a od něho se pak odvozují totožné heslové postupnosti [2]. Generátory PNP jsou buď hardwarové nebo softwarové. HW generátory jsou jednoúčelové elektronické obvody, které poskytují heslovou postupnost o rychlosti až desítek Mb/s. SW generátory jsou sice pomalejší, avšak nevyžadují speciální obvody. [2]

V praxi se používají HW generátory založené na některé kombinaci posuvných registrů se zpětnou vazbou (např. šifra A5 v systému GSM) [2]. Základním stavebním kamenem těchto generátorů PNP je lineární generátor s posuvným registrem a s lineární zpětnou vazbou (LFSR – Linear Feedback Shift Register). Jeho princip bude vysvětlen v pozdější kapitole popisující šifru A5.

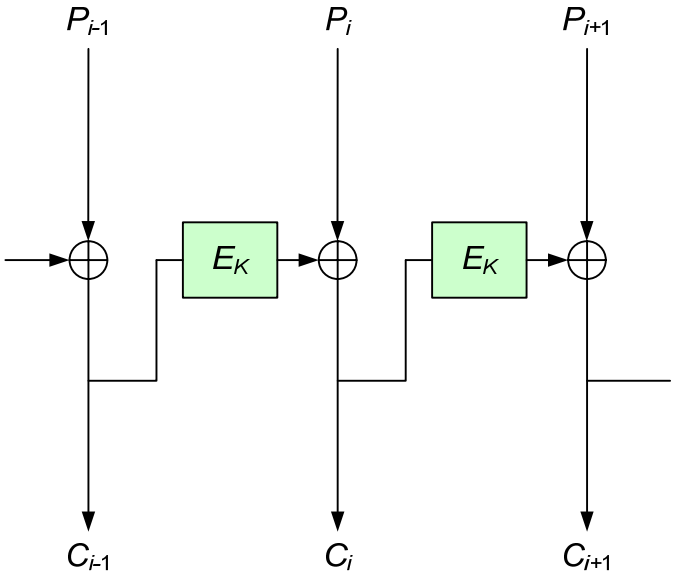
3.5 Režim Cipher-Feedback (CFB)

Bloková šifra může být také implementována jako samosynchronizující proudová šifra [1]. V režimu CFB mohou být šifrovány data o menší velikosti než je velikost bloku. Namísto 64 bitů je tak možno šifrovat třeba 8-bitová data (8-bit CFB). Obecně tedy lze šifrovat data o velikosti n, kde n je menší nebo rovno velikosti bloku (n-bit CFB). Při šifrování se pracuje s frontou o stejné délce jako je velikost bloku. Na začátku je tato fronta naplněna inicializačním vektorem IV stejně jako tomu bylo u CBC režimu.

Princip 8-bit CFB je takový, že nejdříve se vezme prvních 8 bitů z fronty, tj.prvních 8 bitů inicializačního vektoru IV, ty se zašifrují podle klíče K a poté se xorují s prvními 8 bity zprávy P_i a vzniká tak prvních 8 bitů kryptogramu C_{i-1}. Těchto 8 bitů kryptogramu se odešle a zároveň se vloží na konec vstupní fronty, čímž se data ve frontě posunou a prvních 8 bitů je tak vyřazeno z fronty. Postupně se data ve frontě obmění, až dojde k tomu, že vstupem šifrování bude oněch prvních 8 bitů kryptogramu C_{i-1} a začne se šifrovat další blok zprávy. Proces šifrování popisuje Obr. 3.4. Proces dešifrování je obrácený, příchozí blok zašifruje podle stejného klíče K, pošle do fronty a poté xoruje s následujícím blokem. Šifrování a dešifrování popisují níže uvedené rovnice:

$$C_i = P_i \bullet E_K\left(C_{i-1}\right), \tag{3.6}$$

$$P_i = C_i \bullet E_K\left(C_{i-1}\right). \text{ [1]} \tag{3.7}$$



Obr.3.4: Šifrování v režimu Cipher-Feedback.

Inicializační vektor musí stejně jako v případě režimu CBC utajený. Na rozdíl od CBC režimu, musí být inicializační vektor u CFB režimu unikátní, musí se tudíž pro každou další šifrovanou zprávu tvořit nový. Pokud by nebyl, útočník by mohl odhalit obsah posílaných zpráv.

3.6 Režim Output-Feedback (OFB)

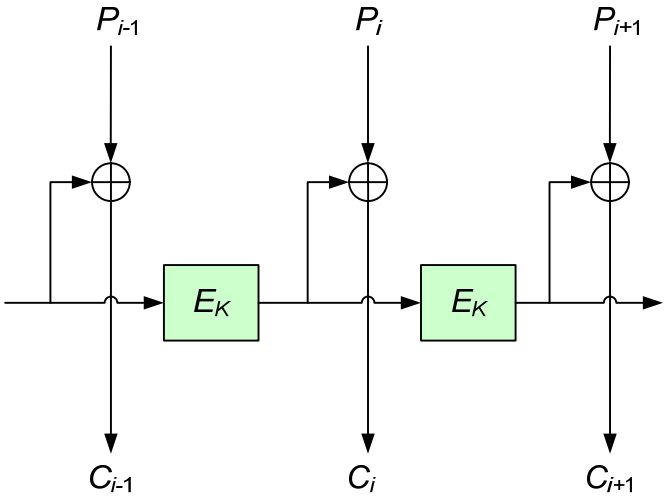
Jedná se o metodu provozující blokovou šifru jako synchronní proudovou šifru. Režim OFB je podobný režimu CFB až na to, že na konec fronty (vpravo) je přesunuto *n* bitů vystupujících už z operace šifrování, nikoli *n* bitů vystupujících až z operace xor jako tomu bylo u CFB. Režim OFB popisuje Obr. 3.5. Používá se označení *n*-bit OFB. U šifrování i dešifrování je blok algoritmu použit v režimu šifrování. [1]

Pokud *n* je velikost bloku algoritmu, potom *n*-bit OFB vypadá takto:

$$C_i = P_i \bullet S_i; S_i = E_K\left(S_{i-1}\right), \tag{3.8}$$

$$P_i = C_i \bullet S_i; S_i = E_K\left(S_{i-1}\right). \tag{3.9}$$

S_i je stav, který je nezávislý jak na vstupní zprávě tak na kryptogramu. [1]



Obr.3.5: Šifrování v režimu Output-Feedback.

OFB posuvný registr musí být také na začátku naplněn inicializačním vektorem, který by měl být alespoň unikátní, nemusí být tajný. [1]

3.7 Režim čítače (CTR)

Zkratka CTR je odvozena z anlg. counter (čítač). Tento režimu využívá jako vstup sekvenci čísel. Jako vstup do registru používá právě čítač namísto výstupu šifrování. Po zašifrování každého bloku se čítač inkrementuje, většinou o hodnotu 1. Režim CTR řeší problém režimu OFB, kdy výstupní blok má délku menší než *n*.

Není pevně dáno, že režim CTR musí všechny možné hodnoty vstupů načítat postupně, jako vstup do algoritmu se může použít generátor náhodných čísel. [1]

3.7.1 Proudové šifry v režimu CTR

Proudové šifry v režimu CTR mají jednoduchou funkci určující následující stav čítače a složitou funkci výstupu závislou na klíči. U proudové šifry v CTR režimu je možné vygenerovat libovolný *i*-tý bit klíče k_i, aniž by se musely generovat všechny předcházející bity klíče. Jednoduše se čítač nastaví na požadovanou hodnotu *i* a generuje se bit klíče. Tato možnost je velmi užitečná při dešifrování souboru s náhodným přístupem, takže je možné dešifrovat požadovaný blok souboru, aniž by se musel dešifrovat celý soubor. [1]

4 Šifrovací algoritmy

V této kapitole jsou popsány některé známé i méně známé šifrovací algoritmy, hlavním úkolem je přiblížit princip fungování jednotlivých algoritmů, u složitějších algoritmů (DES, GOST) jsou přiloženy i obrázky ilustrující průběh a pořadí prováděných operací, v závěru nechybí ani stručná zmínka o použití algoritmů v praxi.

4.1 Data Encryption Standard (DES)

DES je bloková šifra, do níž při šifrování vstupují 64-bitové bloky zprávy a vystupují 64-bitové bloky kryptogramu. Pro šifrování i pro dešifrování je u DES použit stejný algoritmus a klíč, tudíž hovoříme o symetrickém algoritmu.

Klíč je dlouhý 56 bitů, je ale obvykle zobrazen jako 64-bitové číslo, v němž je však každý osmý bit paritní a je vynechán. Klíč může být vyjádřen kterýmkoliv 56-bitovým číslem a může být kdykoliv změněn. Několik málo čísel je ale považováno za slabé klíče, kterým se dá ovšem lehce vyhnout. Veškerá bezpečnost spočívá uvnitř klíče [1].

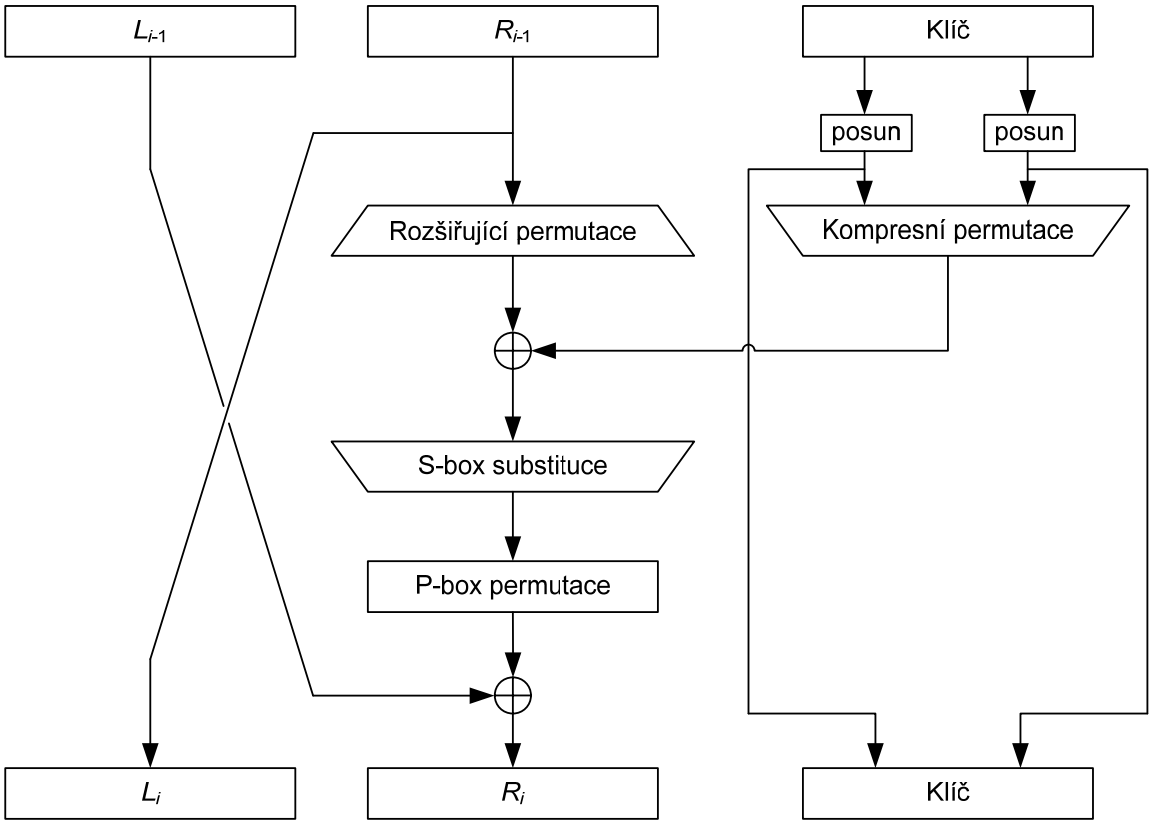
Z nejjednoduššího pohledu je DES kombinace dvou základních technik šifrování: záměny a rozptýlení. Základním stavebním blokem DESu je samostatná kombinace těchto dvou technik (substituce následovaná permutací) v textu, založená na klíči. Toto je známo jako runda. DES má 16 rund, tzn. na jeden blok zprávy aplikuje stejné kombinace technik 16-krát.[1]

Algoritmus využívá standardní výpočetní a logické operace a pracuje s maximálně 64-bitovými čísly, tudíž mohl být velice jednoduše implementován do hardwarové technologie pozdních sedmdesátých let 20. století [1]. Opakovací vlastnost algoritmu jej učinila ideálním pro použití u čipů na speciální účely [1]. Prvotní softwarové implementace byly těžkopádné oproti těm současným, které jsou již lepší.

4.1.1 Profil Algoritmu

Vstupní 64-bitový blok zprávy je po prvotní permutaci rozdělen na pravou a levou polovinu, každou o velikosti 32 bitů . Poté proběhne 16 rund identických operací, nazývaných Funkce *f*, ve kterých jsou data spojena s klíčem [1]. Po 16-té rundě se pravá a levá polovina spojí a závěrečná permutace (inverzní k té prvotní permutaci) zakončí algoritmus.

V každé rundě jsou bity klíče posunuty a poté je vybráno 48 bitů z 56-bitového klíče. Pravá polovina dat je rozšířena na 48 bitů pomocí rozšiřující permutace, dále je spojena se 48-bitovým klíčem, vzniklým posunutím a permutací pomocí XOR, pravá polovina je dále poslána skrz 8 S-boxů (substituční boxy, o nichž bude řeč později), vytvářejících 32 nových bitů, a znovu permutována. Tyto 4 operace tvoří Funkci *f*. Výstup Funkce *f* je poté spojen s levou polovinou pomocí dalšího XOR. Výsledek těchto operací se pak stává novou pravou polovinou, stará pravá polovina se stává novou levou polovinou. Tyto operace se opakují 16-krát a vytvářejí tak 16 rund algoritmu DES. [1] Uvedené operace ilustruje Obr. 4.1.



Obr.4.1: Runda algoritmu DES.

Jestliže B_i je výsledek i-té iterace, L_i a R_i jsou levé a pravé poloviny B_i, K_i je 48-bitový klíč i-té rundy a f je funkce, která provádí veškeré substituce a permutace a xorování s klíčem, poté jedna runda vypadá takto:

$$L_i = R_{i-1}, \tag{4.1}$$

$$R_i = L_{i-1} \bullet f\left(R_{i-1}, K_i\right). \text{ [1]} \tag{4.2}$$

4.1.2 Prvotní permutace

Počáteční permutace se provede ještě před první rundou samotné šifry. Přemisťuje bity vstupního souboru tak, že ve výsledku například 58-mý bit umístí na první pozici, na druhou pozici přemístí bit s pořadovým číslem o 8 menším, tj. 50, dále bit 42 zapíše do třetí pozice atd..

Počáteční permutace a k ní odpovídající závěrečná permutace neovlivňují bezpečnost DESu. Jelikož tento bitový způsob permutace je složitý v softwaru (ačkoliv triviální v hardwaru), mnoho softwarových implementací DESu vynechává počáteční i závěrečnou permutaci. [1]

4.1.3 Klíčová transformace

Nejdříve je 64-bitový klíč zmenšen na 56-bitový klíč vynecháním každého 8-mého bitu. Tyto bity mohou být použity pro kontrolu parity kvůli ujištění, že je klíč bezchybný. Poté, co

je 56-bitový klíč vytvořen, jiný 48-bitový „podklíč“ je vygenerován pro všech 16 jednotlivých rund DESu. Tyto podklíče *K*i jsou určovány následujícím způsobem.

56-bitový klíč je rozdělen na dvě poloviny o 28-mi bitech. Potom jsou tyto poloviny kruhově posunuty doleva buď o 1 nebo 2 bity, v závislosti na rundě. Poté, co proběhne posun, je vybráno 48 z 56 bitů. Protože tato operace permutuje pořadí bitů a také vybírá podskupinu bitů, je nazývána kompresní permutací (compression permutation). Tato operace produkuje podskupinu 48 bitů. Například bit na 33. pozici posunutého klíče je přesunut na 35. pozici výstupu, bit na pozici 18 je vynechán. Právě kvůli tomuto posouvání jsou použity různé podskupiny bitů v každém podklíči. Každý bit je použit přibližně ve 14-ti ze 16-ti podklíčů, ačkoliv ne všechny bity jsou použitý stejně často. [1]

4.1.4 Rozšiřující permutace

Při této operaci dochází k rozšíření pravé poloviny dat (R_i) z 32 bitů na 48 bitů. Jelikož tato operace mění pořadí bitů a také opakuje určité bity, je nazývána rozšiřující permutací (expansion permutation). Tato operace učiní pravou polovinu stejně velkou jako klíč pro XOR operaci a vytváří delší výsledek, který může být komprimován v průběhu substituce. Nicméně ani jedno z toho není hlavní kryptografický účel. Díky tomu, že jeden bit může ovlivnit 2 substituce, se závislost výstupních bitů na vstupních šíří rychleji. Tomuto se říká lavinový efekt. DES je navržen tak, aby byl každý bit kryptogramu ovlivněn každým bitem zprávy a klíčem jak nejrychleji to půjde.

Rozšiřující permutace je někdy nazývána jako E-box. Z každých 4 bitů vstupního bloku představují oba, první a čtvrtý bit, 2 bity výstupního bloku, zatímco druhé a třetí bity představují jeden bit výstupního bloku. Ačkoliv je výstupní blok větší než ten vstupní, každý výstupní blok je jedinečný. [1]

4.1.5 S-box substituce

Komprimovaný klíč je xorován s rozšířeným blokem a výsledek o velikosti 48 bitů je přesunut k substituční operaci. Substituce jsou vykonávány pomocí osmi substitučních boxů, takzvaných S-boxů, z nichž každý má 6-bitový vstup a 4-bitový výstup a každý z osmi S-boxů je jiný. Paměťová náročnost osmi S-boxů u DES je tedy 256 bytů.

Vstupní 48-bitový blok je rozdělen na osm 6-bitových podbloků, z nichž je každý substituován odděleně na jednom S-boxu. Prvního podbloku se ujme první S-box, druhý podblok je substituován druhým S-boxem, atd.

Každý S-box je vlastně tabulka o čtyřech řádcích a šestnácti sloupcích. Oněch 6 bitů na vstupu jen specifikuje, pod kterým řádkem a sloupcem hledat výstup.

První a šestý bit tvoří dvoubitové číslo, od 0 do 3, které určuje číslo řádku. Prostřední bity, tj.2, 3, 4 a 5, vytvoří čtyřbitové číslo určující sloupec. [1]

4.1.6 P-box permutace

32-bitový výstup S-box substituce permutován podle P-boxu. Tato permutace zakresluje každý vstupní bit do výstupní pozice. Žádný z bitů není použit vícenásobně ani není vynechán. Toto bývá nazýváno přímá permutace. Například bit 21 je přesunut na pozici 4,

zatímco bit 4 je posunut na pozici bitu 31. Nakonec je výsledek P-box permutace xorován s levou polovinou 64-bitového bloku. A po záměně levé a pravé poloviny může začít další runda. [1]

4.1.7 Závěrečná permutace

Jedná se o inverzní operaci k počáteční permutaci. Připomeňme si, že po poslední rundě DESu nejsou levá a pravá polovina ještě vyměněny. Jako vstup do závěrečné permutace je použit zřetězený blok $R_{16}L_{16}$. Stejný výsledek by byl dosažen i výměnou polovin a posunem. [1]

4.1.8 Dešifrování DES

Po všech substitucích, permutacích, xorováních a posunováních by se mohlo zdát, že dešifrovací algoritmus bude naprosto odlišný a stejně matoucí jako šifrovací. Opak je pravdou, rozmanité operace byly vybrány tak, aby zajistily velmi užitečnou vlastnost, že stejný algoritmus se dá použít jak pro šifrování tak pro dešifrování. [1]

4.1.9 Použití DES

Použití algoritmu DES je například šifrování zpráv v protokolu IPSec (Internet Protocol Security). Pro IPSec je DES standardizován v režimu CBC, kde je inicializační vektor přenášen jako součást paketu [2]. Dále se také DES používá k šifrování v protokolu SSL (Secure Societ Layer), který je určen k zajištění bezpečnosti informací v komunikačních sítích založených na přenosovém protokolu TCP (Transmission Control Protocol) [2]. V SSL je DES taktéž použit v CBC režimu.

4.1.10 Prolomení DES

Podle [4] byl algoritmus DES definitivně prolomen v červenci roku 1998, kdy Electronic Frontier Foundation[1] (dále jen EFF) oznámila, že DES byl prolomen za pomocí speciálního stroje postaveného za méně než $250 000. Útok přitom trval necelé tři dny. EFF zveřejnila detailní popis onoho stroje prolamovače, takže si mohl kdokoliv sestrojit svůj vlastní prolamovač. Podle dalšího zdroje ([5]) 56-bitový klíč algoritmu DES začínal být nedostačující z hlediska odolnosti proti útoku hrubou silou, kdy je zkoušen každý možný klíč dokud není nalezen ten správný a nedojde k úspěšnému průniku. Toto mělo za následek, že od listopadu roku 1998 již nebylo nadále povoleno používat algoritmus DES pro vládní účely USA a do nalezení vhodnějšího algoritmu byl namísto DES používán Triple-DES [5]. Princip algoritmu Triple-DES tkví v tom, že vstupní data jsou šifrována třikrát po sobě a to různými způsoby (se stejnými nebo jinými klíči pro každé ze tří šifrování) [5].

[1] **Electronic Frontier Foundation** (**EFF**) je mezinárodní nezisková advokátní a zákonná organizace založena ve Spojených Státech za účelem ochraňování práva svobody projevu ve smyslu dnešního digitálního věku. Jejím hlavním cílem je poučovat tisk, politiky a veřejnost o záležitostech občanského práva spojeného s technologií a v tomto směru toto právo i bránit. Více informací na http://www.eff.org.

4.2 IDEA

Zkratka IDEA vznikla složením slov International Data Encryption Algorithm. Tento algoritmus je postaven na velmi působivých teoretických poznatcích a i přesto, že kryptoanalytici udělali velký pokrok při prolamování variant algoritmu IDEA s omezeným počtem rund, stále se jeví jako silný. [1]

IDEA je bloková šifra operující s 64-bitovými bloky vstupní zprávy. Klíč je 128 bitů dlouhý a pro šifrování i dešifrování je použit totožný algoritmus. Tak jako některé jiné blokové šifry IDEA používá techniky záměnu a rozptýlení. Byly skombinovány tři algebraické skupiny, které jsou jednoduše implementovatelné jak v softwaru tak v hardwaru. Jsou to XOR, sčítání modulo 2^{16} a násobení modulo $2^{16} + 1$. [1]

Uvedené operace jsou kompletním výčtem operací použitých v algoritmu IDEA. Nejsou zde použity žádné bitové permutace [1]. Všechny operace pracují s 16-bitovými subbloky, takže algoritmus je efektivní na 16-bitových mikroprocesorech.

4.2.1 Popis algoritmu

64-bitový blok je rozdělen na čtyři 16-bitové subbloky X_1, X_2, X_3 a X_4, které se přivedou na vstup první rundy algoritmu. Celkem proběhne osm rund. Během jedné rundy jsou ony čtyři vstupní subbloky xorovány, sčítány a násobeny mezi sebou a taky s klíčem. Mezi jednotlivými rundami se druhý a třetí subblok mezi sebou prohodí. Podrobný sled operací v jedné rundě je podle [1] popsán následovně:

1) Násobení X_1 s prvním subklíčem.
2) Sečtení X_2 a druhého subklíče.
3) Sečtení X_3 a třetího subklíče.
4) Násobení X_4 se čtvrtým subklíčem.
5) Xorování výsledků kroků 1) a 3).
6) Xorování výsledků kroků 2) a 4).
7) Násobení výsledku kroku 5) s pátým subklíčem.
8) Sečtení výsledků kroků 6) a 7).
9) Násobení výsledku kroku 8) se šestým subklíčem.
10) Sečtení výsledků kroků 7) a 9).
11) Xorování výsledků kroků 1) a 9).
12) Xorování výsledků kroků 3) a 9).
13) Xorování výsledků kroků 2) a 10).
14) Xorování výsledků kroků 4) a 10).

Výstupem z rundy jsou čtyři subbloky vzniklé z kroků 11), 12), 13) a 14). Dojde k již zmíněnému prohození druhého a třetího subbloku (prohození se neprovádí po poslední rundě) a máme vstup pro další rundu. [1]

Po osmé rundě se provede výstupní transformace takto:

Násobení X_1 s prvním subklíčem.

Sečtení X_2 a druhého subklíče.

Sečtení X_3 a třetího subklíče.

Násobení X_4 se čtvrtým subklíčem. [1]

Tvorba subklíčů probíhá také jednoduše. Algoritmus používá 52 subklíčů, 6 pro každou z osmi rund a ještě 4 další pro výstupní transformaci. Nejdříve je 128-bitový klíč rozdělen na osm 16-bitových subklíčů. Jedná se o prvních osm subklíčů algoritmu (6 pro první rundu a první dva pro druhou rundu). Potom je klíč rotován o 25 bitů doleva a znovu rozdělen na osm subklíčů. První čtyři jsou použity v druhé rundě a zbývající čtyři se použijí v rundě třetí. Klíč je znovu rotován o 25 bitů doleva a takto to probíhá až do konce algoritmu. [1]

4.2.2 Použití algoritmu IDEA

Na závěr ještě dodám, že tato šifra se v CBC režimu používá jako jedna z vícero možností k šifrování zpráv v protokolu SSL (Secure Societ Layer), který zajišťuje autentičnost a důvěrnost přenášených informací [2].

4.3 GOST

Zkratka GOST může být chápána dvěma způsoby. Za prvé se jedná o celý souhrn sovětských vládních standardů ("Gosudarstvennyi Standard", nebo-li Vládní Standard (Government Standard) [1]), které ani nemusejí mít cokoliv společného s kryptografií. Celé jméno zní Gosudarstvennyi Standard Soyuza SSR nebo Government Standard of the Union of Soviet Socialist Republice [1]. Po rozpadu SSSR získaly standardy GOST status regionálních standardů a jsou spravovány Euroasijskou radou pro normalizaci, metrologii a certifikaci (Euro-Asian Council for Standardization, Metrology and Certification neboli EASC). A za druhé je zkratka GOST chápána jako bloková šifra z bývalého Sovětského svazu. Tato šifra byla roku 1989 schválena vládní komisí pro standardy v SSSR a to pod označením standardu GOST 28147-89 [1].

Protože tato práce pojednává o kryptografii, bude v ní zkratka „GOST" výhradně představovat zmíněný standard GOST 28147-89.

Tento algoritmus uspokojuje všechny kryptografické požadavky a nelimituje stupeň informací, které mají být chráněny. Údajně měl být GOST původně použit pro vysoce kvalitní komunikaci, včetně tajné armádní komunikace, ale toto nebylo potvrzeno. [1]

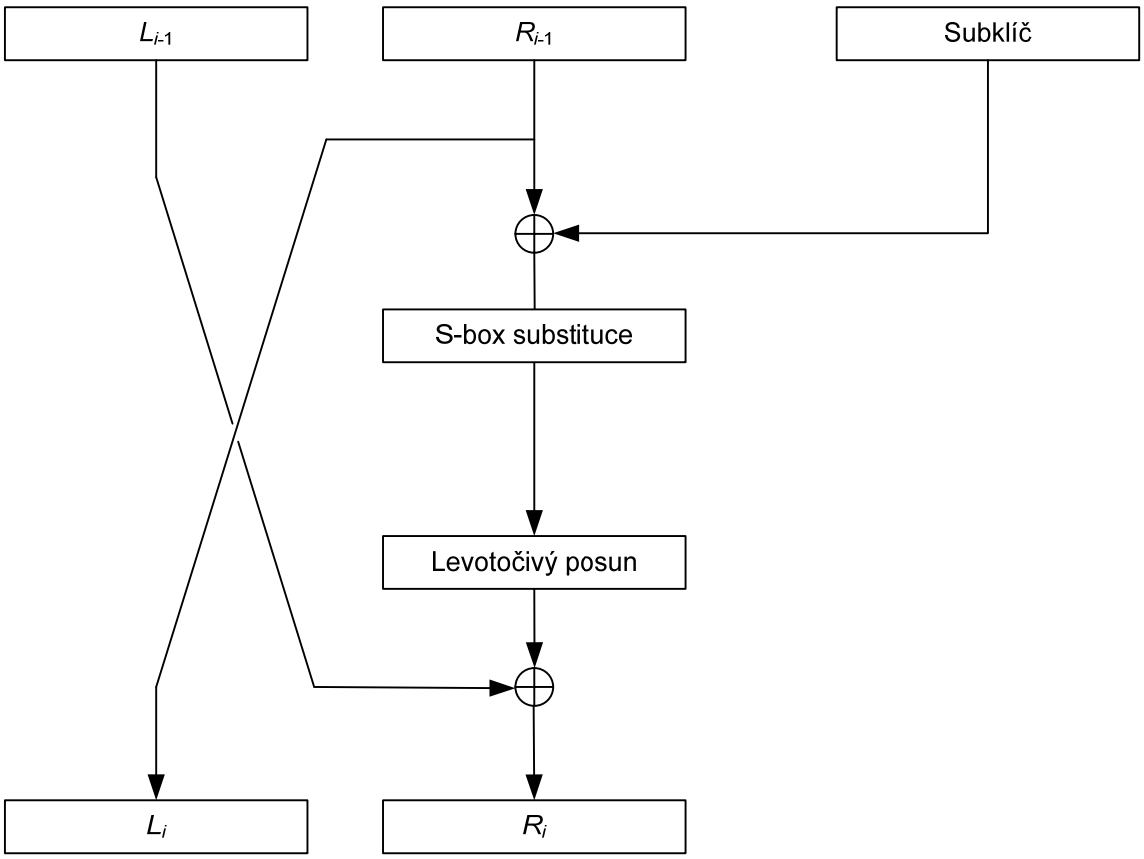
4.3.1 Popis algoritmu

GOST je 64-bitový blokový šifrovací algoritmus s 256-bitovým klíčem. Algoritmus opakuje jednoduchý šifrovací algoritmus ve 32 rundách. Před šifrováním se musí nejdříve vstupní zpráva rozdělit na dvě poloviny, levou L a pravou R. Subklíč pro rundu i je K_i. Jedna runda GOSTu vypadá podle [1] takto:

$$L_i = R_{i-1},\ R_i = L_{i-1} \bullet f\left(R_{i-1}, K_i\right). \tag{4.3}$$

Jednu rundu algoritmu GOST popisuje Obr. 4.2. Funkce *f* je přímočará. Nejdříve je pravá polovina s *i*-tým subklíčem sečtena modulo 2^{32}. potom je výsledek této operace rozdělen na osm 4-bitových kusů, z nichž se každý stává vstupem do jednoho z osmi odlišných S-boxů, které GOST obsahuje. První čtveřice bitů vstupuje do prvního S-boxu, druhá čtveřice do druhého S-boxu, atd. Každý S-box představuje permutaci čísel od 0 do 15. Takže S-box může například vypadat takto:
7, 10, 2, 4, 15, 9, 0, 3, 6, 12, 5, 13, 1, 8, 11. [1]

Jestliže je v tomto případě vstupem do S-boxu 0, výstup je 7. Jestliže je vstup 1, výstup je 10, atd. Všech 8 S-boxů je jiných, jsou považovány za doplňující klíče. S-boxy je nutno držet v tajnosti. [1]



Obr.4.2: Runda algoritmu GOST.

Výstupy všech osmi S-boxů jsou smíšeny do 32-bitového slova, které potom celé podstoupí 11-bitový levotočivý posun. Výsledek předchozí operace je xorován s levou polovinou vstupu a vytvoří se tak nová pravá polovina, pravá polovina vstupu se pak stane novou levou polovinou. GOST toto provádí 32-krát. [1] Po provedení poslední rundy je potřeba, jako v každé jiné blokové šifře využívající Feistelovo schéma, prohodit obě výstupní poloviny. [2]

Subklíče jsou tvořeny jednoduše. 256-bitový klíč je rozdělen na osm 32-bitových subklíčů: k_1, k_2,..., k_8. Každá runda používá jiný subklíč. Dešifrování je stejné jako šifrování s obráceným pořadím subklíčů k_i. [1]

GOST standard nepojednává o tvorbě S-boxů, pouze informuje, že jsou nějakým způsobem zásobené. V pozdější době výrobce GOST čipů v Rusku vytvořil S-box permutace užitím generátoru náhodných čísel. [1]

4.3.2 Hlavní rozdíly mezi DES a GOST

- DES má komplikovanější proceduru generování subklíčů. GOST má velmi jednoduchou proceduru.
- DES má 56-bitový klíč, GOST má 256-bitový klíč.
- S-boxy v DESu mají 6-bitové vstupy a 4-bitové výstupy, kdežto GOST má S-boxy se 4-bitovými vstupy i výstupy. Oba algoritmy ale mají shodný počet S-boxů – osm.
- DES používá nepravidelnou permutaci, nazývanou P-box, GOST oproti tomu má 11-bitový levotočivý posun.
- DES má 16 rund a GOST jich má 32. [1]

4.4 A5

A5 je proudová šifra používaná pro šifrování v systému GSM. Šifrováno je spojení mezi telefonem a základnovou stanicí. Zbytek spojení zašifrován není. [1]

A5 používá registry LFSR (Linear Feedback Shift Register), jejichž princip je vysvětlen v následujících odstavcích ještě před samotným popisem fungování šifry A5.

4.4.1 LFSR

Jedná se o variantu generátoru pseudonáhodné posloupnosti používaného pro generování heslových posloupností. Je to lineární generátor s posuvným registrem a s lineární zpětnou vazbou.

Generátor s posuvným registrem a se zpětnou vazbou se skládá ze dvou částí: posuvný registr a funkce zpětné vazby. Posuvný registr je sekvence bitů. Délka posuvného registru se udává v bitech (například 3-bitový posuvný registr). Vždy když je nějaký bit spotřebován, všechny ostatní bity v registru jsou posunuty o jeden bit doprava. A na volné místo úplně vlevo je zapsán bit vzniklý nějakou funkcí z ostatních bitů v registru. Perioda posuvného registru je definována jako délka výstupní sekvence bitů před tím, než se začne opakovat. [1]

Posuvné registry jsou v kryptografii oblíbené, neboť jsou jednoduše hardwarově implementovatelné. [1]

Nejjednodušším typem generátoru s posuvným registrem a se zpětnou vazbou je lineární generátor s posuvným registrem a lineární zpětnou vazbou (LFSR). Funkcí zpětné vazby je zde xorování určitých bitů v registru. [1]

Princip LFSR je podrobněji vysvětlen v [2].

4.4.2 Popis A5

Generátor LFSR je z hlediska kryptoanalýzy málo odolný. V praxi se proto několik generátorů LFSR nelineárně kombinuje. [2]

A5 se skládá z tří registrů LFSR G_1, G_2 a G_3 o délkách 19, 22 a 23 bitů. Všechny generátory mají pevně nastavenou zpětnou vazbu a tak klíčem K je v tomto případě počáteční nastavení buněk. Klíč je tedy 19 + 22 + 23 = 64 bitů dlouhý. Důležitou součástí zdroje hesla je blok logiky. Tento blok sleduje hodnoty prostředních bitů generátorů G_1 až G_3 a na jejich základě řídí změnu stavů těchto generátorů. K této změně dochází u toho generátoru, jehož prostřední bit je shodný s prostředním bitem alespoň jednoho dalšího generátoru. Bit hesla vznikne sečtením modulo 2 hodnot posledních buněk (bitů) všech tří generátorů. [2]

Jiným stylem je vysvětleno fungování registrů LFSR u šifry A5 v [6]:

Ona nelinearita je do šifry vnesena nelineární funkcí C, která určuje, zda se příslušný registr posune („go“), či zůstane stát („stop“). Proto registry LFSR bývají nazývány „stop and go“ LFSR nebo obecněji clock-controlled LFSR. Do funkce C vstupuje prostřední bit každého registru a vystupují z ní tři bity (c1, c2 a c3), které určují pohyb odpovídajících registrů. Funkce je tvořena tak, aby se vždy posunuly alespoň dva registry. Jsou to ty, jejichž řídící buňky mají stejnou hodnotu, zatímco registr s odlišnou hodnotou řídící buňky stojí. V případě, že jsou všechny tři řídící bity shodné, posunují se podle tohoto pravidla všechny tři registry. Postup je pak následující: nejprve se přečtou řídící bity registrů, na základě hodnot c1, c2 a c3 pak dojde k posunu dvou nebo tří odpovídajících registrů (a vytvoření nových hodnot bitů lsb-least significant bit) a ze tří nově vytvořených bitů msb-most significant bit je vytvořena hodnota hesla. Základem bezpečnosti je mimo jiné neznámé počáteční nastavení registrů. [6]

4.4.3 Bezpečnost A5

Jediná známá slabost tohoto systému je ta, že záznamy jsou natolik krátké, že umožňují naprosté prozkoumání systému. Varianty A5 s delší záměnou systému a hustší zpětnou vazbou mnohočlenů by měli být bezpečné. [1]

4.5 RC4

RC4 je proudová šifra s proměnnou velikostí klíče. Byla vynalezena v roce 1987. Je velice jednoduchá. Algoritmus pracuje v režimu OFB. Proud klíče je nezávislý na vstupní zprávě. Obsahuje 8x8 S-boxů, $S_{0\text{-}255}$. Vstupy jsou permutacemi čísel od 0 do 255 a permutace je funkce klíče s proměnnou délkou. Má dvě počitadla, i a j, inicializované na 0. [1]

4.5.1 Popis RC4

Pro vygenerování náhodného bytu se provede:

$$i = (i + 1) \bmod 256, \tag{4.4}$$

$$j = (j + S_i) \bmod 256, \tag{4.5}$$

prohození S_i a S_j,

$$t = (S_i + S_j) \bmod 256, \tag{4.6}$$

$$K = S_t. \text{ [1]} \tag{4.7}$$

Byte K je xorován se vstupní zprávou, aby vznikl kryptogram, nebo je xorován s kryptogramem, aby vznikla zpráva. Šifrování RC4 je cca 10-krát rychlejší než šifrování pomocí DES. [1]

Inicializace S-boxu je také jednoduchá. Nejdříve se naplní lineárně: S_0=0, S_1=1…S_{255}=255. Poté se další 256-bytové pole naplní klíčem a to tak, že klíč se v poli opakuje nezbytně-krát, aby bylo v poli zcela zaplněno všech 256 bytů. Index j se nastaví na 0 a provede se následující cyklus:

pro i = 0 až 255:

$$j = (j + S_i + K_j) \bmod 256, \tag{4.8}$$

prohození S_i a S_j

a je hotovo. [1]

4.5.2 Použití RC4

Tato šifra je použita v protokolu WEP (Wired Equivalent Privacy), který zajišťuje bezpečnost přenosu paketů na úrovni linkové vrstvy v první verzi standardu 802.11 pro bezdrátové lokální sítě WLAN. Šifrou RC4 je každý paket šifrován na základě 64-bitového klíče $KP=IV||K$. Náhodný inicializační IV vektor o délce 24 bitů se generuje pro každý paket a tajný parametr K o délce 40 bit je klíčem daného přístupového bodu sítě WLAN. [2]

5 Implementace šifrovacích algoritmů

Obecně je očekáváno, že hardwarová implementace je rychlejší než softwarová, zpracování dat se dá totiž paralelizovat. Moderní technologie tak umožní až gigabytové procedury. Na druhé straně jsou ale hardwarové šifry neflexibilní a v případě, že se časem odhalí nějaké slabiny, update bude velmi nákladný. [8] V této kapitole je porovnána hardwarová implementace se softwarovou a to hlavně z praktického úhlu pohledu. Pro návrh hardwarové implementace algoritmů jsou popsány některé programovací jazyky, jejich vývoj, výhody a nevýhody, je uveden i jeden moderní návrhový systém, který značně ulehčuje „překlad“ systémů navrhnutých v jazyce C/C++ do jazyku pro popis hardware.

5.1 Hardwarové šifrování versus softwarové šifrování

5.1.1 Hardware

Většina produktů zajištující šifrování se donedávna vyskytovala jen jako specializovaný hardware ve formě „krabiček“ zapojených do komunikačního kanálu a šifrujících veškerá data proudící tímto kanálem. Ačkoli softwarové šifrování se od té doby značně rozšířilo, hardware je nadále s výhodou využíván v armádě a v důležitých komerčních aplikacích [1]. Existuje pro to několik důvodů.

Prvním z nich je rychlost. Jak mohlo být vidět v kapitole 4, šifrovací algoritmy sestávají z různých druhů komplikovaných operací, které přemění vstupní zprávu na výsledný kryptogram. Takové operace nejsou zabudovány do běžných počítačů (resp. jejich procesorů), na kterých potom tyto algoritmy běží neefektivně a tudíž pomalu. Ačkoli snahy kryptografů, aby jejich algoritmy byly co nejlépe použitelné pro softwarovou implementaci, byly veliké, specializovaný hardware vždy zvítězil na poli rychlosti šifrování [1].

Druhým důvodem ve prospěch hardwarové implementace je bezpečnost. Šifrovací algoritmus běžící na obyčejném počítači nemá žádnou fyzickou ochranu. Útočník může nerušeně algoritmus například pozměnit za pomocí debugovacích nástrojů a nikdo si toho nikdy nevšimne. Oproti tomu hardwarové šifrovací zařízení je možné bezpečně opouzdřit a předejít tak případné snaze o modifikaci algoritmu. Například se použijí boxy odolné proti nepovolené manipulaci nebo čipy uvnitř tohoto boxu mohou být potřeny nějakou chemickou látkou, která v případě jakéhokoliv pokusu o průnik do boxu způsobí destrukci čipu [1]. Elektronické zařízení vysílá do svého okolí elektromagnetické záření, které může prozradit, co se uvnitř onoho zařízení děje, proto jsou některé hardwarové šifrovací moduly odstíněny, aby z nich citlivé informace neprosakovaly [1]. Samozřejmě, běžné počítače mohou být také odstíněny, jenže to je mnohem složitější záležitost, než odstínit malou krabičku.

A posledním důvodem, proč zvolit hardwarovou implementaci, je jednoduchost instalace. Většina aplikací využívající šifrování nezahrnuje do svého pole působnosti běžné počítače [1], nýbrž se šifrují telefonní hovory, faxové přenosy, předplacené televizní vysílání, emailová korespondence a vůbec veškerá citlivá a utajovaná komunikace prostřednictvím

internetu, interní firemní telefonní a datové linky atd. V dnešní době si sice běžný a průměrně zkušený uživatel počítače může svá důležitá data bezpečně zašifrovat pomocí dobře dostupných freewarových šifrovacích programů, ale jedná se o velice malé procento využití kryptografie v porovnání s hardwarovými implementacemi šifrování.

Existuje několik základních typů hardwarových šifrovacích zařízení. Takové, co zajišťují ověřování hesel, bezpečnou distribuci klíčů v bankovnictví (samostatné šifrovací moduly) [1]. Potom jednoúčelové šifrovací boxy pro komunikační linky a nakonec desky zapojitelné do osobního počítače [1]. Některé boxy jsou navrženy pro určitý druh komunikační linky, například T-1 šifrovací boxy nešifrují synchronizační bity [1]. Používají se rozdílné boxy pro synchronní a asynchronní linky. Novější boxy dokáží pojmout i vyšší bitový tok a jsou přizpůsobivější. Tato jednoúčelová zařízení mají ovšem i své nevýhody, které pramení z toho pojmenování „jednoúčelová“. Pokud si chce zákazník takové zařízení pro nějaký konkrétní účel pořídit, musí velmi pečlivě zkonfrontovat omezení pro provoz takového zařízení, jako je podporovaný hardware počítače, operační systém, síťové a softwarové aplikace atd. [1]. PC-deskové šifrátory většinou šifrují veškerá data, která jsou zapisována na harddisk [1]. Tyto desky ale nejsou stíněné od elektromagnetického záření, což není nejbezpečnější [1].

5.1.2 Software

Každý šifrovací algoritmus je možné implementovat softwarově. Nevýhody jsou ovšem v rychlosti a ceně. Výhody jsou flexibilita, lehkost modifikace, jednoduchost použití a snadný upgrade. Softwarové šifrovací programy jsou oblíbené a dostupné pro většinu operačních systémů. Slouží jako ochrana individuálních dat uživatele, který si je sám zašifruje a dešifruje. Je důležité, aby bylo zajištěno bezpečné nakládání s klíči, které by neměly být nechráněně uloženy na disku. Po zašifrování by měly být klíče a nezašifrované soubory smazány, na což mnoho programů nedbá a uživatel by měl v být v tomto směru velmi obezřetný. [1]

Samozřejmě může případný útočník proniknout k uživatelově počítači, zaměnit šifrovací algoritmus za nějaký jiný mizerný nebo může modifikovat šifrovací program [1]. Takže by bylo bezpečnější použít šifrovací box, který bude odstíněný a ještě dalšími bezpečnostními prvky vybavený, aby měl uživatel větší jistotu, že mu jeho data nikdo neukradne.

5.2 Programovací jazyky pro popis hardware

5.2.1 VHDL

Jazyk VHDL je jazyk vysoké úrovně navržený speciálně pro účely popisu (návrhu) a simulace velmi rozsáhlých číslicových obvodů a systémů. Jedná se tedy o jazyk určený pro popis hardware [9]. Jazyky pro popis hardware jsou označovány zkratkou HDL (Hardware Description Language). Výhodou tohoto jazyka jsou jeho bohaté vyjadřovací schopnosti a značná nezávislost číslicového systému popsaného jazykem VHDL na cílové technologii jeho realizace nebo výroby [9]. O rozmanitosti jazyka VHDL svědčí i fakt, že jeho úplný výklad čítá cca 300 stran.

Je třeba zdůraznit, že popis číslicového systému (hardware) v jazyce VHDL je činnost značně odlišná od programování v klasických programovacích jazycích typu C nebo Pascal

[9]. Při popisu číslicového systému jazykem VHDL musíme myslet na to, že vytvořený kód bude muset projít syntézou, jejímž výsledkem bude zapojení z hradel a klopných obvodů určené pro programovatelný logický. Konstrukce vytvořené v jazyce VHDL při popisu navrhovaného číslicového systému musí tedy být syntetizovatelné (výjimkou je jedině testovací program nebo model určený pouze pro simulaci na simulátoru). Testovací program (angl. testbench) je program v jazyce VHDL, jímž generujeme vstupní signály pro námi navrhovaný číslicový systém a kontrolujeme správnost jeho funkce [9]. Velká část příkazů, vyjadřovacích prostředků a vymožeností jazyka VHDL nelze nebo prozatím nelze syntetizovat do hardware, ale lze pouze simulovat na simulátoru. Také je třeba podotknout, že syntezátory jednotlivých výrobců se drobně liší v tom, co umí syntetizovat, což se projevuje při syntéze složitějších konstruktů jazyka VHDL. Obecně však lze říci, že schopnosti syntezátorů se neustále zvětšují a zvládají již syntézu poměrně složitých konstrukcí jazyka VHDL [9].

5.2.2 Historie, současnost, budoucnost a vlastnosti jazyka VHDL

Vývoj jazyka VHDL byl zahájen v roce 1981 v rámci výzkumného projektu VHSIC (z anlg. Very High Speed Integrated Circuits) ministerstva obrany Spojených států amerických [9]. V rámci projektu byl řešen mimo jiné problém efektivního popisu velmi rozsáhlých integrovaných obvodů, neboť vývoj a simulace navrhovaných obvodů byl tehdy řešen pomocí různých vzájemně nekompatibilních jazyků a nástrojů [9]. Jedním z cílů výzkumného projektu bylo navrhnout jazyk vysoké úrovně s rozsáhlými vyjadřovacími schopnostmi umožňující simulaci a návrh číslicových obvodů nezávisle na cílové technologii.

Vlastní vývoj základů jazyka byl v roce 1983 zadán firmám, které se výzkumného projektu VHSIC účastnily (IBM, Intermetrics a Texas Instruments). Základ definice jazyka VHDL byl poprvé veřejně publikován roku 1985. Označení VHDL vzniklo jako akronym z názvu VHSIC Hardware Description Language. Pro další rozvoj, vývoj a standardizaci jazyka postoupilo ministerstvo obrany jazyk VHDL v roce 1986 organizaci IEEE (angl. Institute of Electrical and Electronics Engineers). V roce 1987 byl organizací IEEE poprvé publikován standard jazyka VHDL (IEEE Std 1076-1987). Standard jazyka VHDL od svého prvního vydání prošel zatím celkem třemi revizemi a jedním rozšířením. V současné době jsou ve všech návrhových a simulačních systémech všech výrobců podporovány první dvě verze standardu (1987 a 1993), jelikož druhá a třetí revize (2000 a 2002) nepřinesla mnoho významných změn, nesetkala se tak s odezvou ze strany firem. Díky tomu se v současnosti používá pro návrh a simulaci v jazyce VHDL většinou standard IEEE Std 1076-1993. [9]

5.2.3 Další HDL jazyky

Kromě jazyka VHDL existují samozřejmě i další jazyky pro popis číslicových systémů. Jmenujme proto další dva HDL jazyky standardizované IEEE, a to Verilog a SystemVerilog. Standard jazyka Verilog byl poprvé publikován v roce 1995 pod označením IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language (IEEE Std 1364-1995), v roce 2001 pak byla publikována jeho první revize. Jazyk Verilog má syntaxi podobnou jazyku C a má také podobné vyjadřovací schopnosti jako má jazyk VHDL. Verilog je společně s VHDL dnes používán k návrhu většiny rozsáhlých číslicových systémů (např. procesorů, čipsetů, programovatelných logických obvodů, aj.). Mnoho dnešních

návrhových systémů a simulačních systémů se tak dnes může skládat z částí systému napsaných v jazyce Verilog i VHDL. Jen pro zajímavost používání jazyka VHDL a Verilog je také geograficky odlišné. Používání jazyka VHDL je rozšířeno zejména na univerzitách a firmách v Evropě, zatímco Verilog je rozšířen zejména v Asijských zemích. V USA je v univerzitním i firemním prostředí rozšířeno používání obou jazyků. Dalším výše zmíněným jazykem je SystemVerilog, který je založen na jazyce Verilog a silně rozšířil možnosti původního jazyka. Standard jazyka SystemVerilog byl publikován v roce 2005 pod označením IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language (IEEE Std 1800-2005). [9]

5.2.4 Ne-HDL jazyky

V současné době se velmi rozvíjejí jazyky určené pro popis číslicových systémů, které jsou založené přímo na jazyce C nebo C++. Mezi tyto jazyky patří například Handel-C a SystemC. Jazyk handel-C je založen na jazyce C, jazyk SystemC pak na objektovém jazyce C++. Standard jazyka SystemC byl publikován v roce 2006 pod označením IEEE Standard SystemC Language Reference Manual (IEEE Std 1666-2005). Tyto jazyky umožňují buď přímou syntézu číslicového systému navrženého v těchto jazycích do cílové technologie, nebo syntézu založenou na překladu do RTL (Register Transfer Level) popisu systému v jazycích VHDL nebo Verilog a následnou syntézu do cílové technologie. Tyto nové jazyky byly vytvořeny z důvodu dosažení ještě větší abstrakce od hardware, než jaké dosahují dnes již „klasické“ jazyky pro popis hardware jako je VHDL nebo Verilog. Tyto nové jazyky již tedy nejsou koncipovány pro přímý popis hardware, nejedná se tedy o HDL jazyky, ale jedná se spíše o jazyky vysoké úrovně, jejichž výstupem je hardware. Návrhář číslicového systému v těchto nových jazycích tak již nemusí klást takovou pozornost a důraz na to, jak bude syntézu jednotlivých sekvencí kódu implementovat na úrovni hradel a klopných obvodů, a může se mnohem více věnovat vývoji a optimalizaci návrhu systému na algoritmické úrovni. Optimalizace navrhovaného systému na úrovni hradel, klopných obvodů a podobně je v těchto jazycích již hodně skryta a ponechána na kompilátoru a syntezátoru těchto jazyků. Je vidět, že vývoj v oblasti jazyků pro popis číslicového systému se velmi rozvíjí a jeho tempo i úroveň abstrakce se bude v budoucnu zřejmě dále zvětšovat. [9]

Výše uvedené jazyky se samozřejmě mezi sebou navzájem liší vyjadřovacími schopnostmi, úrovní abstrakce, striktní typovostí nebo naopak typovou tolerancí, úsporností nebo naopak rozvláčností zápisu, mají však mnoho společných vlastností, z nichž ty nejdůležitější jsou:

- Číslicové systémy navržené v těchto jazycích jsou značně nezávislé na cílové implementaci v reálném hardware a lze je tak implementovat v programovatelných obvodech různých typových řad od různých výrobců. V případě potřeby výroby velkých sérií lze takto navržený sytém použít i pro implementaci plně zákaznického obvodu ASIC.
- Uvedené jazyky umožňují simulovat vytvořený číslicový systém na simulátorech příslušných jazyků a také v těchto jazycích psát vlastní testovací prostředí pro navrhovaný číslicový systém.

- Většina uvedených jazyků byla publikována formou otevřeného IEEE standardu. Nejedná se tedy o uzavřené proprietární firemní jazyky, jejichž využití jinými firmami by tím bylo značně ztíženě. Díky tomu je možné používat pro návrh v těchto jazycích simulátory, syntezátory, případně celá vývojová prostředí různých výrobců. Výsledkem toho je také vznik konkurenčních prostředí na poli výrobců těchto návrhových nástrojů.
- Některé jazyky lze používat pro návrh, simulaci a implementaci číslicového systému společně a využít tak naplno možností a vhodnosti použití jednotlivých jazyků pro jednotlivé části navrhovaného systému.

  [9]

A díky těmto zmíněným vlastnostem se v této práci budu zabývat pouze návrhem šifrovacího algoritmu v jednom z jazyků pro popis hardware a výběr konkrétní technologie, v níž by se můj návrh měl případně realizovat, ponechám otevřený.

5.2.5 SystemC a SystemCrafter

Za zmínku stojí ještě další programovací jazyky pro popis hardware. Jsou jimi SystemC a SystemCrafter, které vycházejí z jazyků C a C++, jenž byly oblíbenými pro vývoj hardwaru a systémů po mnoho let.

Znalost jazyků C a C++je široce rozšířena, píše se v nich velice rychle a poskytují proveditelnou specifikaci, která umožňuje rychlou simulaci. Standardní algoritmy ve verzích C či C++ jsou lehce dostupné z legálních veřejných zdrojů, takže je možné tyto kódy algoritmů jednoduše znovu použít. Pro návrh na úrovni systému umožňují jazyky C a C++ popsat hardware či software v jediné struktuře. Nicméně existují jisté dvě nevýhody. Za prvé, C a C++ nepodporují popis některých důležitých hardwarových pojmů jako je časování a souběžnost. Toto vedlo k vyvinutí patentovaných rádoby-C jazyků, které nejsou oblíbené pro jejich uživatelskou vázanost k jedinému softwarovému dodavateli. Za druhé, pro implementaci C a C++ do hardwaru je třeba kódy v těchto jazycích manuálně přepsat do HDL jazyků jako například VHDL či Verilog. Tento časově náročný krok vyžaduje odbornost hardwarového experta a často vnáší do kódu chyby, které se jen stěží hledají. První problém byl vyřešen díky vyvinutí jazyka SystemC, který je velmi rozšířeným průmyslovým standardem doplňujícím jazyk C++ o chybějící hardwarové prvky. Druhý problém byl vyřešen vývojem nástrojů jako je například SystemCrafter SC, který automaticky překládá popis hardware z jazyka SystemC do jazyka VHDL. [10]

a) SystemC

SystemC poskytuje a zajišťuje průmyslový standard modelování a verifikace hardware a systémů za použití standardních kompilátorů. Veškeré materiály potřebné k simulaci SystemC na standardních C++ kompilátorech, jako Microsoft Visual C++ nebo GNU GCC, jsou zdarma ke stáhnutí na stránce SystemC (www.systemc.org). SystemC obsahuje sadu knihoven pro C++ popisující hardwarové konstrukce a koncepce. Znamená to, že je možné vyvíjet cyklicky přesné modely hardwaru, softwaru a rozhraní pro simulaci a debugging

uvnitř existujících C++ vývojových prostředích. SystemC dokáže vykonat počáteční návrh, debugging a doladění za použití stejných testbenchů, což eliminuje chyby vzniklé překladem a zajišťuje rychlou a snadnou verifikaci. Veškeré výhody jazyka C++ využívané softwarovými inženýry po mnoho let jsou tak nyní k dispozici i hardwarovým a systémovým návrhářům. SystemC je kompaktnější než VHDL nebo Verilog, rychleji se v něm píše a je lépe udržitelný a snáze čitelný. [10]

b) SystemCrafter SC

SystemC byl vyvinut jako jazyk pro modelování a verifikaci, pro výrobu hardware je ještě stále potřeba ručního překladu do HDL jazyka. Tento proces je automatizován právě pomocí SystemCrafter SC, který rychle syntetizuje SystemC do RTL jazyka VHDL. Také umožňuje generovat popis syntetizovaného obvodu v jazyce SystemC, což dovoluje ověřit syntetizovaný kód s existujícím testovacím vybavením. SystemCrafter SC tedy umožňuje syntetizovat SystemC do hardwaru, návrh na úrovni systému a vlastní návrh výroby a hardwarové akcelerace FPGA. SystemCrafter SC dává jeho uživateli kontrolu nad kritickými fázemi postupu. A tak jsou výsledky vždy předvídatelné, kontrolovatelné a budou se shodovat s očekáváními uživatele. Umožňuje vývoj, doladění, debug a syntézu hardwaru a systémů uvnitř uživatelova vývojového prostředí existujícího C++ kompilátoru. Je možno velice rychle provést verifikaci návrhu proveditelným SystemC kódem.

Kód VHDL vytvořený pomocí SystemCrafter je jádrem implementace. Vytvořený projekt je kompilován ve standardním nástroji některého z výrobců programovatelných logických obvodů (například nástrojem XST od Xilinx) a je vygenerován konfigurační soubor pro FPGA. SystemCrafter nahlíží na vývoj hardwaru jako na činnost vyšší úrovně než je psaní v HDL a umožňuje se zaměřit na algoritmickou stránku návrhu než na detaily implementace. Může to snížit riziko při návrhu, ušetřit čas a návrháři umožnit navrhnout komplexní systém aniž by potřeboval znalosti HDL či elektroniky.

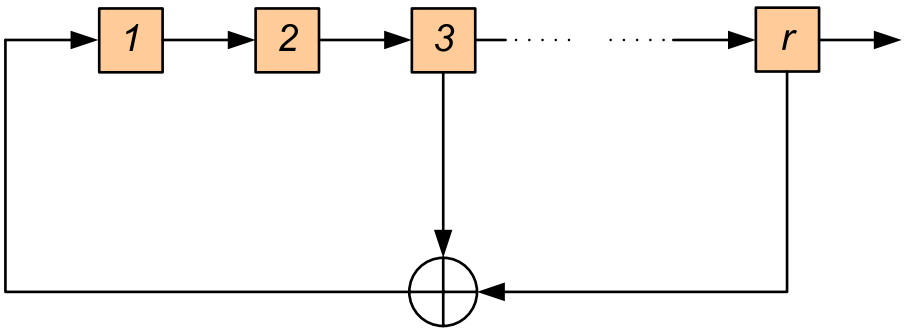
Všechny tyto jmenované skvělé výhody SystemCrafter SC však nejsou zadarmo. Na oficiální stránce SystemCrafter (www.systemcrafter.com) je software SystemCrafter SC Compiler k mání za 3 000 USD [11], což není zrovna málo. Navíc tento software generuje RTL VHDL či Verilog pro syntézu pouze na Xilinx FPGA [11], což z něj rozhodně nedělá univerzálního pomocníka při návrhu hardware.

6 Návrhy HW implementací šifer

Pro návrh hardwarové implementace šifrovacího algoritmu jsem si vybral proudovou šifru s generátorem pseudonáhodné posloupnosti založeným na LFSR a jako další implementaci jsem zvolil blokovou šifru GOST.

6.1 Návrh a simulace proudové šifry

Proudová šifra s generátorem PNP se v praxi často využívá (například proudová šifra A5 v systému GSM), proto jsem se rozhodl navrhnout právě proudovou šifru. Na Obr. 6.1 je uvedeno schéma LFSR se třemi až *r* buňkami a takovým nastavením vazeb, že na XOR je přiveden výstup z třetí a *r*-té buňky. Pro upřesnění se jedná o tzv. Fibonacciho LFSR. Výstup celého LFSR generátoru (z *r*-té buňky) slouží jako heslo pro šifrování zprávy v proudové šifře, resp. pro operaci XOR mezi heslem a zprávou (viz. kapitola 3.4). Protože se jedná o generátor pseudonáhodné posloupnosti, dojde za určitý čas k opakování generované posloupnosti, tento čas se nazývá perioda hesla a je dán vztahem $2^r - 1$, kde *r* je počet buněk registru [2].



Obr. 6.1: Generátor pseudonáhodné postupnosti LFSR.

6.1.1 Proudová šifra ve VHDL

V prostředí návrhového systému Quartus II Version 9.0 jsem v jazyce VHDL navrhnul proudovou šifru s generátorem pseudonáhodné posloupnosti LFSR se 32 buňkami. Generátor LFSR bude mít tedy délku periody rovnu $2^{32} - 1 = 4\ 294\ 967\ 295$, což je pro účely této práce dostatečná délka. Nastavení vazeb mezi buňkami a XOR je prováděno polynomem, který je reprezentován sledem binárních hodnot. Chceme-li nastavit generátor například polynomem $x^{13} + 1$, zadáme mu následující vektor binárních hodnot:

[1 0 0 0 0 0 0 0 0 0 0 0 0 1],

který v případě výskytu čísla 1 realizuje vazbu mezi XOR a buňkou LFSR o pořadovém číslu odpovídajícímu řádu prvku v polynomu (v příkladu je to řád 13 a 0). Dále zadáme hodnoty výchozího stavu jednotlivých buněk, většinou se volí tyto hodnoty:

[0 0 0 0 0 0 0 0 0 0 0 0 0 1]

(všechny buňky kromě té poslední jsou nastaveny na 0) a vše pro generování pseudonáhodné posloupnosti máme připraveno.

V prostředí Quartus byly k návrhu proudové šifry použity dvě entity – jedna nadřazená provádějící šifrování („sifrator.vhd“), která si bere jako heslo výstup z entity podřazené (generátoru PNP „lsfr.vhd“). Název „lsfr“ vznikl přesmyknutím prostředních písmen akronymu LFSR. Zde uvádím zdrojový kód VHDL (komentáře mají zelenou barvu písma):

```
-----------------------------------------------------------------
-- sifrator.vhd --
-- proudova sifra - operace XOR mezi vstupni zpravou a heslem
--                                - vystupem je kryptogram
-----------------------------------------------------------------
-- pouzij knihovnu ieee 1164 -- definuje rozsirene udalosti signalu - hrany
library ieee;
use ieee.std_logic_1164.all;
-- definice rozhrani obvodu
entity sifrator is -- vstupy a vystupy obvodu
    port ( vstup   : in std_logic; --vstup pro bity zpravy
           vystup  :out std_logic; --vystup zasifrovanych bitu
           hodiny  : in std_logic;
           reset   : in std_logic;
           klic    : in std_logic_vector (0 to 31) --vstup pro klic generatoru
           );
end sifrator;

architecture behavior of sifrator is -- popisuje chovani (behavior) obvodu:
--pripojeni podrazene entity
component lsfr is
    port ( hodiny  : in std_logic;
           reset   : in std_logic;
           vystup  : out std_logic; --bity hesla
           klic    : in std_logic_vector (0 to 31)--bity klice
           );
end component;
-- deklarace vnitrnich signalu:
signal      hi      : std_logic; --signal nesouci vystup generatoru "lsfr" (heslo) na vstup
operace XOR pri sifrovani
-- telo bloku:
begin
gen: lsfr --propojeni portů entity "sifrator" na podrazenou entitu "lsfr" a zpristupneni
signalu ki
    port map (hodiny, reset, hi, klic); --mapovani portu sifrator.hodiny -> lsfr.hodiny, totez
reset, sifrator.hi -> lsfr.vystup , sifrator.klic -> lsfr.klic
    vystup <= vstup xor hi; --operace XOR mezi vstupni zpravou "vstup" a vystupem lsfr "hi"
(heslem), ulozeni do "vystup"
end behavior;

-------------------------------------------------------------
-- lsfr.vhd --
-- generator pseudonahodneho signalu-generuje PNP dle zadaneho vektoru, ktery
--   reprezentuje koeficienty polynomu
-------------------------------------------------------------
-- pouzij knihovnu ieee 1164 -- definuje rozsirene udalosti signalu - hrany
library ieee;
use ieee.std_logic_1164.all;
-- definice rozhrani obvodu
entity lsfr is -- vstupy a vystupy obvodu
    port ( hodiny         : in std_logic;
           reset          : in std_logic;
           vystup         : out std_logic; --bity hesla
           klic           : in std_logic_vector (0 to 31) --bity klice
           );
end lsfr;
```

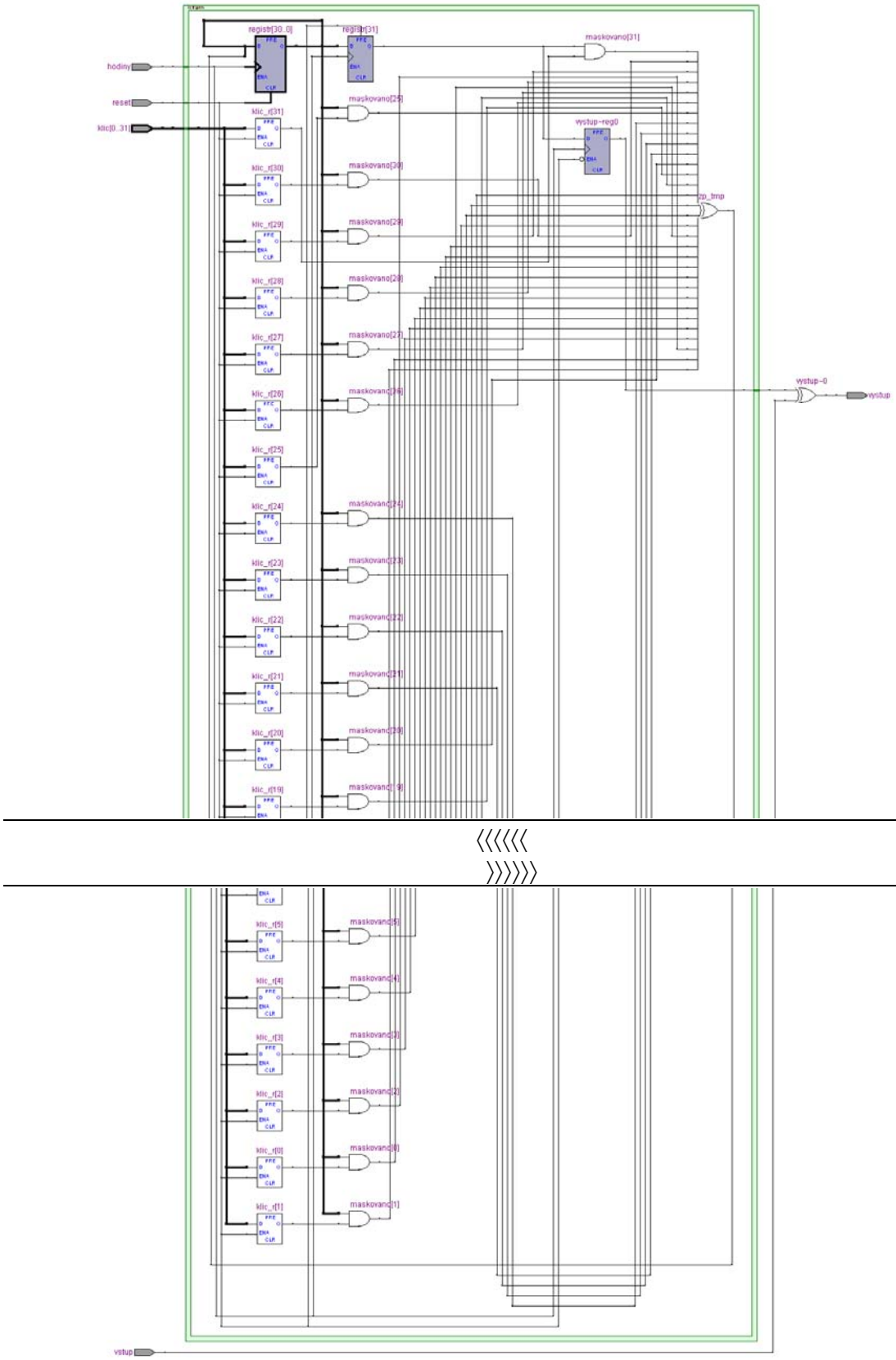
```
architecture behavior of lsfr is -- popisuje chovani (behavior) obvodu:
-- deklarace vnitrnich signalu:
     signal registr : std_logic_vector(0 to 31); --definice 32bit registru
     signal klic_r  : std_logic_vector (0 to 31); --definice 32bit registru pro klic
     signal maskovano: std_logic_vector(0 to 31); --pomocny 32bit signal
     signal zp      : std_logic; --pomocny signal pro heslo
-- telo bloku:
begin
 process(reset, hodiny) begin--process reagujici na signaly reset a hodiny
     if(reset = '1') then --na hodnotu 1 signalu "reset" provede:
      registr <= B"0000_0000_0000_0000_0000_0000_0000_0001"; --vychozi stav registrů generatoru
      klic_r <=klic; --priradi data z portu klic do registru klic_r generatoru
     elsif(hodiny'event and hodiny = '1') then --na hodnotu 1 signalu "hodiny" provede:
      vystup <= registr(31); --posledni bit z registru je presunut na port vystup jakozto 1 bit
hesla
      registr <= zp & registr(0 to 30); --vlozi vystup XORu do prvni bunky "registru" pred
bunky 0 az 30
     end if;
 end process;

 process (maskovano) --process reagujici na signal maskovano
      variable zp_tmp : std_logic; --pomocna promenna k vypoctu konecne hodnoty "zp", coz je 1-
bit. vystup z generatoru
        begin
      zp_tmp := maskovano(0);      --inicializace pomocne promenne zp_tmp
      for i in 1 to 31 loop --cyklus nastavujici vazby(odbocky) z jednotlivych bunek lsfr do
XORu
            zp_tmp := zp_tmp xor maskovano(i);
      end loop;
      zp <= zp_tmp;--ulozeni pomocne promenne do "zp"
 end process;

maskovano <= registr(0 to 31) and klic_r(0 to 31); --"maskovano" popisuje, z jakych bunek lsfr
budou vest zpetne vazby do XORu

end behavior;
```

Funkce kódu by se dala popsat následovně. Do signálu *klic* se v simulátoru zadá vektor koeficientů reprezentující nastavení vazeb v generátoru LFSR, dále se zadá vstupní zpráva (*vstup*), kterou chceme zašifrovat.

Entita „sifrator.vhd“ vykonává vlastně jen operaci XOR mezi výstupem z generátoru a vstupní zprávou. Výstup z generátoru si bere pomocí mapování portů, konkrétně přes signál *hi*, který odpovídá signálu *vystup* entity „lfsr.vhd“. Jakmile „lfsr“ tento signál vypustí, „sifrator“ ho přivede na jeden vstup šifrovací operace XOR a na druhý vstup operace XOR je přivedena zadaná zpráva. Výsledek pak uloží do signálu *vystup* a ten představuje kryptogram.

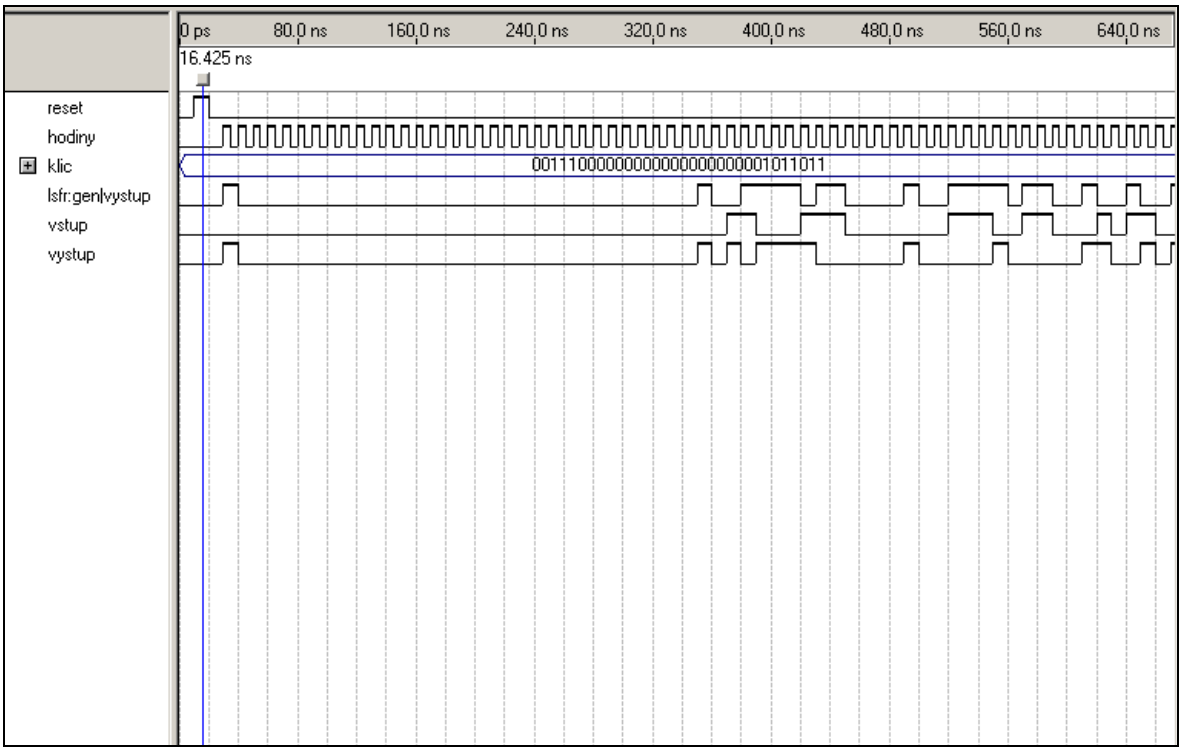
Entita „lfsr.vhd“ obsahuje 32-bitový registr představující 32 buňek generátoru. Při hodnotě „1“ signálu *reset* se do pomocného signálu *klic_r* díky mapování portů v entitě „sifrator“ uloží zadaný klíč (*klic*) a načte se výchozí stav buněk (31 nul a jedna jednička). Na hodnotu „1“ signálu *hodiny* se potom provádí uložení bitu poslední buňky do signálu *vystup*, pak se obsah buněk přesune o 1 směrem k poslední buňce tak, aby bylo v první buňce místo na nový bit vycházející z operace XOR, obsah poslední buňky se tak přepíše novou hodnotou, která je opět poslána na *vystup* a generování takto pokračuje až do konce simulace. Operace XOR je plněna hodnotami z těch buněk, které jsou vybrány podle zadaného klíče (*klic*). Takže když je na prvním místě klíče hodnota 1, vytvoří se vazba XORu s první buňkou, když je na 16-tém místě klíče hodnota 1, vede do XORu 16-tá buňka apod.

Kód je v programu Quartus po úspěšné analýze a syntéze kompilován a je vytvořen RTL soubor, jenž je ukázán ve zmenšené a oříznuté podobě na Obr. 6.2. V celé velikosti a ve větším rozlišení včetně popisu funkcí částí schématu je uveden v příloze „Příloha A“.

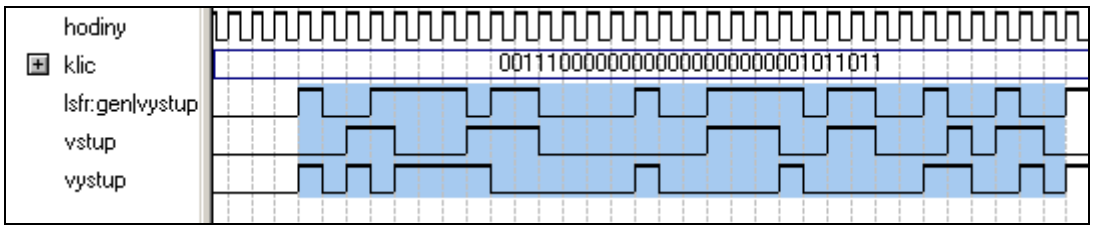


Obr. 6.2: Náhled do RTL souboru proudové šifry s LFSR.

Po tomto kroku je potřeba v programu Quartus vygenerovat funkcionální simulační netlist a poté nastavit simulačních parametry. Na signál *vstup* se přivede nějaká posloupnost 32 binárních znaků, je možné použít i náhodných znaků vygenerovaných programem. Dále se do signálu *klic* načtou koeficienty polynomu nastavujícího vazby buněk LFSR a to v okamžiku, kdy signál *reset* nabude hodnoty „1“. Před samotnou simulací je ještě třeba nastavit hodinový signál a také impulz signálu *reset* (obojí je vidět na výsledku simulace v obrázku Obr. 6.3). Detail výsledných hodnot simulace je potom zobrazen v Obr. 6.4. Modře je v Obr. 6.4 vyznačena část simulace obsahující 32 bitů vstupní zprávy a jí odpovídající heslo a kryptogram.



Obr. 6.3: Výsledek simulace proudové šifry s LFSR.



Obr. 6.4: Detail výsledku simulace proudové šifry s LFSR.

Z výsledku na Obr. 6.4 je možné vidět vektor koeficientů polynomu zadaný do signálu *klic* a dále hodnoty signálů *lsfr:gen|vystup* (heslo), *vstup* (vstupní zpráva) a *vystup* (zašifrovaná zpráva), jejichž číselné vyjádření je následovné:

lsfr:gen|vystup: 10011110110000100111101100100100

vstup: 00110001110000000111001100010110

vystup: 10101111000000100000100000110010

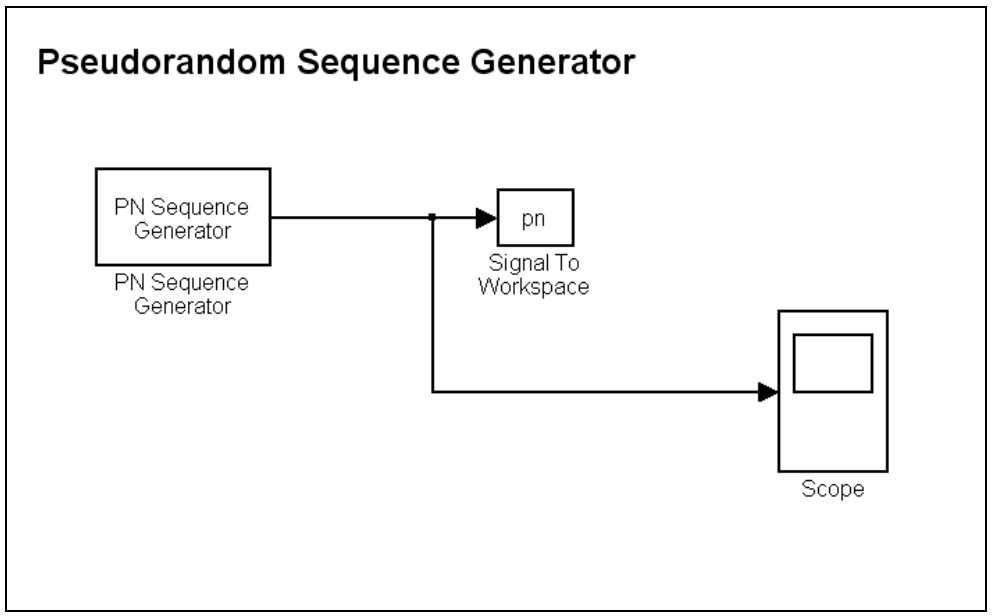
Mezi těmito signály očividně platí následující vztah:

lsfr:gen|vystup XOR *vstup* = *vystup*. (6.1)

Proudová šifra tedy pracuje korektně. Bylo odzkoušeno, že při zadání jiného (např. náhodného) klíče (do signálu *klic*), se PNP posloupnost změnila. Takže pokud bychom každou další zprávu šifrovali pomocí jiného klíče, jednalo by se principielně o dokonalou šifru.

6.1.2 Proudová šifra v Matlabu

Pro porovnání výsledků proudové šifry s LFSR navrhnuté ve VHDL jsem navrhnul proudovou šifru v programu Matlab, v němž proudová šifra rovněž čerpala heslo z generátoru LFSR. Generátor LFSR byl v tomto případě realizován blokem *PN Sequence Generator* obsaženým v knihovně Simulink. Blokové schéma generátoru je uvedeno na Obr. 6.5.



Obr. 6.5: Generátor PNP v Simulinku.

Blok PN Sequence Generator generuje pseudonáhodnou posloupnost užitím LFSR, který je konfigurován parametrem Polynom generátoru (*Generator polynomial*). Například při požadování polynomu generátoru $x^6 + x + 1$ se do parametru bloku zadá vektor koeficientů [1 0 0 0 0 1 1]. Jak je vidět z Obr. 6.5, výstup generátoru je přiveden na osciloskop a také je uložen do vektoru *pn*, aby se s jeho výstupními hodnotami dalo dále pracovat. Samotnou šifru potom vykonává skript „proudova_sifra.m“, jehož kód vypadá následovně (komentáře mají zelenou barvu písma):

```
% ***************************************************************
% ***************** proudova sifra: ****************************
% ***************************************************************
function proudova_sifra(Z,PN)
 %Z je vektor vstupni zpravy
 %PN je vektor pseudonahodne posloupnosti

% transponovani vektoru PN - vygenerovane pseudonahodne posloupnosti:
PN=PN';
%omezeni delky hesla H a kryptogramu C:
n=32;
%inicializace vektoru hesla:
H=zeros(1,n);
%inicializace vektoru kryptogramu:
C=zeros(1,n);
% zobrazeni vstupni zpravy:
Z
%nacteni n hodnot z vektoru PN s vynechanim pocatecniho nastaveni
%generatoru:
H=PN(33:(33+n-1))
%************** sifrovani ***********************
%operace XOR mezi zpravou Z a heslem H:
C = xor (Z ,H)
%************ desifrovani **********************
%operace XOR mezi kryptogramem C a heslem H:
Z = xor (C, H)
```

V příkazovém okně Matlabu se nejprve uloží vstupní zpráva do vektoru Z a poté se zadá příkaz `proudova_sifra(Z,PN)`, kde parametr Z představuje vstupní zprávu (pokud by se vstupní zpráva zadala do vektoru s jiným písmenem, zadalo by se toto písmeno namísto Z) a parametr PN představuje vektor pseudonáhodné posloupnosti vzniklý v generátoru v Simulinku.

Uvnitř skriptu se odehrává následující. Vstupem do skriptu je tedy zadaná zpráva Z a posloupnost PN získaná z generátoru PNP. Vektor PN je v Simulinku uložen jako sloupec, takže je potřeba ho nejprve transponovat. Tento skript se dá teoreticky použít pro bezpečné šifrování zpráv o délce až $n=2^{32}-1$, což je délka periody použitého LFSR generátoru. Vektory C a H jsou inicializovány na nastavenou délku n a naplněny nulami. Do vektoru H je načteno n bitů vektoru PN tak, aby neobsahoval prvních n bitů z generátoru, neboť tyto se rovnají počátečnímu nastavení generátoru. Poté je provedena stěžejní operace, a to sice operace XOR mezi zprávou Z a heslem H. V posledním řádku skriptu je provedeno dešifrování. Výstupem je pak kryptogram C a také pro kontrolu i dešifrovaný kryptogram, který se v případě správnosti algoritmu rovná původní zprávě Z. Příklad funkce skriptu na konkrétních hodnotách Z, H a C použitých i ve VHDL proudové šifře je ukázán níže.

Zadané koeficienty polynomu (klíč):

[1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 1]

(první jednička je zde navíc oproti zadávání klíče v Quartus, neboť Simulink tuto vyžaduje jakožto reprezentaci vazby výstupu XORu na vstup první buňky LFSR),

zadaný počáteční stav buněk:

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1],

zadaná zpráva Z:
Z = 00110001110000000111001100010110,

použité heslo:
H = 10011110110000100111101100100100,

kryptogram:
C = 10101111000000100000100000110010,

dešifrovaný kryptogram:
Z = 00110001110000000111001100010110.

Je vidět, že kryptogram C vzniklý operací XOR mezi zprávou Z a heslem H je správný. Rovněž dešifrování kryptogramu C zpět na zprávu pomocí operace Z = C XOR H proběhne správně.

6.1.3 Shrnutí výsledků

Správnost šifrování proudové šifry navrhnuté ve VHDL byla ověřena proudovou šifrou navrhnutou v Matlabu. Oběma byly zadány shodné vstupní zprávy, obě šifry mají generátor LFSR nastavený na stejné vazby buněk XOR, z obou vyšly stejné kryptogramy a pomocí šifry v Matlabu byl kryptogram dešifrován a odpovídal původní zprávě. Návrh implementace proudové šifry do hardwaru v jazyce VHDL tedy považuji za korektní.

6.2 Návrh a simulace šifry GOST

V prostředí Quartus jsem dále navrhnul blokovou šifru GOST. Šifru GOST jsem si vybral proto, že je podobná hojně užívané šifře DES, ale není sama příliš rozšířena. Jedná se o blokovou šifru vhodnou k bezpečnému šifrování důvěrných dat. V prostředí Quartus v rámci VHDL kódu byly vytvořeny tři entity, nadřazená „gost_32.vhd“ provádí opakování 32 rund šifry GOST a také přiřazuje subklíče k jednotlivým rundám, jí podřazená „gost_runda.vhd“ provádí operace jedné rundy šifry GOST a jí podřazená „sboxes.vhd“ vykonává S-box substituce.

Kód všech tří entit uvádím zde (komentáře mají zelenou barvu písma):

```
-------------------------------------------------
-- gost_32.vhd --
-- blokova sifra GOST 28147-89
-------------------------------------------------
library ieee; -- pouzij uvedene knihovny:
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
-- definice rozhrani obvodu
entity gost_32 is  -- vstupy a vystupy obvodu
    port ( klic   : in std_logic_vector(255 downto 0); --vstup pro 256bit klic
           zprava : in std_logic_vector(63 downto 0); --vstup pro 64bit zpravu
           reset  : in std_logic;
           hodiny : in std_logic;
           done   : out std_logic; --signal, ze skoncil vypocet kryptogramu
           sifra  : out std_logic_vector(63 downto 0) --kryptogram
           );
end gost_32;

architecture behavior of gost_32 is-- popisuje chovani (behavior) obvodu
-- deklarace vnitrnich signalu:
signal mezivysledek_L     : std_logic_vector(31 downto 0); --vstup do rundy (leva polovina)
signal mezivysledek_R     : std_logic_vector(31 downto 0); --vstup do rundy (prava polovina)
```

```
signal mezivysledek_L2     : std_logic_vector(31 downto 0); --vystup z rundy (leva polovina)
signal mezivysledek_R2     : std_logic_vector(31 downto 0); --vystup z rundy (prava polovina)
signal citac_rund  : std_logic_vector(5 downto 0); --citac rund
signal pre_count   : std_logic_vector(5 downto 0); --pomocny signal citace
signal rundovni_klic       : std_logic_vector(31 downto 0); --subklic pro danou rundu
signal done_tmp    : std_logic;   -- pomocny signal pro docasne ulozeni noveho "done"
--pripojeni podrazene entity:
component gost_runda
    port ( vstup_L: in std_logic_vector (31 downto 0); --vstup pro levy blok zpravy
    vstup_R: in std_logic_vector (31 downto 0); --vstup pro pravy blok zpravy
           vyst_L  : out std_logic_vector (31 downto 0); --vystupni levy blok zpravy
           vyst_R  : out std_logic_vector (31 downto 0); --vystupni pravy blok zpravy
           klic    : in std_logic_vector (31 downto 0) --vstup pro klic
     );
end component;

begin
    process(hodiny, reset) --process reagujici na signaly hodiny a reset
    begin
     if reset = '1' then --na hodnotu 1 "reset" provede vynulovani uvedenych signalu:
                  mezivysledek_L <= (others => '0');
                  mezivysledek_R <= (others => '0');
                  pre_count <= (others=>'0');
                  done_tmp <= '0';
     elsif hodiny'event and hodiny = '1' then --na hodnotu 1 signalu "hodiny" provede:
                  pre_count <= citac_rund + '1'; --konstrukce, ktera je bezpecna pro syntezu
- nepracuje se primo se signalem "citac_rund", ale s pomocnym "pre_count", jehoz hodnota se na
konci zapise do "citac_rund"
     if citac_rund = b"000000" then --pokud je "citac_rund" = 0, tak rozdeli vstupni zpravu a
ulozi ji do praveho a leveho mezivysledku
           mezivysledek_L <= zprava(63 downto 32);
           mezivysledek_R <= zprava(31 downto 0);
           done_tmp <= '0'; --pomocny signal "done_tmp" nastaven na nulu
     elsif citac_rund = b"100000" then -- pokud je "citac_rund" = 32:
           done_tmp <= '1';--pomocny signal "done_tmp" nastaven na 1
     else --ostatni pripady, tj. pokud "citac_rund" je v otevřeném intervalu (0;32),
                -- vystup z rundy zapise na vstup rundy nasledujici:
           mezivysledek_L <= mezivysledek_L2;
           mezivysledek_R <= mezivysledek_R2;
           done_tmp <= '0';--pomocny signal "done_tmp" nastaven na nulu
     end if;
    end if;
end process;
--zapsani hodnot pomocnych signalu do hlavnich signalu:
done <=done_tmp;
citac_rund <= pre_count;

process (done_tmp, mezivysledek_L2, mezivysledek_R2)--tento blok osetruje, aby data v "sifra"
nevisela az do dalsiho impulzu "done",tj. do ziskani vysledku dalsiho sifrovani
    begin
     if (done_tmp = '1') then
           sifra <= mezivysledek_R2 & mezivysledek_L2 ; --zapsani vysledku sifrovani do
"sifra" v prohozenem poradi (pravidlo Feistelovy rundy)
     else
           sifra <= (others => '0');
     end if;
end process;

process(citac_rund, klic) -- prirazovani subklicu podle toho, jaka runda prave probiha:
--uvedene reseni vytvari v RTL schematu kaskadu 32 muxu (coz je pocet vetvi elseif v kodu)
    begin
           if citac_rund = O"01" then -- osmickova soustava 01 = 1
                  rundovni_klic <= klic(31 downto 0);
           elsif citac_rund = O"02" then --02 = 2
                  rundovni_klic <= klic(63 downto 32);
           elsif citac_rund = O"03" then
                  rundovni_klic <= klic(95 downto 64);
           elsif citac_rund = O"04" then
                  rundovni_klic <= klic(127 downto 96);
           elsif citac_rund = O"05" then
                  rundovni_klic <= klic(159 downto 128);
           elsif citac_rund = O"06" then
                  rundovni_klic <= klic(191 downto 160);
           elsif citac_rund = O"07" then
                  rundovni_klic <= klic(223 downto 192);
           elsif citac_rund = O"10" then --= 8
                  rundovni_klic <= klic(255 downto 224);
```

```
            elsif citac_rund = O"11" then -== 9
                   rundovni_klic <= klic(31 downto 0);
            elsif citac_rund = O"12" then -== 10
                   rundovni_klic <= klic(63 downto 32);
            elsif citac_rund = O"13" then -== 11
                   rundovni_klic <= klic(95 downto 64);
            elsif citac_rund = O"14" then -== 12
                   rundovni_klic <= klic(127 downto 96);
            elsif citac_rund = O"15" then -== 13
                   rundovni_klic <= klic(159 downto 128);
            elsif citac_rund = O"16" then -== 14
                   rundovni_klic <= klic(191 downto 160);
            elsif citac_rund = O"17" then -== 15
                   rundovni_klic <= klic(223 downto 192);
            elsif citac_rund = O"20" then -== 16
                   rundovni_klic <= klic(255 downto 224);
            elsif citac_rund = O"21" then -== 17
                   rundovni_klic <= klic(31 downto 0);
            elsif citac_rund = O"22" then -== 18
                   rundovni_klic <= klic(63 downto 32);
            elsif citac_rund = O"23" then -== 19
                   rundovni_klic <= klic(95 downto 64);
            elsif citac_rund = O"24" then -== 20
                   rundovni_klic <= klic(127 downto 96);
            elsif citac_rund = O"25" then -== 21
                   rundovni_klic <= klic(159 downto 128);
            elsif citac_rund = O"26" then -== 22
                   rundovni_klic <= klic(191 downto 160);
            elsif citac_rund = O"27" then -== 23
                   rundovni_klic <= klic(223 downto 192);
            elsif citac_rund = O"30" then -== 24
                   rundovni_klic <= klic(255 downto 224);
            elsif citac_rund = O"31" then -== 25
                   rundovni_klic <= klic(255 downto 224);
            elsif citac_rund = O"32" then -== 26
                   rundovni_klic <= klic(223 downto 192);
            elsif citac_rund = O"33" then -== 27
                   rundovni_klic <= klic(191 downto 160);
            elsif citac_rund = O"34" then -== 28
                   rundovni_klic <= klic(159 downto 128);
            elsif citac_rund = O"35" then -== 29
                   rundovni_klic <= klic(127 downto 96);
            elsif citac_rund = O"36" then -== 30
                   rundovni_klic <= klic(95 downto 64);
            elsif citac_rund = O"37" then -== 31
                   rundovni_klic <= klic(63 downto 32);
            elsif citac_rund = O"40" then -== 32
                   rundovni_klic <= klic(31 downto 0);
            else
                   rundovni_klic <= (others => 'X'); --osetruje, aby syntetizator nedaval
klopne obvody tam, kde staci jen primy spoj, resp. kde neni potrebo uchovat stav tohoto spoje
            end if;
end process;

runda: gost_runda --mapuje signaly v gost_32.vhd na porty v gost_runda.vhd
    port map (mezivysledek_L, mezivysledek_R, mezivysledek_L2, mezivysledek_R2,
rundovni_klic);
end behavior;


-------------------------------------------------
-- gost_runda.vhd --
-- runda blokove sifry GOST 28147-89
-------------------------------------------------
-- pouzij uvedene knihovny:
library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
-- definice rozhrani obvodu
entity gost_runda is -- vstupy a vystupy obvodu
    port ( vstup_L : in std_logic_vector (31 downto 0); --vstup pro levy blok zpravy
           vstup_R : in std_logic_vector (31 downto 0); --vstup pro pravy blok zpravy
           vyst_L  : out std_logic_vector (31 downto 0); --vystupni levy blok zpravy
           vyst_R  : out std_logic_vector (31 downto 0); --vystupni pravy blok zpravy
           klic    : in std_logic_vector (31 downto 0) --vstup pro klic
```

```
            );
end gost_runda;
architecture behavior of gost_runda is -- popisuje chovani (behavior) obvodu
-- deklarace vnitrnich signalu:
signal      sbox_to        : std_logic_vector (31 downto 0); --pomocny signal sboxu
signal      sbox_from      : std_logic_vector (31 downto 0); --pomocny signal sboxu
signal      rotL11         : std_logic_vector (31 downto 0); --pomocny signal levotociveho
posunu
signal vstup_R_cislo       : unsigned (31 downto 0); -- pomocny signal pro praci s vektorem
jako s cislem bez znam.
signal klic_cislo  : unsigned (31 downto 0);  -- pomocny signal pro praci s vektorem jako s
cislem bez znam.
--pripojeni podrazene entity:
component sboxes
     port (
            sbox_vstup: in std_logic_vector(31 downto 0);
            sbox_vystup: out std_logic_vector(31 downto 0)
             );
     end component;

begin
     vyst_L <= vstup_R; --zapise vstup_R na vyst_L
     --nastaveni pomocnych signalu:
     vstup_R_cislo <= unsigned(vstup_R);
     klic_cislo <= unsigned(klic);
     sbox_to <= vstup_R_cislo + klic_cislo; -- součet modulo 2^32
     rotL11  <= sbox_from(19 downto 0) & sbox_from(31 downto 20); --levotociva rotace o 11
     vyst_R <= vstup_L xor rotL11; --operace XOR a zapsani vysledku prave poloviny do "vyst_R"

      sbox: sboxes --mapuje signaly v sboxes.vhd na porty v gost_runda.vhd
            port map (sbox_to, sbox_from);
end behavior;
```

Z důvodu úspory místa neuvádím kód všech osmi S-boxů, neboť jejich definice je zdlouhavá. Kompletní je uveden pouze první S-box, u zbývajících sedmi je uveden pouze první řádek jejich definice.

```
-------------------------------------------------
-- sboxes.vhd --
-- S-boxy
-------------------------------------------------
-- pouzij knihovnu ieee 1146
library ieee;
use ieee.std_logic_1164.all;
-- definice rozhrani obvodu
entity sboxes is -- vstupy a vystupy obvodu
     port (
            sbox_vstup: in std_logic_vector(31 downto 0);
            sbox_vystup: out std_logic_vector(31 downto 0)
             );
end sboxes;
architecture behavior of sboxes is -- popisuje chovani (behavior) obvodu:
begin
     -- definice S-boxu (zapsano v hexadecimalnim tvaru):
process (sbox_vstup) begin
     case sbox_vstup(3 downto 0) is -- S-box 1 (4,10,9,2,13,8,0,14,6,11,1,12,7,15,5,3)
                    when X"0" =>
                    sbox_vystup(3 downto 0) <= X"4";
                    when X"1" =>
                    sbox_vystup(3 downto 0) <= X"a";--10
                    when X"2" =>
                    sbox_vystup(3 downto 0) <= X"9";
                    when X"3" =>
                    sbox_vystup(3 downto 0) <= X"2";
                    when X"4" =>
                    sbox_vystup(3 downto 0) <= X"d";--13
                    when X"5" =>
                    sbox_vystup(3 downto 0) <= X"8";
                    when X"6" =>
                    sbox_vystup(3 downto 0) <= X"0";
                    when X"7" =>
                    sbox_vystup(3 downto 0) <= X"e";--14
                    when X"8" =>
                    sbox_vystup(3 downto 0) <= X"6";
```

```
                    when X"9" =>
                    sbox_vystup(3 downto 0) <= X"b";--11
                    when X"a" =>
                    sbox_vystup(3 downto 0) <= X"1";
                    when X"b" =>
                    sbox_vystup(3 downto 0) <= X"c";--12
                    when X"c" =>
                    sbox_vystup(3 downto 0) <= X"7";
                    when X"d" =>
                    sbox_vystup(3 downto 0) <= X"f";--15
                    when X"e" =>
                    sbox_vystup(3 downto 0) <= X"5";
                    when X"f" =>
                    sbox_vystup(3 downto 0) <= X"3";
            end case;
    case sbox_vstup(7 downto 4) is -- S-box 2 -- {14,11,4,12,6,13,15,10,2,3,8,1,0,7,5,9}
    ...
    case sbox_vstup(11 downto 8) is -- S-box 3 -- {5,8,1,13,10,3,4,2,14,15,12,7,6,0,9,11}
    ...
    case sbox_vstup(15 downto 12) is -- S-box 4 -- {7,13,10,1,0,8,9,15,14,4,6,12,11,2,5,3}
    ...
    case sbox_vstup(19 downto 16) is -- S-box 5 -- {6,12,7,1,5,15,13,8,4,10,9,14,0,3,11,2}
    ...
    case sbox_vstup(23 downto 20) is -- S-box 6 -- {4,11,10,0,7,2,1,13,3,6,8,5,9,12,15,14}
    ...
    case sbox_vstup(27 downto 24) is -- S-box 7 -- {13,11,4,1,3,15,5,9,0,10,14,7,6,8,2,12}
    ...
    case sbox_vstup(31 downto 28) is -- S-box 8 --  {1,15,13,0,5,7,10,4,9,2,3,14,6,11,8,12}
    ...
    end process;
end behavior;
```

Kód byl v průběhu ladění mnohokrát upravován a optimalizován za účelem zjednodušení výsledné implementace do hardwaru (resp. zjednodušení RTL souboru). Po každé úpravě bylo potřeba znovu otestovat funkčnost kódu, což bylo někdy časově náročné, neboť kompilace trvala i 50 sekund. A jelikož se jedná o poměrně složitý kód, někdy bylo velmi obtížné najít chybu a odstranit tak chybová hlášení, která při kompilaci program Quartus oznamoval.

Kód pracuje následujícím způsobem. Do signálu *klic* se v simulátoru zadá 256 bitů klíče a do signálu *zprava* potom 64 bitů zprávy, kterou chceme zašifrovat. V „gost_32.vhd“ je na začátku tato zpráva rozdělena na pravou a levou polovinu a uložena do pomocných registrů *mezivysledek_L* a *mezivysledek_R*, které jsou namapovány na porty entity „gost_runda“, v níž se provádí rundovní operace a výsledky se vrací zpět do „gost_32“. Probíhá to tak, že výstupy z jedné rundy se zapíšou na vstupy rundy následující. Výsledné poloviny poslední rundy jsou potom v prohozeném pořadí zapsány do výstupního signálu *sifra*. Signál *citac_rund* je na začátku inicializován na nulu a poté počítá provedené rundy, což slouží jako ukazatel pro zápis výsledku poslední rundy na výstup, který je proveden při dosažení hodnoty čítače 32. Entita „gost_32“ má také na starost přiřazování subklíčů jednotlivým rundám. Máme k dispozici osm 32-bitových subklíčů ($k1$ … $k8$), které jsou k jednotlivým rundám ($K1$ … $K32$) přiřazeny podle Tab. 6.1.

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

Tab. 6.1: Tabulka rundovních klíčů pro šifrování.

V entitě „gost_runda.vhd“ se provádí všechny rundovní operace, jak již bylo řečeno. Z „gost_32“ se vždy přivede levá a pravá polovina vstupní zprávy (každá o velikosti 32 bitů), se kterými jsou provedeny podle standardu šifry GOST sečtení modulo 2^{32}, substituce pomocí s-boxů a XOR.

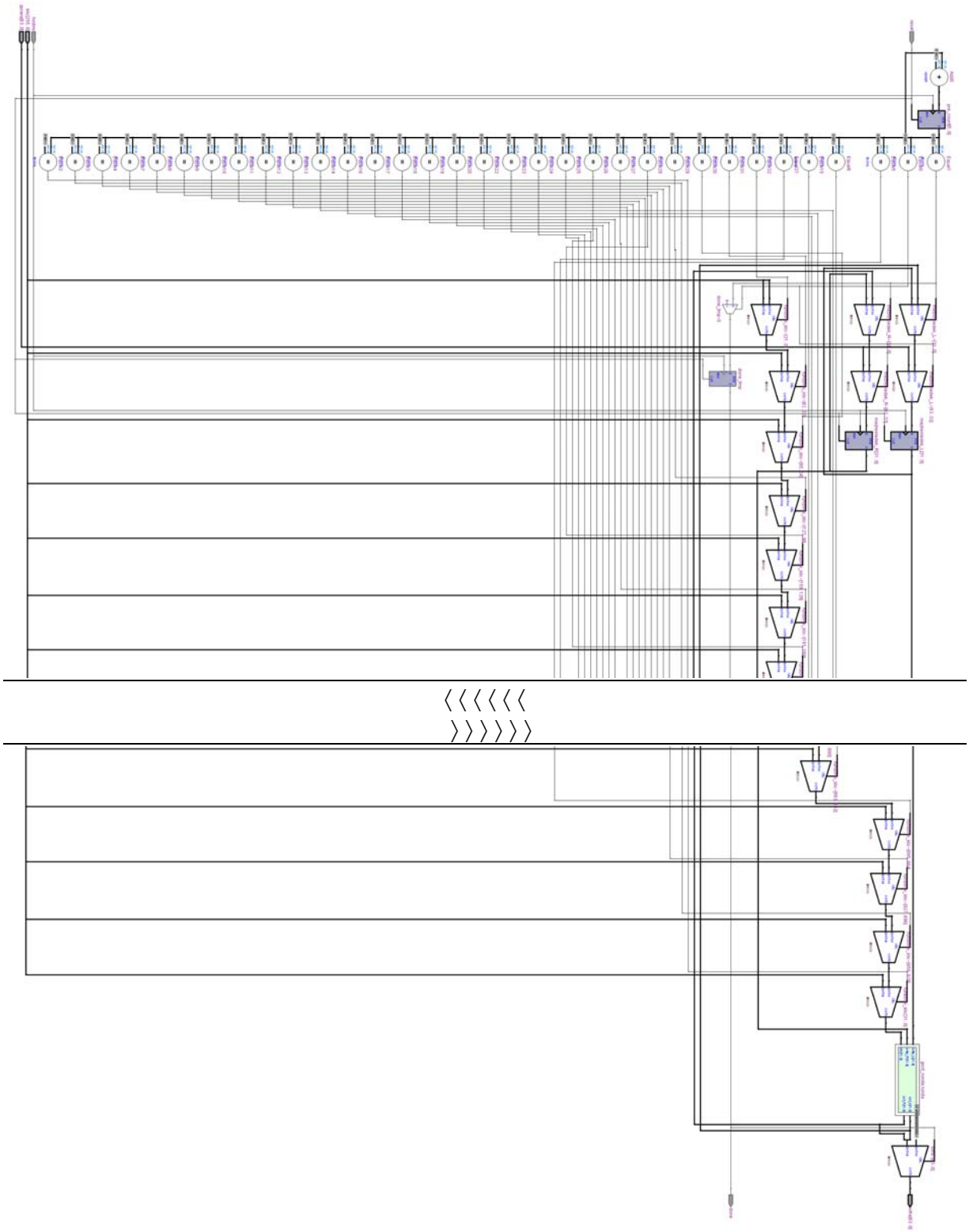
S-box substituce jsou prováděny odděleně ve třetí nejpodřazenější entitě „sboxes.vhd“. Z „gost_runda“ je vždy na vstup „sboxes“ (*sbox_vstup*) přiveden 32-bitový výsledek první operace rundy (sčítání modulo 2^{32}), který je potom rozkouskován na osm 4-bitových části, s nimiž jsou provedeny substituce v S-box 1 až 8 podle Tab. 6.2.

vstup	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
výstup	**S-box 1:**															
	4	10	9	2	13	8	0	14	6	11	1	12	7	15	5	3
	S-box 2:															
	14	11	4	12	6	13	15	10	2	3	8	1	0	7	5	9
	S-box 3:															
	5	8	1	13	10	3	4	2	14	15	12	7	6	0	9	11
	S-box 4:															
	7	13	10	1	0	8	9	15	14	4	6	12	11	2	5	3
	S-box 5:															
	6	12	7	1	5	15	13	8	4	10	9	14	0	3	11	2
	S-box 6:															
	4	11	10	0	7	2	1	13	3	6	8	5	9	12	15	14
	S-box 7:															
	13	11	4	1	3	15	5	9	0	10	14	7	6	8	2	12
	S-box 8:															
	1	15	13	0	5	7	10	4	9	2	3	14	6	11	8	12

Tab. 6.2: Tabulka substitucí.

Po provedení substitucí je oněch 8 bloků složeno dohromady a jako 32-bitový signál *sbox_vystup* odchází zpět do „gost_runda“.

RTL soubor vytvořený programem Quartus je ukázán na Obr. 6.6. Schéma šifry GOST je značně složité a rozsáhlé, není zde proto možné ukázat celé schéma s veškerými detaily. Celé schéma ve větším rozlišení včetně popisu funkcí jeho jednotlivých částí je uvedeno v příloze „Příloha B“.



Obr. 6.6: Náhled do RTL souboru blokové šifry GOST.

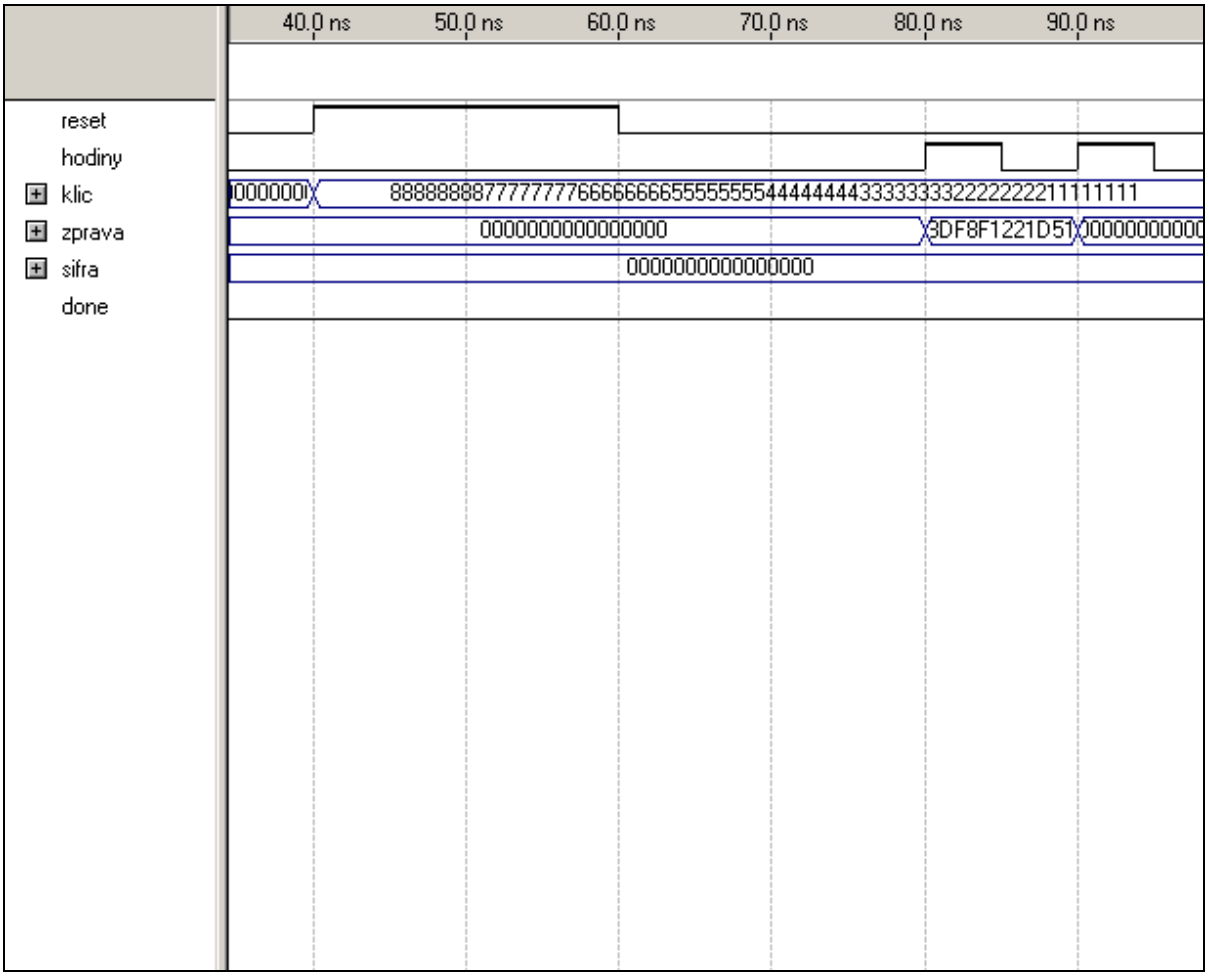
6.2.1 Ukázka šifrování

Klíč je načten v okamžiku úrovně „1“ signálu *reset* a zpráva je načtena při prvním „tiku“ hodin (první úroveň s hodnotou „1“ signálu *hodiny*), jak je vidět z Obr. 6.7. Z uvedeného obrázku lze také vyčíst vektor klíče (*klic*) a je vidět část vektoru vstupní zprávy (*zprava*). Pro přehlednost jsou hodnoty vyjádřeny v hexadecimálním tvaru a jsou tedy takovéto:

klic: 88888888777777776666666655555555444444443333333322222222211111111,

zprava: 1A8DF8F1221D51AE.

Signál *sifra* je prozatím správně nastaven na samé nuly.

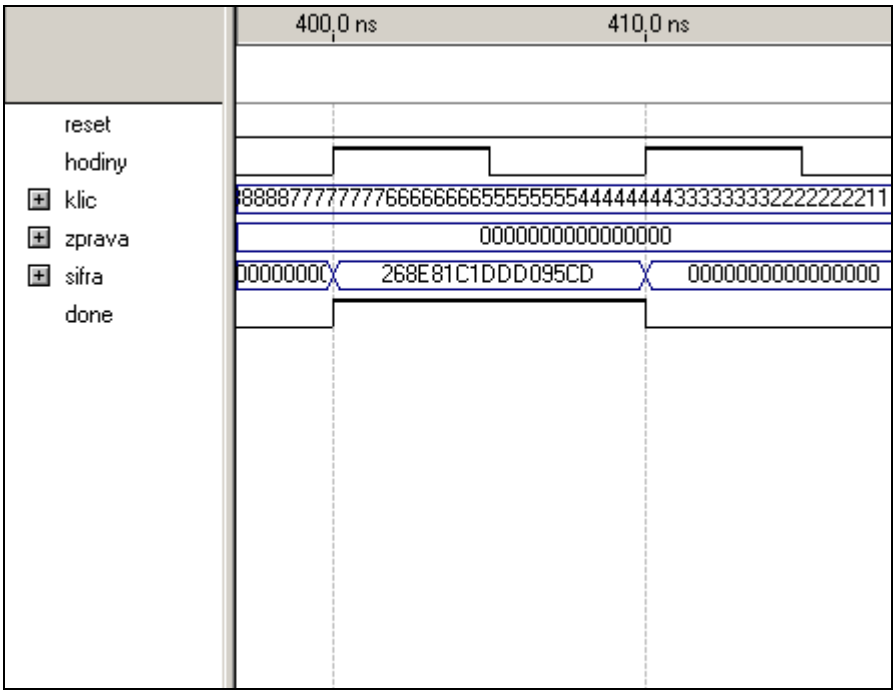


Obr. 6.7: Načtení klíče a zprávy v simulaci šifrování GOST.

Jak je vidět v Obr. 6.8, po provedení simulace je při hodnotě „1“ signálu *done* zobrazen v signálu *sifra* výsledný kryptogram 32 rund šifry GOST:

sifra: 268E81C1DDD095CD.

Mohlo by se zdát, že vstupní zpráva na Obr. 6.8 jsou samé nuly, ale není tomu tak, zpráva byla totiž v čase simulace 80-90ns načtena a uložena do registru a dále již není zobrazena.



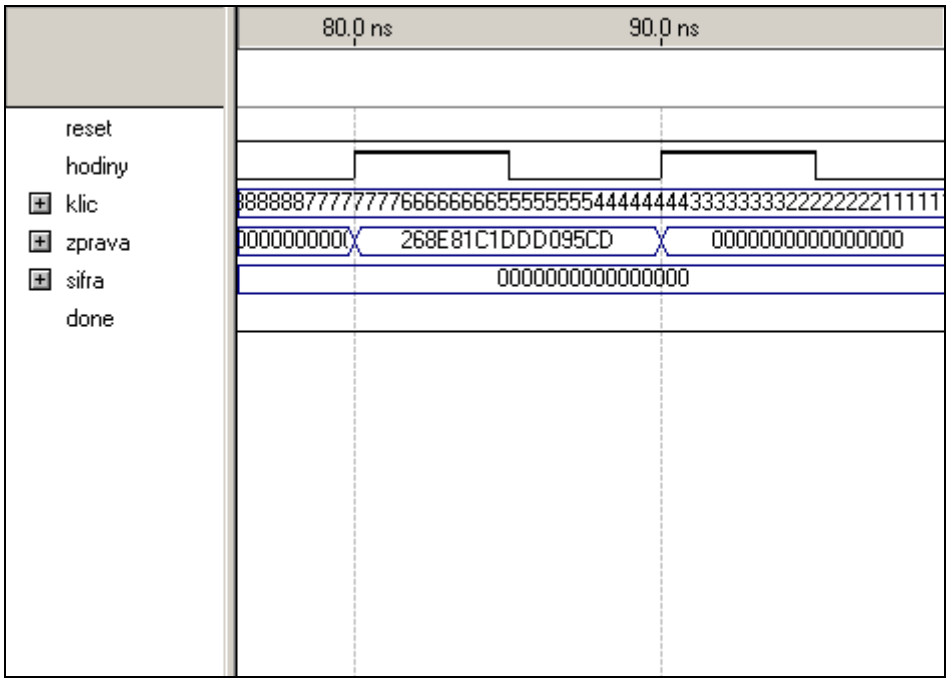
Obr. 6.8: Výsledek simulace šifrování GOST.

6.2.2 Dešifrování

Aby bylo ověřeno, že navrhnutá implementace šifry GOST zašifrovala vstupní zprávu správně, bylo nutné zprávu dešifrovat. Šifra GOST funguje jako dešifrátor v případě, že subklíče jsou k jednotlivým rundám přiřazovány v obráceném pořadí než při šifrování (viz. Tab. 6.3), jinak vše ostatní zůstává stejné. V Obr. 6.9 je ukázáno načtení klíče a kryptogramu.

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

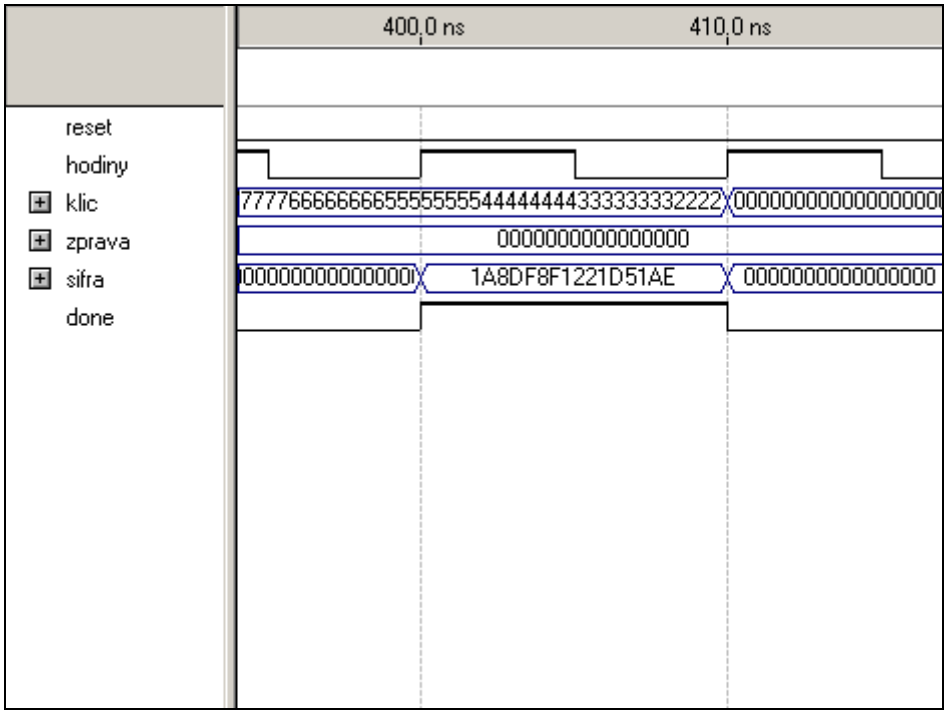
Tab. 6.3: Tabulka rundovních klíčů pro dešifrování.



Obr. 6.9: Načtení klíče a vstupní zprávy (kryptogramu) v simulaci dešifrování GOST.

Výsledek dešifrování je ukázán v Obr. 6.10. Je z něj patrné, že kryptogram byl rozšifrován správně na původní vstupní zprávu:

sifra: 1A8DF8F1221D51AE.



Obr. 6.10: Výsledek simulace dešifrování GOST.

Závěr

Cílem této práce bylo pojednat o šifrovacích algoritmech. Byl objasněn smysl a princip základních algoritmů. Byly popsány symetrické a asymetrické systémy se zjištěním, že asymetrické algoritmy jsou vhodné spíše k zajištění autentičnosti přenášených dat a symetrické k zajištění důvěrnosti dat, tedy šifrování dat. Kapitola 3 se zabývala druhy a režimy algoritmů, jako například nejčastěji používané CBC, CFB či OFB. V další kapitole byly prezentovány některé algoritmy používané v současné či nedávné době. U algoritmů DES a GOST byly porovnány jejich vlastnosti se závěrem, že DES má složitější způsob generování subklíčů a GOST má 256-bitový klíč, což je o 200 bitů víc než klíč DESu. Tyto dva algoritmy se ještě liší například v parametrech S-boxů a v neposlední řadě se liší počtem rund.

V kapitole šesté byly pomocí VHDL jazyku navrženy dvě šifry. První byla proudová šifra s generátorem LFSR, který se skládal z 32 posuvných registrů, což zajistilo dostatečnou periodu opakování generované pseudonáhodné posloupnosti. Touto šifrou byla zašifrována testovací zpráva o délce 32 bitů. Aby bylo možné ověřit správnost kryptogramu, byla navrhnuta stejná proudová šifra s generátorem LFSR v programu Matlab, kde byla zašifrována stejná testovací vstupní zpráva a výstupem byl stejný kryptogram. Tento kryptogram byl ještě v programu Matlab dešifrován a výsledkem byla původní testovací zpráva, což dokázalo, že proudová šifra ve VHDL byla navrhnuta správně. Druhou navrženou šifrou byla bloková šifra GOST. Podle standardu obsahuje 32 rund šifrování, v každé rundě sčítání modulo 2^{32}, s-box substituce, levotočivý posun a XOR. Pro přehlednost byla šifra v jazyce VHDL rozdělena do tří entit, které dohromady zahrnují veškeré výše uvedené operace provedené na vstupní zprávě 32-krát. Opět byla zašifrována testovací zpráva a výsledný kryptogram byl pro kontrolu dešifrován. Šifra GOST je postavena tak, že dešifrování se provádí stejným algoritmem jako šifrování jen s tím rozdílem, že pořadí subklíčů v rundách bylo otočené. Výstup z dešifrování se shodoval se vstupní zprávou šifrování, takže šifra GOST fungovala správně.

Seznam použité literatury a zdrojů

[1] SCHNEIER, Bruce. Applied Cryptography, Second Edition. John Wiley & Sons, N. York, 1996.

[2] BURDA, Karel. Bezpečnost informačních systémů. Elektronické skriptum FEKT VUT v Brně, 2005.

[3] KNUDSEN, L.R. Block Ciphers - Analysis, Design, Applications. Ph.D. dissertation, Aarhus University, listopad 1994.

[4] WILLIAM, Stallings. Cryptography and Network Security Principles and Practices. Prentice Hall, Fourth Edition, listopad 2005.

[5] RSA Laboratories' Frequently Asked Questions About Today's Cryptography, v4.0. RSA Data Security, Inc., 1998.

[6] KLÍMA, V., ROSA, T. Bezpečnost mobilních telefonů GSM: Důvěrnost a šifra v GSM. CHIP: magazín informačních technologií, září 1998, č 9, s. 148-151.

[7] JOYE, Marc, QUISQUATER, Jean-Jacques. Cryptographic Hardware and Embedded Systems – CHES 2004. 6th International Workshop Cambridge, MA, USA, August 11-13, Springer, 2004.

[8] RITTER, Terry. New Encryption Technologies for Communications Designers. Ciphers by Ritter. Ritter software engineering, 1999.

[9] PINKER, Jiří, POUPA, Martin. Číslicové systémy a jazyk VHDL. BEN – technická literatura, Praha 2006, 1. vydání. ISBN 80-7300-198-5.

[10] SAUL, Jonathan. Using SystemC and SystemCrafter to Implement Data Encryption. Xcell Journal, třetí čtvrtina 2006, str. 32-34.

[11] www.systemcrafter.com. Oficiální webová stránka SystemCrafter Ltd.

[12] Manuál k programu Quartus: Quartus II Handbook v9.0-Volume 1-Design and Synthesis-Section II. Design Guidelines, Alatera Corporation, březen 2009, <http://www.altera.com/literature/hb/qts/qts_qii5v1_02.pdf>.

[13] Tutoriál Quartus II Simulation with VHDL Designs.

[14] Manuál VHDL Reference manual. Sinario Design Automation, březen 1997.

[15] Oreku, Georgie, S. – LI, Jianzhong – Pazynyuk, Tamara – Tenzi, Fredrick, J. Modified S-box to Archive Accelerated GOST. IJCSNS International Journal of Computer Science and Network Security, červen 2007, roč. 7, č 6, s.88-98.

Seznam použitých zkratek

Zkratka	Popis
ASIC	Application-Specific Integrated Circuit
CBC	Cipher Block Chaining
CFB	Cipher-Feedback
CTR	Counter
DES	Data Encryption Standard
EASC	Euro-Asian Council for Standardization, Metrology and Certification
ECB	Electronic Codebook
EFF	Electronic Frontier Foundation
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection
GOST	Gosudarstvennyi Standard (Government Standard)
GSM	Global System for Mobile communications
HDL	Hardware Description Language
HW	hardware
IDEA	International Data Encryption Algorithm
IEEE	Institute of Electrical and Electronics Engineers
IPSec	Internet Protocol Security
LFSR	Linear Feedback Shift Register
OFB	Output-Feedback
PNP	pseudonáhodná posloupnost
RTL	Register Transfer Level
SSL	Secure Societ Layer

TCP	Transmission Control Protocol
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
WEP	Wired Equivalent Privacy
WLAN	Wireless Local Area Network
XOR	eXclusive OR

Seznam použitých symbolů a veličin

Symbol / veličina	**Popis**
B_i	výsledek i-té iterace
C, c	kryptogram
C	nelineární funkce u šifry A5, vektor kryptogramu
c	prostřední bit registru LFSR
D	operace dešifrování
D_K	dešifrovací operace za použití klíče *K*
E	operace šifrování
E_K	šifrovací operace za použití klíče *K*
f	funkce
G	registr LFSR
H	vektor hesla
IV	inicializační vektor
i, j	počitadla u šifry RC4
K, k, KP	klíč
L	levá polovina (např. zprávy, či výsledku operace)
M	zpráva, kterou hodláme zašifrovat (z angl. message)
m	velikost bloku zprávy
P	zpráva, kterou hodláme zašifrovat (z angl. plaintext)
R	pravá polovina (např. zprávy, či výsledku operace)
r	počet buněk registru
S	stav nezávislý na vstupní zprávě i na kryptogramu, S-box
X	subblok
Z	vektor zprávy

Přílohy

Použité programy a software

MS Office

Psaní a formátování textu, tvorba tabulek a generování obsahu byly provedeny v MS Office Word 2003 a MS Office Excel 2003.

MS Office Visio

Schémata vznikla v programu Microsoft Office Visio 2003.

Quartus II Version 9.0

Návrh v jazyce VHDL, analýza, syntéza, generování RTL souborů se schématy a simulace proběhly v prostředí Quartus II Version 9.0 Build 184 SJ Web Edition (Copyright © 1991-2009 Altera Corporation).

Matlab 7

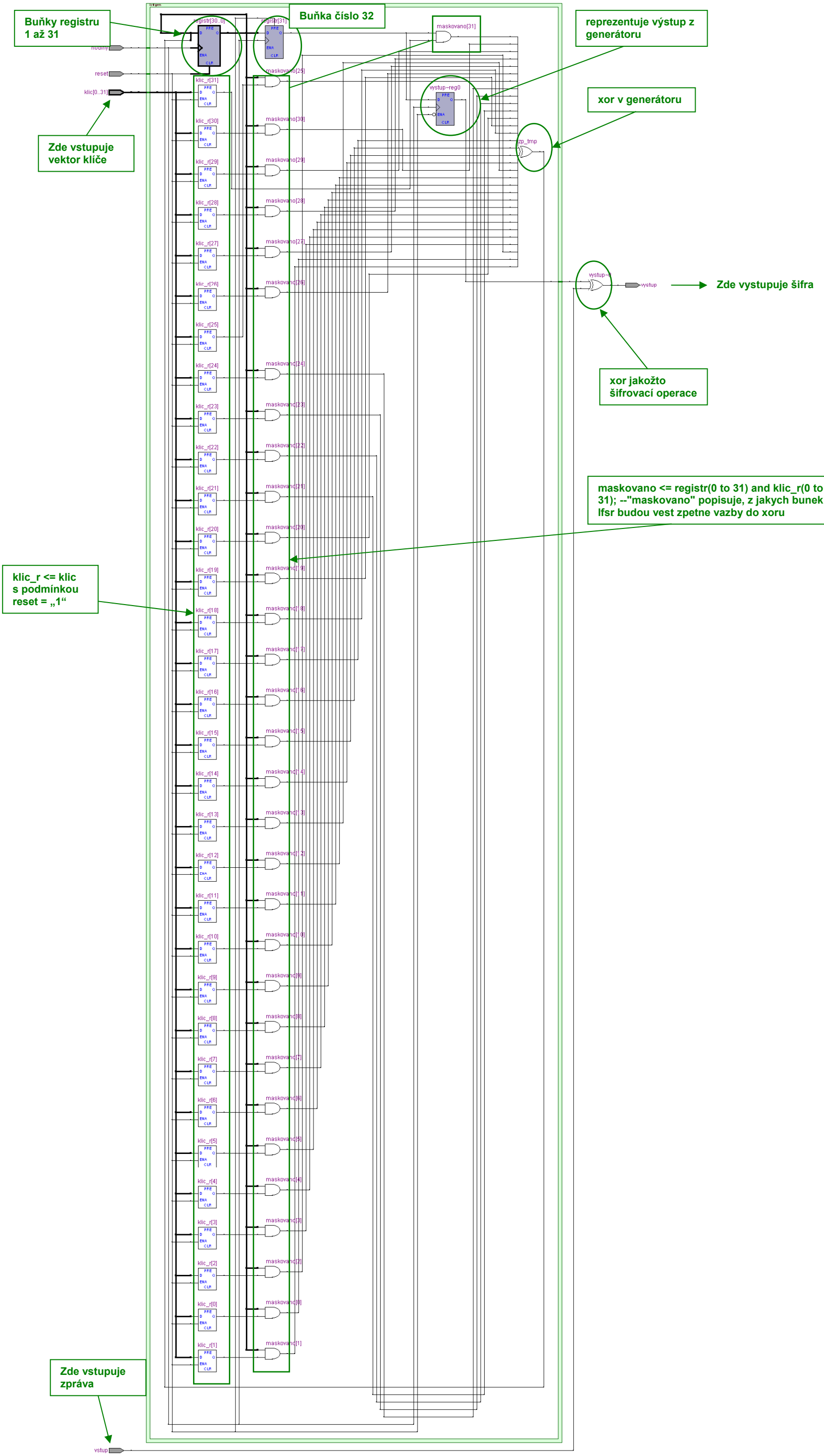
Návrh proudové šifry a generátoru PNP v prostředí Matlab 7.1 a Simulink 6.3.

Schémata hardwarových implementací šifer

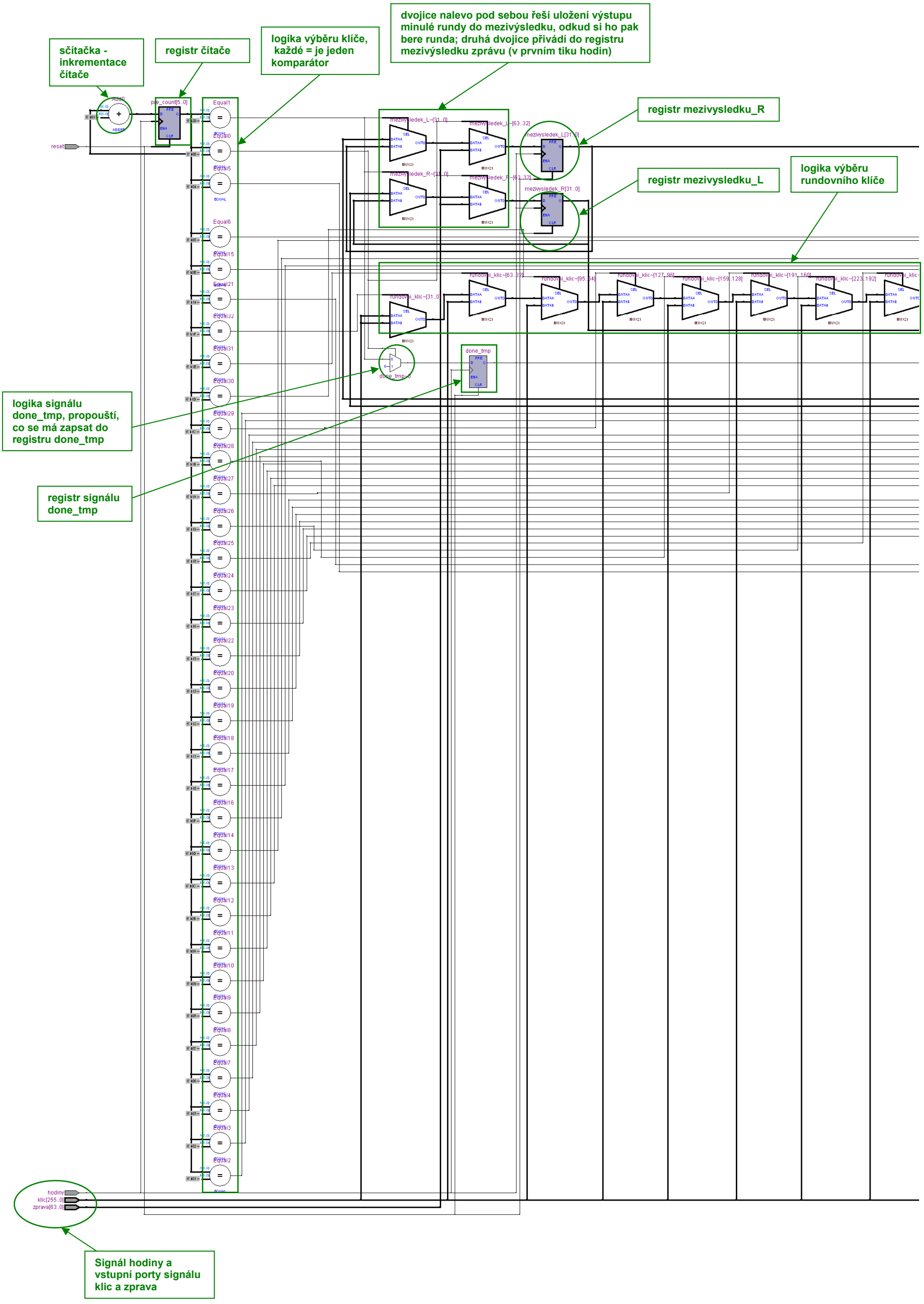
Příloha A - RTL schéma proudové šifry s generátorem LSFR.

Příloha B - RTL schéma blokové šifry GOST (rozdělené na 4 části).

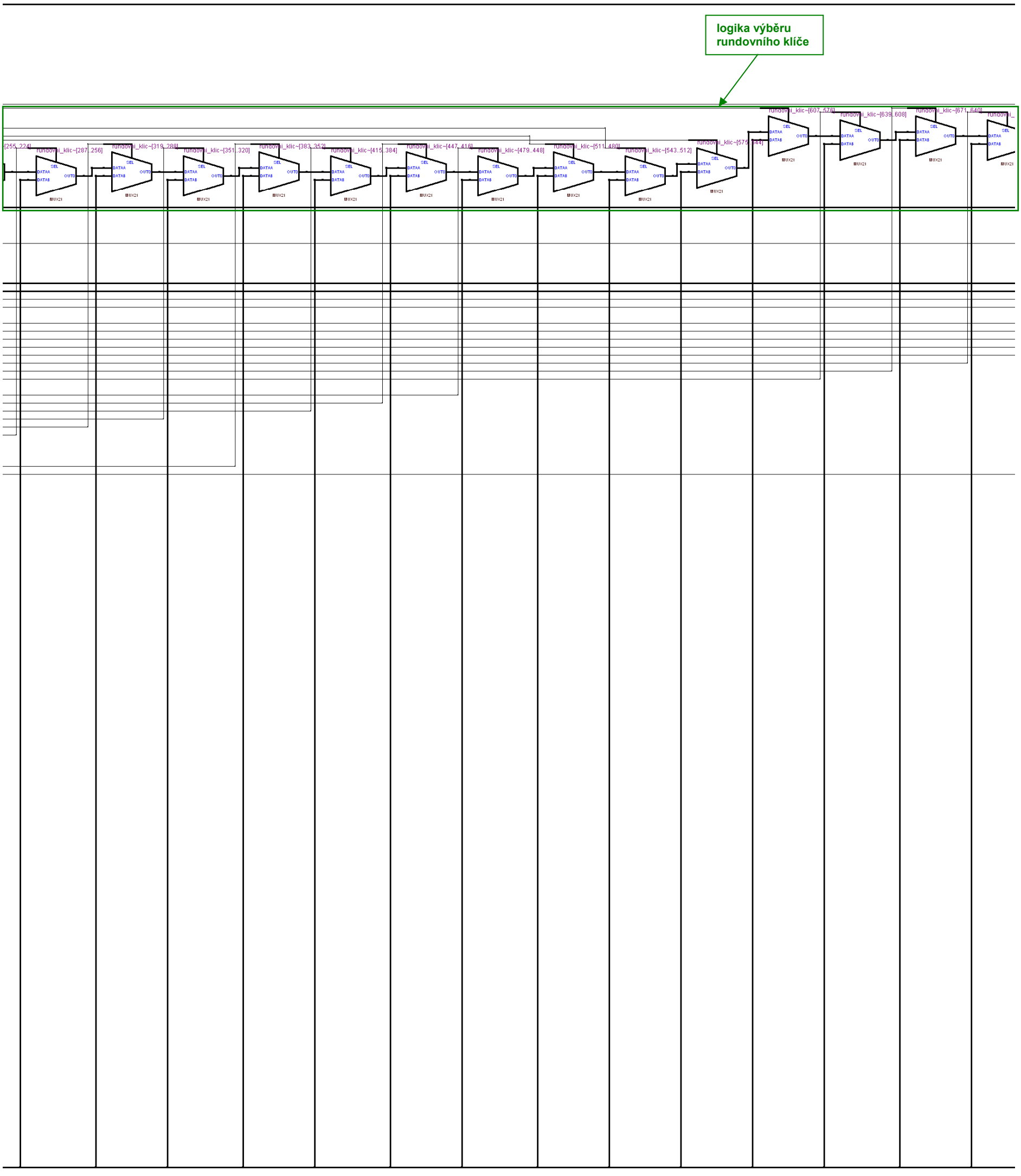
A - RTL proudová šifra



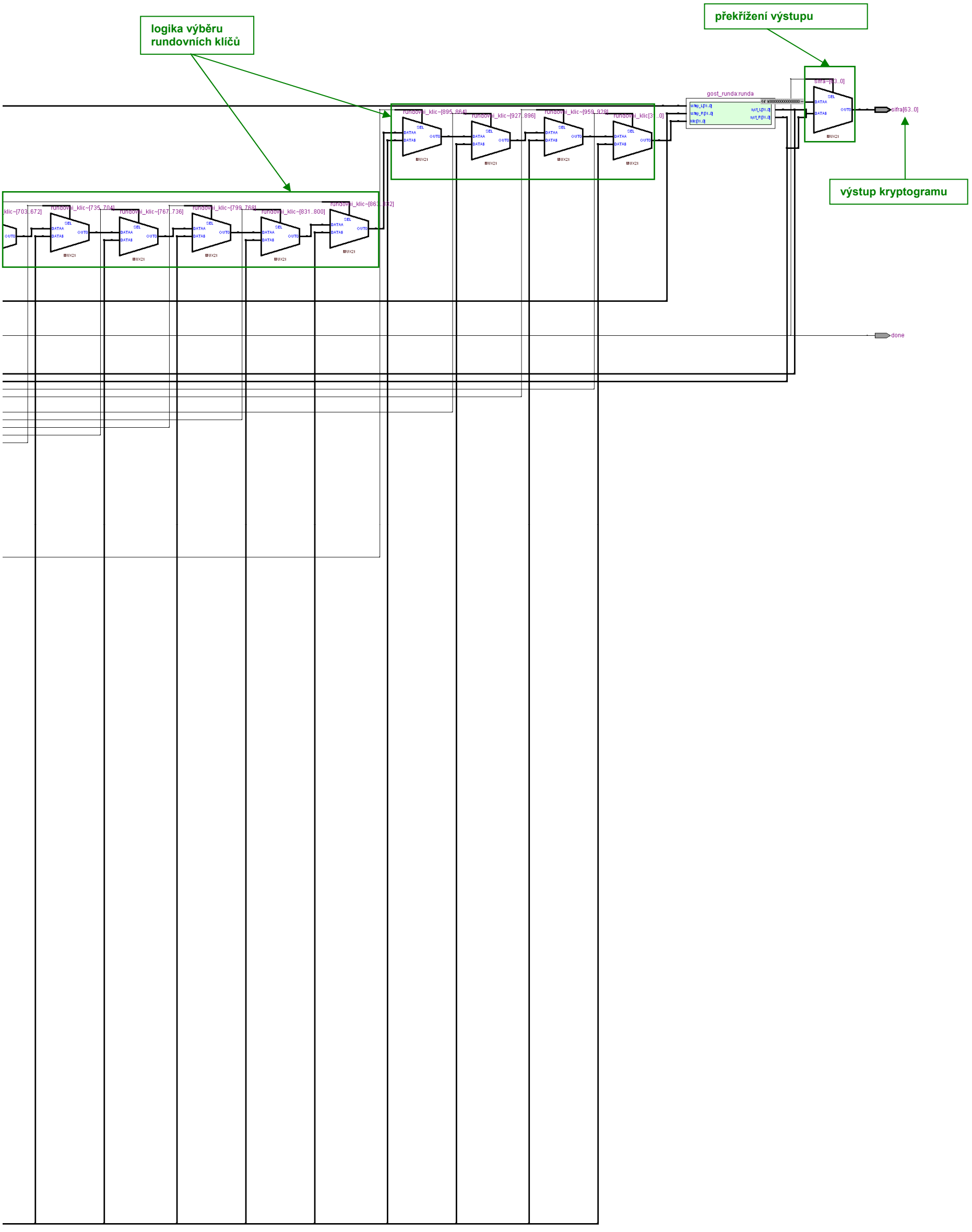
B.1 - RTL GOST první část



B.2 - RTL GOST druhá část



B.3 - RTL GOST třetí část



B.4 - RTL GOST čtvrtá část