VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

IMPLEMENTACE ŠIFROVACÍCH ALGORITMŮ NA PLATFORMĚ FPGA
IMPLEMENTATION OF CRYPTOGRAPHIC ALGORITHMS ON THE FPGA PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Adam Zugárek**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. David Smékal**
SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní obor **Teleinformatika**
Ústav telekomunikací

Student: Adam Zugárek ***ID:*** 173785
Ročník: 3 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Implementace šifrovacích algoritmů na platformě FPGA

POKYNY PRO VYPRACOVÁNÍ:

V rámci práce se student seznámí se síťovými kartami založenými na platformě FPGA. V teoretické části práce shrne problematiku symetrických šifrovacích algoritmů, seznámí se s programovacím jazykem VHDL, FPGA kartami COMBO, vývojovým frameworkem NetCOPE a platformou Xilinx Vivado.
V praktické části práce realizuje návrh programu pro možné šifrování síťového provozu pomocí vybraných šifrovacích algoritmů.

DOPORUČENÁ LITERATURA:

[1] STALLINGS, William. Cryptography and network security: principles and practice. 6. vyd. Upper Saddle River: Prentice Hall, 2013, 752 s. ISBN 01-333-5469-5.

[2] PINKER, Jiří, Martin POUPA. Číslicové systémy a jazyk VHDL. 1. vyd. Praha: BEN - technická literatura, 2006, 349 s. ISBN 80-7300-198-5.

Termín zadání: 1.2.2017 ***Termín odevzdání:*** 8.6.2017

Vedoucí práce: Ing. David Smékal
Konzultant:

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

ABSTRAKT

Tato bakalářská práce popisuje problematiku zabezpečení dat pomocí šifrování a autorovu vlastní implementaci na platformě FPGA. Cílem práce je implementace šifry na hardwarově akcelerované síťové kartě COMBO. Úvod práce je zaměřen na teoretický popis šifrování pomocí blokových šifer. Pro implementaci byla vybrána šifra AES, která je nejznámější používanou šifrou. Její detailní popis je popsán v první části této práce. V druhé části je popsána autorova vlastní implementace kódu šifry AES v jazyce VHDL. Dále je popsaná implementace a způsob, jakým je výsledný program propojen s frameworkem FPGA karty – NetCOPE. Závěrem práce jsou prezentovány dosažené výsledky. Výsledný program nešifruje síťovou komunikaci jako takovou, pouze zpracovává data uložená v hardwaru karty, které předává hostujícímu počítači.

KLÍČOVÁ SLOVA

FPGA, VHDL, NetCOPE, COMBO, AES, bloková šifra

ABSTRACT

This bachelor's thesis describes methods of data encryption and author's own implementation on FPGA. The goal of this thesis is to implement cipher on a hardware accelerated network card COMBO. In the introduction is described encryption using block ciphers. Cipher AES was chosen to implement, which is famous and most using cipher. Its detailed description is described in the first part of the thesis. In the second part is described the author's own implementation of AES cipher in VHDL. In the next part is method of interconnecting the resulting program with a framework of the FPGA card – NetCOPE. Achieved results are in the end of this thesis. The resulting program cannot encrypt network communication. It only transforms data stored in the card, which then send to host computer.

KEYWORDS

FPGA, VHDL, NetCOPE, COMBO, AES, block cipher

ZUGÁREK, A. Implementace šifrovacích algoritmů na platformě FPGA. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2017. 59 s. Vedoucí bakalářské práce Ing. David Smékal.

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma Implementace šifrovacích algoritmů na platformě FPGA jsem vypracoval samostatně pod vedením vedoucího bakalářské 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 bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne ....

(podpis autora)

Poděkování

Děkuji vedoucímu bakalářské práce Ing. Davidu Smékalovi za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady. A za podporu při zpracování mé bakalářské práce.

V Brně dne **8. června 2017** ……

podpis autora

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

V současném digitálním světě probíhá téměř veškerá komunikace pomocí digitálních telekomunikačních systémů, kterými prochází nespočetné množství informací. Tyto komunikační prostředky se neustále vyvíjí a vznikají nové služby, které ulehčují naši práci a každodenní život. Aby používání těchto služeb (hlavně služby komunikující přes internet) bylo užitečné, je potřeba do jisté míry zaručit integritu dat (manipulace s daty po přenosové trase), autentičnost odesílatele (ověření odesílatele) a v některých případech také utajení informací (zabránění třetím osobám přečíst si zprávu). Pro tento účel vznikla a neustále se vyvíjí řada šifer a algoritmů, které poskytují různé formy a úrovně zabezpečení. S tím se také neustále vyvíjí útoky a metody k překonání těchto bezpečnostních opatření. To také žene vývoj nových opatření. Je to neustálá snaha překonat toho druhého. K vývoji nových a lepších šifer kromě nových útoků také přispívá neustálé zrychlování výpočetních zařízení – nárůst výpočetní síly a rychlosti internetu. Technologie Ethernet dosahuje rychlosti 100 Mbit/s a už se pracuje na rychlosti 400 Mbit/s.

Tato práce popisuje problematiku symetrických šifer a zaměřuje se na detailní popis a implementaci šifry AES, která byla vybrána autorem, protože se jedná o nejznámější a nejpoužívanější šifru pro zabezpečení dat. Problematika symetrických šifer je popsána v kapitole č. 1 společně s FPGA technologií a platformou NetCOPE.

Hardwarově akcelerovaná karta COMBO byla vybrána z toho důvodu, že je osazena čipem FPGA, který umožňuje paralelní zpracování dat, a tak v tomto případě urychluje proces šifry AES. Mimo jiné je čip umístěn přímo na kartě, takže je možné zpracovávat data přímo na kartě. Na rozdíl od síťových karet, které nedisponují čipem FPGA a šifrování dat je tak prováděno na procesoru hostitelského počítače, nedochází ke snížení rychlosti v důsledku posílání a zpracování dat procesorem počítače. Navíc nedokáže jádro procesoru zpracovávat data paralelně jako FPGA.

Variantou k FPGA jsou čipy ASIC, které ale nelze přeprogramovat a plní pouze jednu funkci na rozdíl od FPGA.

Kapitola č. 2 obsahuje teoretický popis šifry AES. Popisuje se zde rozložení a práce s daty a matematické funkce, které jsou použity v jednotlivých transformacích šifry. Kromě jednotlivých transformací je zde popsán proces šifrování, dešifrování a generování podklíčů.

V první polovině kapitoly č. 3 jsou popsány implementace transformace z kapitoly č. 2 v jazyce VHDL. Následující kapitoly pak popisují zakomponování jednotlivých částí do jednoho funkčního celku, který se chová jako „černá skříňka[1]“. V závěru této kapitoly je v simulačním prostředí popsána funkce programu a jeho částí a jsou prezentovány dosažené výsledky.

[1] Černá skříňka (Black box) je označení pro objekt, který provádí nějakou činnost a tato činnost je skryta všem mimo obsah skříňky. Se svým okolím komunikuje pomocí rozhraní. Např. auto. Vnitřní části auta (motor, čerpadlo, …) a jejich funkce jsou uživateli skryty a uživatel ovládá auto pouze pomocí pedálů, řadící páky, atd.

1 ZÁKLADNÍ POJMY

Pro šifrování dat je potřeba šifra, která bude odolná proti prolomení, nebo její prolomení bude trvat tak dlouho, že zašifrované informace se stanou bezcennými. Vytvářením takových šifer se zabývá kryptografie a jejich prolamováním potom kryptoanalýza.

Šifra popisuje algoritmus, který má *n* počet úkonů – transformací, které se provádí nad množinou dat. Takto se vstupní data (zpráva) transformují z čitelné podoby (plaintext) do nečitelné. Data jsou nyní zašifrovaná a zabezpečená. A záleží na složitosti šifry, jestli a kdy bude možné šifru prolomit, a zabezpečená data tak dešifrovat.

Pro šifrování a dešifrování se používá šifrovací klíč. Ten může a nemusí být tajný, záleží na typu šifry. Některé šifry vyžadují, aby byl šifrovací klíč držen v tajnosti. Tento klíč vstupuje společně se vstupními daty do procesu šifrování, kde může být různě transformován a přidáván ke vstupním datům. Pro každý klíč mají zabezpečená data jinou podobu.

1.1 Šifry

Šifry můžeme dělit podle způsobu, jakým zpracovávají data. To jsou proudové a blokové šifry. Anebo podle použitých klíčů na symetrické a asymetrické šifry.

Asymetrické šifry používají dva různé klíče. Jeden pro šifrování zpráv, označován jako veřejný klíč, a druhý pro dešifrování, označován jako privátní klíč. U těchto klíčů je kladen velký důraz na to, aby nebylo možné z jednoho klíče odvodit nebo odhadnout druhý. [1]

Komunikace za pomocí asymetrické šifry probíhá následovně. Mějme dva účastníky zabezpečené komunikace – příjemce a odesílatele. Oba dva si vygenerují své vlastní privátní a veřejné klíče. Příjemce zveřejní svůj veřejný klíč. Kdokoliv bude chtít zaslat šifrovanou zprávu příjemci, stáhne si příjemcův veřejný klíč a pomocí něj pak tuto zprávu zašifruje. Jedinou možností dešifrování této zprávy bude pouze pomocí privátního klíče příjemce. Ten musí svůj privátní klíč uchovávat v tajnosti. Šifrování pomocí asymetrických šifer je pomalejší než při použití symetrické šifry, avšak výhodou oproti symetrickým šifrám je jednoduší distribuce klíčů. Asymetrické šifry se používají při ověřování pomocí digitálních podpisů: RSA[2], DSA[3] a šifry, využívající eliptické křivky. [1]

Symetrické šifry na rozdíl od těch asymetrických používají stejný klíč pro šifrování i dešifrování zpráv. Anebo takové dva klíče, které jsou odvoditelné navzájem. Rychlost šifrování je vyšší než u asymetrických šifer, ale nastávají komplikace při distribuci klíče. Odesílatel musí předat příjemci klíč, aniž by byl zachycen nebo prozrazen třetí straně. [1]

---

[2] RSA je šifra sloužící k zabezpečení dat při komunikaci využívající system používající veřejný klíč. Název RSA jsou iniciály autorů - R.L. Rivest, A. Shamir, and L. Adleman https://people.csail.mit.edu/rivest/Rsapaper.pdf

[3] DSA (Digital Signature Algorithm) je algoritmus pro digitalní podpis vyvinutý americkou agenuturou NIST[8] v dokumentu FIPS 186. http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

Symetrické šifry lze pak dělit na proudové a blokové. Proudové šifry zpracovávají vstupní zprávu bit po bitu společně s šifrovací posloupností. Zdrojem takové posloupnosti jsou obvykle generátory pseudonáhodných posloupnosti. Na straně příjemce a odesílatele jsou potom stejné generátory, které se nastaví do počátečního stavu klíčem. Díky tomu dojde ke generování stejné šifrovací posloupností na obou koncích. Každý *i*-tý bit zašifrované zprávy je pak výsledkem operace XOR *i*-tého bitu původní zprávy a šifrovací posloupnosti. Zašifrovaná zpráva je pak odeslána příjemci, který opět provede operaci XOR s šifrovací posloupností a dostane původní zprávu. [1]

Blokové šifry zpracovávají *n* bitů zprávy o konstantní délce. Výstupem je pak blok zašifrované zprávy, kde každý jednotlivý bit na výstupu závisí na všech *n* bitech na vstupu najednou. Tato vlastnost se označuje jako difuze. Podobnou vlastností je konfuze a ta označuje stav, kdy výstup závisí na každém bitu klíče. Návrh blokové šifry by měl pak splňovat obě vlastnosti, aby byla ztížena kryptoanalýza. [1]

Koncept většiny šifer je navržen jako kaskáda šifer. Jde o spojení více šifer za sebou tak, že výstupní blok předchozí šifry je předáván na vstup následující šifry. Většinou se používá návrh, kdy je za sebou několik stejných šifer – iterovaná šifra. Zpráva se šifruje při každé iteraci tzv. iteračním klíčem. Ten je odvozený z původního klíče pomocí expanze klíče. Každá iterace má svůj vlastní klíč. Dešifrování se pak provádí pomocí inverzních funkcí k šifrovacím funkcím. A také použitím opačného pořadí iteračních klíčů. [1]

Důležité je také to, že délku zprávy může být potřeba před šifrováním upravit na požadovanou délku. Pokud se zpráva rozdělí na *n*-bitové úseky, může se stát, že na konci zprávy vznikne úsek menší než *n*. Takový úsek je třeba doplnit na délku *n*. Toho se docílí tak, že za poslední bit posledního úseku se zapíše bit s hodnotou 1 a dále se za něj zapisují bity s hodnotou 0, než se dosáhne požadované délky *n*. Takto upravená zpráva se zašifruje, odešle příjemci, který ji na druhé straně dešifruje. Pokud zpráva po dešifrování končí 0, tak příjemce prochází zpětně zprávu, dokud nenarazí na bit s hodnotou 1. Celý tento úsek včetně bitu s hodnotou 1 odstraní a dostane tak původní zprávu. Pokud délka zprávy je násobkem délky úseku *n*, nevznikne na konci žádný přebytek a je potřeba tuto zprávu rozšířit o celý úsek o délce *n*, který bude začínat 1 a dále pokračovat samými 0. [1]

U blokových šifer se pracuje především blokem vstupních dat, který je rozdělen na bajty. Blok se doplňuje celými bajty do plna. Každý takový bajt má uloženou hexadecimální hodnotu počtu doplněných bajtů. Šifra, která by měla délku bloku 10 bajtů a se zprávou o délce 6 bajtů, by se pak doplnila čtveřicí bajtů, každý s hexadecimální hodnotou {04}. [1]

Blokové šifry se provozují v několika různých operačních režimech. Tyto režimy slouží k vylepšení vlastností šifry a ztížení kryptoanalýzy. Každý režim se liší dle potřeby nasazení. V následujícím textu je předpokládáno hotové zarovnání délky zprávy na *n*-tice požadované délky. [1]

1.1.1 Režim ECB – Elektronická kódová kniha

Režim ECB (Electronic Codebook) je základní režim, který používá šifru přímo, jak je nadefinovaná. Každý blok zprávy se zašifruje nezávisle na ostatních. Příjemce pak bloky dešifruje a spojí za sebe. Dostane tak původní zprávu. Tento režim je jednoduchý, avšak nevýhodou je stav, kdy se opakují stejné bloky zpráv. Pokud jsou dva zašifrované bloky stejné, musí být stejné i původní bloky. Ale jestli se tento režim použije pro šifrování zpráv, které neobsahují opakující se bloky, ničemu to nevadí. Například přenos klíčů. [2]

1.1.2 Režim CBC – řetězení šifrových bloků

Režim CBC (Cipher-Block Chaining) využívá předešlé zašifrované bloky k úpravě následujících bloků. Na začátku je náhodný vektor, který se exkluzivně sečte s prvním blokem zprávy a následně se zašifruje. Výsledný blok se opět exkluzivně přičte k dalšímu bloku zprávy a zašifruje. Tím je zajištěno, že dva stejné bloky zpráv nebudou mít s vysokou pravděpodobností stejné zašifrované bloky. Režim je závislý na chybovosti přenášejícího média. Jediná chyba znehodnotí půlku bitů v bloku. [2]

1.1.3 Režim CTR – čítač

Režim CTR (Counter Mode) se používá v komunikačních kanálech s vyšší chybovostí, jako jsou rádiové spoje, kde režimy ECB a CBC jsou nevhodné. Zašifrovaný blok se vytváří exkluzivním sečtením bloku zprávy s výstupem šifry, kde jako vstupní hodnoty jsou *n*-bitové číslo čítače a klíč. Hodnota čítače se cyklicky zvyšuje o hodnotu jedna. Hodnotu není potřeba tajit, takže lze přenášet spolu se zašifrovanou zprávou. Ale je potřeba zajistit, aby se po celou dobu používání jednoho klíče, tato hodnota neopakovala. Tento režim funguje jako proudová šifra. Bloková šifra se používá pro vygenerování náhodné posloupnosti a ta se přičte XORem ke zprávě. Zpráva tak nemusí být zarovnaná. Pokud zpráva bude mít délku *d* menší, než je délka bloku, stačí použít *d* bitů z výstupu blokové šifry. Z toho plyne výhoda oproti předchozím režimu. Pokud se správně přenese počáteční hodnota čítače, bloková šifra na straně příjemce generuje totožné hodnoty, které generoval i odesílatel. Tento výstup pak exkluzivně přičte k zašifrované zprávě a dostane původní zprávu. Pokud dojde při přenosu k chybě na *i*-tém bitu, chyba se po dešifrování projeví na *i*-té pozici a dále se nepřenáší, neboť tento proces dešifrování se provádí proudovou šifrou a ne blokovou. Nevýhodou je útok na autentičnost zprávy, proto by se měl tento režim používat s další autentizační technikou. [3]

Zástupci blokových šifer jsou například Rijndael[4] , Mars[5], Serpent[6], Twofish[7] a RC6[8].

[4] Rijndael je šifra, na které je postavený standard AES (viz kap. 2). http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf

[5] Mars - http://www.nada.kth.se/kurser/kth/2D1449/99-00/mars.pdf

[6] Serpent -. http://jerli.info/docu/Master_2/Crypto/serpent.pdf

[7] Twofish - https://www.schneier.com/academic/paperfiles/paper-twofish-paper.pdf

[8] RC6 - https://people.csail.mit.edu/rivest/pubs/RRSY98.pdf

1.2 FPGA

FPGA (Field Programmable Gate Array) jsou polovodičová zařízení sestávající se z logických bloků, které lze programovat a propojovat je mezi sebou dle potřeby. Tím je možné je používat v řadě různých aplikací. Na rozdíl od ASIC (Application Specific Integrated Circuit), to jsou integrované obvody, které jsou přímo navržené pro konkrétní použití, lze FPGA přeprogramovat v cílovém zařízení i za chodu. Je to výhoda pro návrh aplikace, kdy není zapotřebí pro každou implementaci vytvářet nový integrovaný obvod. Největší výrobci FPGA obvodů jsou Xilinx, Altera, Atmel a Actel. [4]

Základním stavebním blokem FPGA je LUT tabulka (Look-Up Table). Je to generátor logických funkcí s pamětí. Do paměti se uloží požadovaný vstupní a výstupní vektor signálu, kdy potom při přijetí vektoru na vstup paměti paměť vygeneruje výstupní vektor, který byl uložen pro daný vstup. Tímto se simulují výstupy z kombinační logiky. Výhodou je konstantní zpoždění procházejícího signálu pro různé kombinace logických funkcí.

Pro programování nebyly dříve jazyky, a tak se musely hodnoty zapisovat přímo do pamětí. Pro zlehčení práce vznikly jazyky pro popis číslicových obvodů. [5]

1.3 HDL Jazyky

HDL (Hardware Description Language) jsou jazyky popisující chování číslicových obvodů. Dokáží simulovat kód před jeho implementací na cílový čip. To umožňuje vytvářet a testovat různé implementace řešení problému. Dále pomocí syntézy lze zjistit, jestli daný kód bude schopný běžet na cílovém zařízení. [6]

Existují dva nejpoužívanější HDL jazyky: VHDL a Verilog. VHDL se používá více v Evropě a Verilog naopak více v USA, ale není to vždy pravidlem [5]. VHDL je jazyk vyšší úrovně, který byl vytvořen ministerstvem obrany USA v rámci projektu VHSIC (Very High Speed Integrated Circuits). Vývoj začal v roce 1981. VHDL je do jisté míry nezávislý na číslicovém zařízení nebo na technologii, jakou bylo vytvořeno. To umožňuje do jisté míry i přenositelnost kódu. Definice jazyku je dlouhá přibližně 300 stran. Ale ne všechny konstrukce a příkazy je možné syntetizovat. Po navržení kódu se provádí syntéza. To je kompilace kódu VHDL a následný pokus aplikace na cílovou technologii čipu FPGA nebo na jiné zařízení (ASIC). Pokud je kód možné syntetizovat, bude návrh fungovat na zařízení. Kód, který nelze syntetizovat, lze spouštět pouze v simulaci. Možnosti syntézy se odvíjí o syntetizátoru konkrétního výrobce. Postupem času se ale možnosti syntetizátorů rozšiřují. [5]

Programování je rozdílné od jiných vyšších jazyků, například C, protože VHDL a číslicová zařízení zpracovávají příkazy souběžně, neprovádí sekvenční zpracovávání signálů. Na příklad do zařízení vstupuje deset různých signálů, ty zpracuje zároveň a všechny vrátí v jeden moment. Sekvenční programování je možné, ale není nejjednodušší, a navíc snižuje výkon a efektivitu čipu. [5]

1.4 NetCOPE

NetCOPE platforma vznikla k projektu Librerouter, jehož cílem bylo monitorování IPv6 sítí a urychlení směrování v IPv6 sítí s hardwarově akcelerovanými kartami COMBO, které jsou osazené FPGA čipy. Tento projekt vznikl v roce 2003 pod vedením CESNETu, za spolupráce VUT Brno, ČVUT Praha a Masarykovy Univerzity. Nyní nabízeno společností NetCOPE Technologies (dříve INVEA-TECH). [7]

Platforma NetCOPE umožňuje vývoj softwaru pro síťové karty rodiny COMBO. Podporuje rychlosti 1G, 10G, 40G, 80G a 100G. Platforma může být využita pro vytvoření síťových karet, monitorování sítě, detekci vniknutí do sítě, IPv4 a IPv6 router, kryptografii nebo zabezpečení videa. [8]

2 TEORETICKÝ POPIS ŠIFRY AES

Tato kapitola se věnuje teoretickému popisu průběhů a transformací šifry AES. V roce 2001 byl schválen zabezpečovací standard (Advanced Encryption Standard – AES) americkým ministerstvem obchodu a průmyslu, který spravují dále dvě agentury – NIST[9] a ITL[10]. AES je publikován pod číslem publikace FIPS PUB 197 [9]. Pro tento standard byla ve druhém kole výběrového řízení vybrána šifra Rijndael, jedna z pěti šifer – Twofish, Serpent, RC6 a MARS.

Šifra Rijndael je symetrická bloková šifra, která umí zpracovávat různé délky bloku a délky klíčů. Pro standard AES byla vybrána délka bloku 128 bitů a délky klíče 128, 192 a 256 bitů.

V této práci bude dále pojem AES označovat šifru Rijndael. Pro délky klíče pak budou užívány pojmy AES-128, AES-192 a AES-256.

2.1 Úvod

Jako vstup a výstup šifry budou bloky – sekvence bitů o délce přesně 128 bitů. Klíč je sekvence bitů o délce 128, 192 nebo 256 bitů. Jiné délky sekvencí standard nepřipouští.

V následujícím textu nejsou popsány detailně všechny matematické úkony prováděné v šifře, ale použité definice (operace, transformace, rozložení bitů, bajtů a jejich pojmenování) jsou voleny záměrně stejně nebo aspoň podobně, aby nedocházelo k rozporům mezi touto prací a standardem AES.

2.1.1 Reprezentace dat

Jednotkou dat je v šifře AES blok. Tento blok dat se dělí na bajty (skupina 8 bitů). To vytvoří celkem 16 bajtů. Bajty jsou uspořádány v tabulce 4×4 a uspořádání se nazývá „stavové pole“ (The State array) – viz následující podkapitola. Rozdělení bitů vstupního bloku je na obrázku č. 1 společně s rozmístěním bajtů v tabulce.

Bity v bajtu jsou uspořádány sestupně tak, že bit umístěný vlevo je bit s nejvýznamnější hodnotou (MSB) a bit vpravo je nejmíň významnou hodnotou (LSB) – $\{b_7, b_6, b_5, b_4, b_3, b_2, b_1, b_0\}$. To lze vyjádřit i pomocí polynomu:

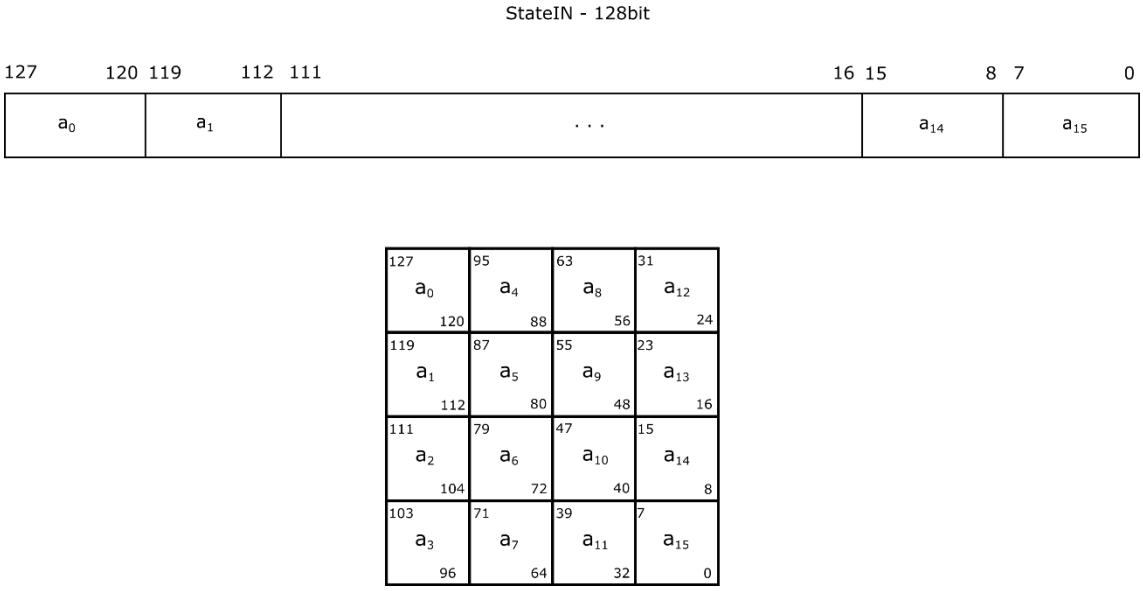
$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 \tag{2.1}$$

Například {01100101} lze zapsat polynomem $x^6 + x^5 + x^2 + 1$

Bajt se označuje písmenkem a s příslušným indexem a lze jej zapsat také za použití dvou hexadecimálních číslic. Vstupní blok lze pak zapsat jako řetězec bajtů $a_0a_1a_2\ldots a_{15}$.

[9] NIST – National Institute of Standards and Technology – je Národní institut standardů a technologií v USA (měřící laboratoř) pod ministerstvem obchodu USA. https://www.nist.gov/

[10] ITL – Information Technology Laboratory – jedná se o IT výzkumnou laboratoř, která vytváří standardy a testy pro zlepšení a zabezpečení informačních systémů. https://www.nist.gov/itl



Obrázek 1: Stavové pole – rozvržení bajtů a bitů

2.1.2 Stavové pole (The State array)

Stavové pole má délku 16 bajtů a reprezentuje blok vstupních dat. Bajty ve stavovém poli jsou označené jako dvourozměrné pole s indexy *r* a *c*, kde oba nabývají hodnot 0 až 3. Index *r* označuje řádek pole a *c* označuje sloupec. Každý sloupec stavového pole vytváří slovo o délce 32 bitů, kde index *c* označuje index slova w_c.

Stavové pole slouží jako dočasná proměnná (paměť), která prochází jednotlivými transformacemi a iteracemi.

2.2 Matematické funkce

Tato kapitola se zabývá matematickým popisem úkonů, které jsou použity při jednotlivých transformacích. Podrobnější popis a vysvětlení matematických úkonů je k nalezení v [9]. Tento text používá záměrně vztah mezi bajtem a polynomem, který byl jíž dříve definován, aby definoval a ukázal, jakými způsoby se manipuluje s daty. Stejně je tomu u polynomů, které reprezentují bajty. To se týká redukčního polynomu *m*(*x*) a funkce *xterm*, která se používá při násobení dvou bajtů (příklad násobení je uveden níže v textu). Násobení bajtů se používá výhradně v transformaci *MixColumns* a její inverzi, a také při výpočtu hodnoty *Rcon* při expanzi klíče.

2.2.1 Sčítání a násobení

Operaci sčítání provádí operace XOR – exkluzivní součet. Jedná se o matematickou operaci modulo 2. Sčítání dvou bitových řetězců se provádí po jednotlivých bitech.

Příklad 1:

$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x2$

$\{01010111\} \oplus \{10000011\} = \{11010100\}$

$\{57\} \oplus \{83\} = \{d4\}$

Při násobení se v polynomiální podobě postupuje tak, že se dva polynomy vynásobí a následně zmodulují nerozložitelným polynomem osmého stupně. Pro AES je takovým polynomem

$$m(x) = x^8 + x^4 + x^3 + x + 1. \tag{2.2}$$

Příklad 2: $\{57\} \cdot \{83\} = \{c1\}$

$$(x^6 + x^4 + x^2 + x + 1)\,(x^7 + x + 1) = x^{13} + x^{11} + x^9 + x^8 + x^7 + x^7 + x^5 + x^3 + x^2 + x + x^6 + x^4 + x^2 + x + 1$$

$$= x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1$$

a

$$x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \bmod (x^8 + x^4 + x^3 + x + 1) = x^7 + x^6 + 1$$

Polynomem $m(x)$ pak můžeme upravit jakékoli číslo, které je většího řádu jak 7 a nelze vyjádřit pomocí bajtu.

Násobením polynomu číslem x vyjde polynom s řádem o jedna vyšší, než je původní. To lze vyjádřit pomocí bitové operace posun vlevo o jeden bit. Pokud byl použit 8. bit, tímto posunutím se dostane na pozici 9. bitu a vzniklý výraz nelze vyjádřit pomocí jednoho bajtu. V této fázi se provede redukce polynomem $m(x)$ a výsledkem je číslo vyjádřitelné jedním bajtem.

Tuto vlastnost využívá funkce *xtime()* použita při násobení. Funkce násobí číslem x – $\{02\}$. Opakovaným použitím lze pak dosáhnout součinu různých činitelů.

Příklad 3: $\{57\} \cdot \{13\} = \{fe\}$, neboť

$\{57\} \cdot \{02\} = \text{xtime}(\{57\}) = \{ae\}$

$\{57\} \cdot \{04\} = \text{xtime}(\{ae\}) = \{47\}$

$\{57\} \cdot \{08\} = \text{xtime}(\{47\}) = \{8e\}$

$\{57\} \cdot \{10\} = \text{xtime}(\{8e\}) = \{07\}$

$$a\ \{57\} \cdot \{13\} = \{57\} \cdot (\{01\} \oplus \{02\} \oplus \{10\})$$
$$= \{57\} \oplus \{ae\} \oplus \{07\}$$
$$= \{fe\}$$

2.3 Definice algoritmu šifry

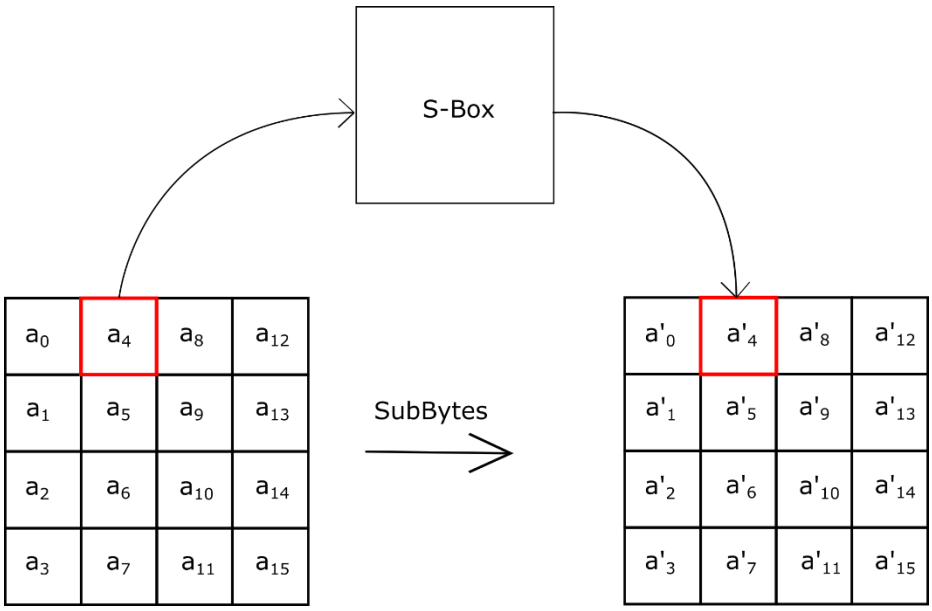
Pro AES je stanovena délka vstupního bloku 128 bitů. Délka klíče 128, 192 nebo 256 bitů. Počet iterací šifry, který pro délky klíče 128 bitů je 10, pro 192 bitů je 12 a pro 256 bitů je 14 iterací.

Na začátku šifrování se zkopíruje vstupní blok do stavového pole a provádí se průchod 4 transformacemi jedné iterace - *SubBytes(), ShiftRows(), MixColumns()* a *AddRoundKey()* - až po předposlední iteraci. Poslední iterace neprovádí transformaci *MixColumns()*.

2.3.1 SubBytes()

Je transformace (obrázek č. 2), která provádí substituci každého bajtu za použití substituční tabulky S-box (Tabulka 1). Substituce lze matematicky vypočítat, anebo si předem uložit nebo vygenerovat celou tabulku do paměti, a přistupovat k ní způsobem tak, že první 4 bity

bajtu udávají číslo řádku a druhé 4 bity udávají číslo sloupce – {xy}.



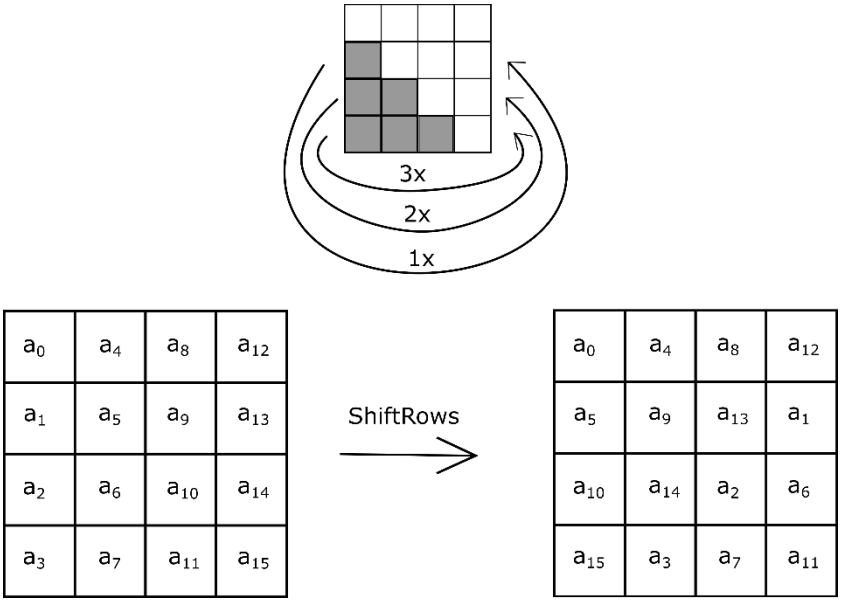
Obrázek 2: Transformace SubBytes

Tabulka 1: S-box

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

2.3.2 ShiftRows()

Tato transformace provádí rotaci bajtů vlevo po řádcích stavového pole (Obrázek 3). Počet rotací je určen indexem řádku začínající od 0. To znamená, že s každým řádkem roste počet rotovaných bajtů.



Obrázek 3: Transformace ShiftRows

2.3.3 MixColumns()

Tato transformace pracuje se sloupci stavového pole tak, že sloupec po sloupci (Obrázek 4), které jsou zde chápány jako polynomy třetího řádu, násobí polynomem $a(x)$ a moduluje polynomem $x^4 + 1$.

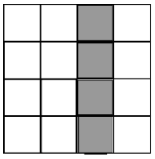
$$a(x) = \{03\}x^3 + \{01\}x^2 + \{01\}x + \{02\} \tag{2.3}$$

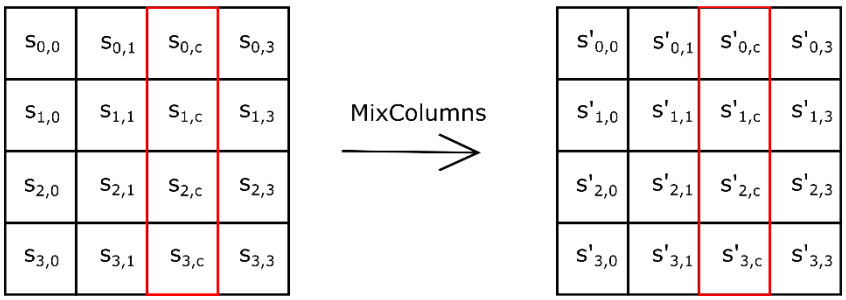
Po úpravě lze výsledek matematických kroků transformace napsat pomocí transformační matice:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{2.4}$$

Kde s je bajt stavového pole a c je index sloupce.

Jednotlivé kroky jsou detailně matematicky popsány ve [9].

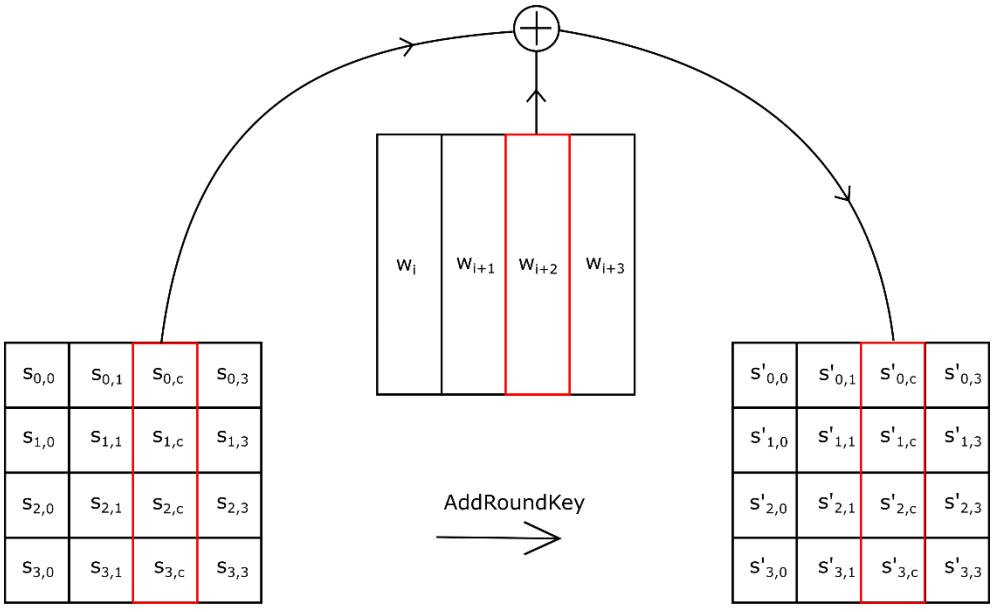




Obrázek 4: Transformace MixColumns

2.3.4 AddRoundKey()

Tato transformace (obrázek č. 5) přičítá ke stavovému poli iterační klíč ze seznamu klíčů, který vygenerovala funkce *KeyExpansion*. Generování vytváří slova o délce 4 bajty. Tyto slova jsou pak přičítány ke slovům (sloupcům) stavového pole.



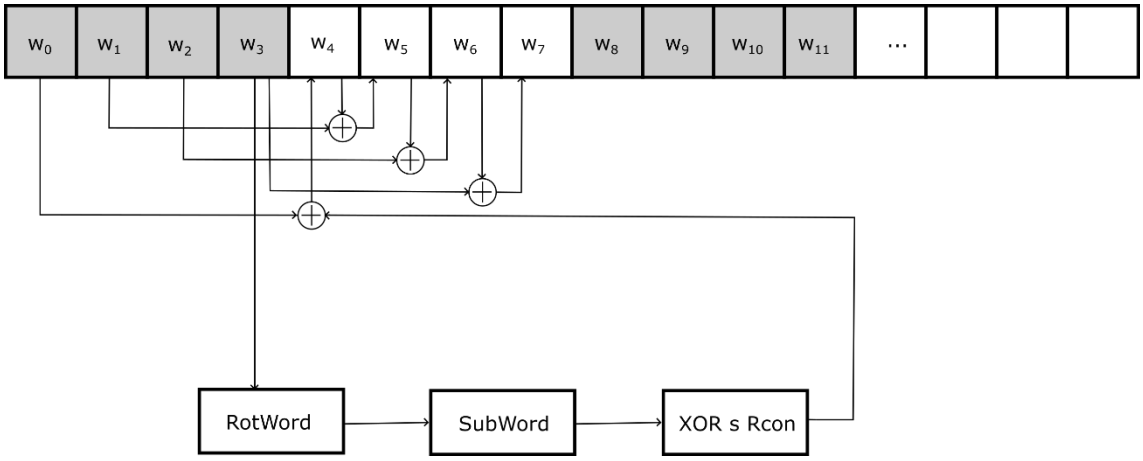
Obrázek 5: Transformace AddRoundKey

2.3.5 KeyExpansion

Vytváří z šifrovacího klíče iterační klíče, které se používají v jednotlivých iterací šifry. Generuje čtyřbajtová slova a celkovém počtu 44, 52 a 60 pro počet iterací 10, 12 a 14.

Funkce se skládá ze dvou transformací. *SubWord()* provádí substituci slova pomocí tabulky S-box. Transformace *RotWord()* provádí rotaci slova o jeden bajt vlevo. *Rcon[i]* je čtyřbajtové slovo, které obsahuje hodnoty dané výrazem $[x^{i-1},\{00\},\{00\},\{00\}]$, kde x je definováno jako {02}, takže se jedná o bitový posun vlevo. Pokud posunutím dojde k posunu na 9. bit, cele číslo se zmoduluje polynomem $m(x)$ a opět se provádí bitový posun.

Na začátku seznamu je načten šifrovací klíč rozdělený po 32 bitech na slova. Zpracování slov (Obrázek č. 6) pak probíhá po skupinách o velikosti 4, 6 a 8 (podle délky klíče). Další slova ve skupině jsou tvořena sčítáním slov z předešlé skupiny na stejné pozici. První slovo ve skupině má jinou transformaci. Nejprve prochází přes transformace *SubWord()* a *RotWord()* a nakonec je přičteno slovo *Rcon* a první slovo z předešlé skupiny.



Obrázek 6: Expanze klíče

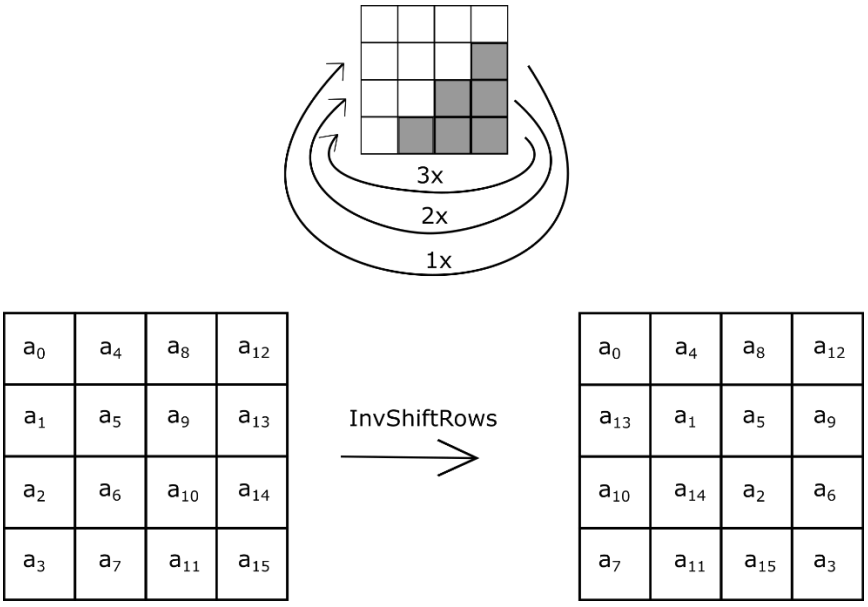
2.4 Definice algoritmu inverzní šifry

Pro dešifrování je zapotřebí aplikovat jednotlivé transformace v opačném pořadí než při šifrování. Jednotlivé transformace musí být inverzní ke svým protějškům. Ty jsou definovány jako *InvShiftRows()*, *InvSubBytes()*, *InvMicColumns()* a *AddRoundKey()*. Funkce *AddRoundKey()* nemá inverzní transformaci, protože jejím úkolem je provádět pouze operaci XOR a k této operaci je inverzní operací opět XOR.

Celý proces vypadá následovně: Na začátku dešifrování se načte první blok dat do stavového pole. Přičte se poslední iterační klíč. Dále se cyklicky provádí následující série transformací s tím, že index iterace se snižuje postupně k jedné. *InvShiftRows()*, *InvSubBytes()*, *AddRoundKey()* a *InvMixColumns()*. Po ukončení cyklu se ještě provede *InvShiftRows()*, *InvSubBytes()* a *AddRoundKey()*. Výsledkem jsou dešifrovaná původní data.

2.4.1 InvShiftRows()

Transformace je inverzní ke transformaci *ShiftRows()*. Provádí se rotace bajtů po řádcích doprava (Obrázek 7). Počet bajtů, které se rotují, je dán opět indexem řádku. Na prvním řádku není provedena žádná změna, na dalších řádcích se pak rotuje vpravo postupně o jeden, dva a tři bajty.



Obrázek 7: Inverzní transformace InvShiftRows

2.4.2 InvSubBytes()

Tato transformace je inverzní k *SubBytes()*. Provádí zpětnou inverzi podle tabulky (Tabulka 2), která je inverzní k S-boxu.

Tabulka 2: Inverzní S-box

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

2.4.3 InvMixColumns()

Transformace je inverzní k *MixColumns()*. Stejně jako u *MixColumns()* se data ve stavovém poli transformují sloupec po sloupci (Obrázek 4). S tím rozdílem, že data se násobí inverzním polynomem. Jak již bylo zmíněno, inverzní polynom k $a(x)$(2.3) je:

$$a^{-1}(x) = \{0b\}x^3 + \{0d\}x^2 + \{09\}x + \{0e\}. \tag{2.5}$$

Po různých matematických úpravách vypadá finální transformační matice následovně:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \tag{2.6}$$

3 VLASTNÍ IMPLEMENTACE KÓDU VHDL

Aplikace byla vyvíjena jako studentské řešení implementace AES ve vývojovém prostředí Vivado 2016.2 a 2016.3 od společnosti Xilinx. Syntéza a následné generování výsledného zdrojového souboru *.bit je provedeno na vývojovém serveru na ústavu telekomunikací UTKO, VUT Brno – verze 2013.4.

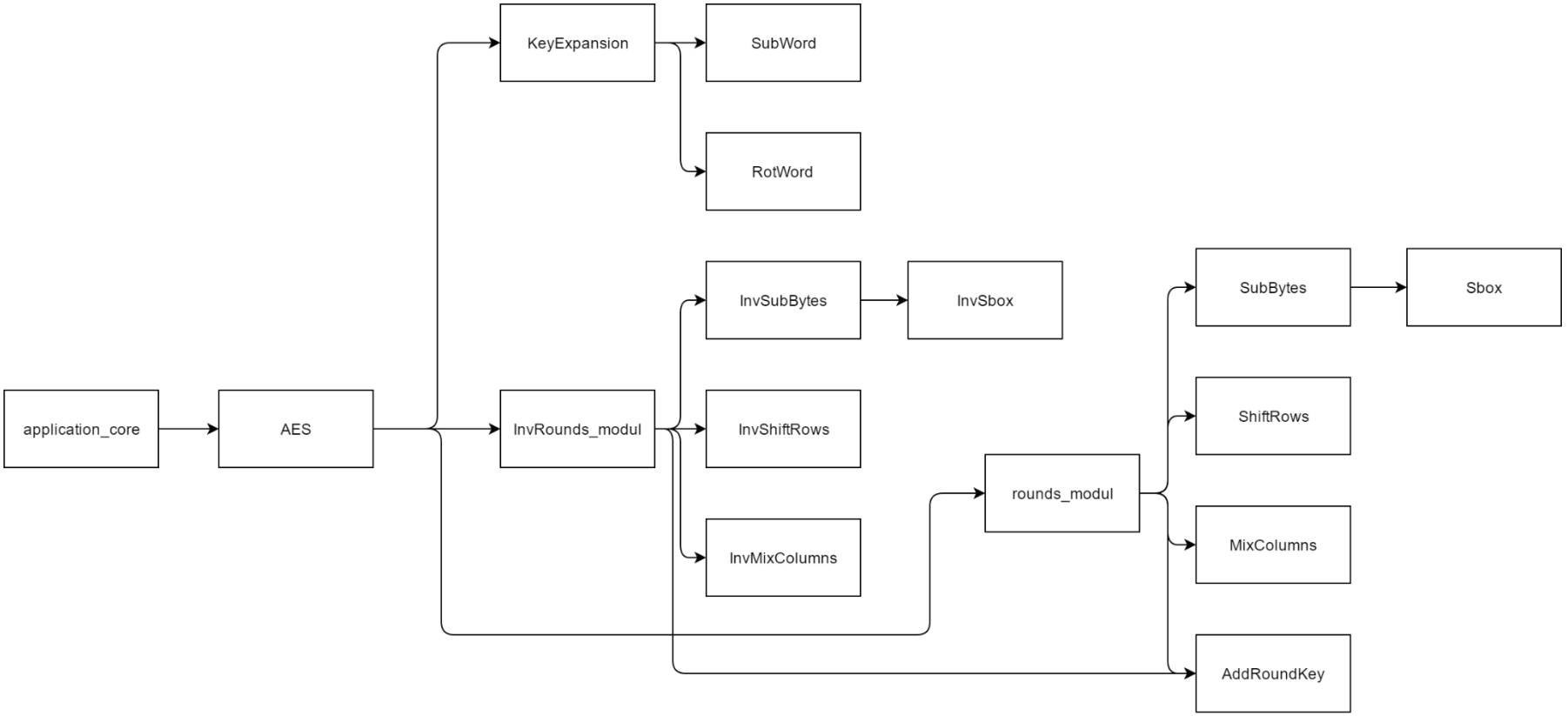
Aplikace byla vyvíjena metodou dekompozice (shora dolů). Řešení zadání (problému) bylo rozděleno na několik jednotlivých části, které implementaci řešení zadání zjednodušují. Rozdělení na jednotlivé části je popsáno v následujícím textu. Vývoj probíhal od základních transformací. Tyto transformace byly seskupovány a „obalovány" dalším kódem, a tak postupně tvořily jednotlivé iterace, potom celou část šifrování a následně funkční aplikaci, která je schopna šifrování i dešifrování. Na obrázku č. 8 je hierarchie aplikace *AES*. Každá položka ve struktuře reprezentuje zdrojový soubor a zároveň entitu (komponentu).

Samotná aplikace – entita – AES (nejvyšší úroveň) obsahuje tři komponenty: komponentu pro šifrování *rounds_modul*, pro dešifrování *invRounds_modul* a pro expanzi šifrovacího klíče *KeyExpansion*. Dále obsahuje sedm bloků, které řídí chod celé aplikace. Tyto bloky nejsou implementovány jako komponenty, ale pouze pomocí procesů, které jsou součástí jazyka VHDL. Je podporováno jen šifrování v základním režimu s délkou klíče 128 bitů. Šifra pracuje jen v základním režimu šifry a jiné režimy nejsou v této chvíli podporovány.

Entita AES komunikuje s nadřazenou úrovní (v tomto případě se bude jednat o uživatelskou aplikaci NetCOPE *application_core* – viz podkapitola 3.5.1) pomocí signálů na svém rozhraní. Seznam signálů je uveden v tabulce č. 3 níže. Řízení obsahuje kromě již zmíněných řídících bloků ještě vnitřní signály – registry. Řízení je řešeno převážně podmínkami if-else a příkazy *wait*. To vnáší do běhu aplikace zpoždění jeden takt. Entita AES je řízená synchronně vnitřními hodinami – signálem *clk*. Komponenty *rounds_modul, invRounds_modul* a *KeyExpansion* mají asynchronní časování (bez *clk*).

Řízení je vysvětleno v kapitole 3.4.

V implementaci jsou definované skupinky signálů (registrů), které popisují podobné vlastnosti nebo například části vstupních dat. Pro zjednodušení jsou takovéto signály popsány společným názvem a hvězdičkou, která říká, že se od toho místa názvy signálu liší (např. číselný popis). Pro ukázku je vybrána skupina signálů sloužící jako registry pro uložení vstupního bloku dat – *reg_wordIN0, reg_wordIN1, reg_wordIN2* a *reg_wordIN3*. Tyto 4 registry budou v textu označovány jako skupina signálů *reg_wordIN**.



Obrázek 8: Hierarchie použitých entit

Tabulka 3: Přehled signálů entity AES

AES				
Jméno	Směr	Délka	Jméno	Délka
Rozhraní			Vnitřní signály	
dataIN	in	128	key	128
dataOUT	out	128	data_Einput	128
keyIN	in	128	data_Dinput	128
clk	in	1	data_Doutput	128
encrypt	in	1	data_Eoutput	128
dataReady	out	1	data_in	128
READY	out	1	data_out	128
			expandedKey	1408
			expansionRdy	1
			systemRdy	1
			encryptionRdy	1
			dataLoaded	1
			startSystem	1
			EoutputRdy	1
			DoutputRdy	1

3.1 Šifrování

Šifrování je prováděno pomocí entity *rounds_modul*. Rozhraní entity je popsáno v tabulce č. 4 společně s rozvržením vnitřních signálů. Entita obsahuje další čtyři komponenty, každá provádí určitou transformaci nad daty, které jsou definovány ve standardu AES a byly již popsány dříve v textu. Jedná se o komponenty *SubBytes*, *ShiftRows*, *MixColumns* a *AddRoundKey*. Obsahuje jeden proces pro řízení předávání dat na výstup. Blokové schéma je na obrázku č. 9. Na obrázku č. 10 je vývojový diagram.

Tabulka 4: Přehled signálů entity rounds_modul (šifrování)

Šifrování - rounds_modul				
Jméno	Směr	Délka	Jméno	Délka
Rozhraní			Vnitřní signály	
dataIN	in	128	TEMP	10×128
Kschedule	in	1408	tempStateOut_SubBytes	10×128
dataOUT	out	128	tempStateOut_ShiftRows	10×128
clk	in	1	tempStateOut_MixColumns	10×128
DataReady	out	1	tempStateOut_SubBytes_FR	128
			tempStateOut_ShiftRows_FR	128
			data_encrypted	128
			D_rdy	1

Zpracování probíhá v devíti iteracích pomocí příkazu *generate*. Poslední iterace není součástí bloku *generate* a následuje hned za tímto blokem, protože se liší od předchozích tím, že neprovádí transformaci MixColumns.

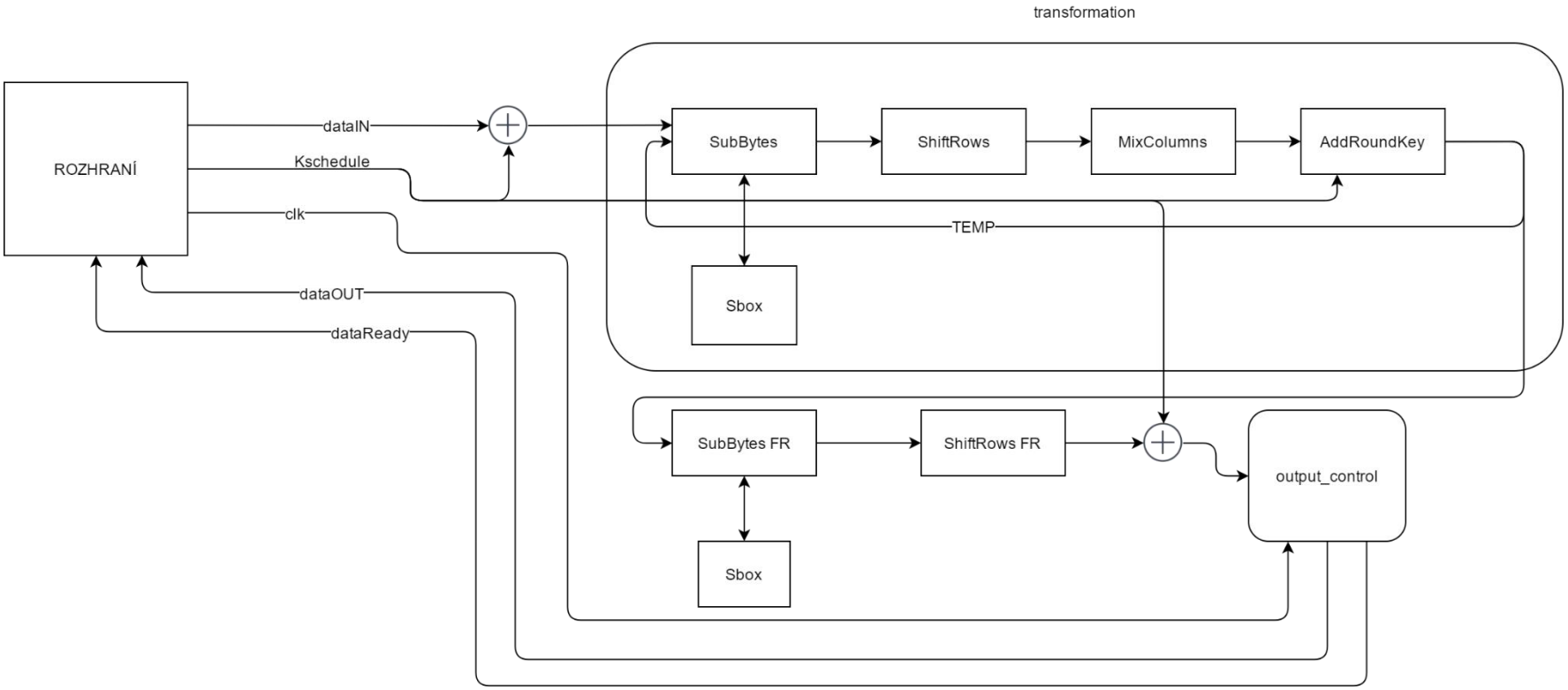
Na začátku zpracování je do pole signálů *TEMP* na pozici 0 uložen exkluzivní součet vstupních dat *dataIN* a prvních 128 bitů expandovaného klíče *Kschedule*. Následuje blok generování devíti iterací postupně přes všechny čtyři transformace. Vstupy a výstupy jednotlivých transformaci jsou provázány mezi sebou – výstup transformace je přímo napojen na vstup následující transformace. Signály *tempStateOut_** jsou pole stejné jako signál *TEMP*. Uchovávání jednotlivých výstupů z transformací a jednotlivých iterací je zbytečné a zabírá to zbytečně místo v paměti, ale je to potřeba kvůli použití příkazu *generate*, které provádí iterace paralelně – vygeneruje kód pro všechny iterace a provede je všechny najednou. Při použití jednoho 128bit signálu jako propojení mezi transformacemi má za následek, že jednotlivé instance transformací, které se vygenerují, budou přistupovat k jednomu signálu ve stejný čas a dojde ke kolizi, která se projeví červenými X v simulaci.

Komponentě *AddRoundKey* je navíc předána část expandovaného klíče *Kschedule*, která reprezentuje iterační klíč pro aktuální iteraci.

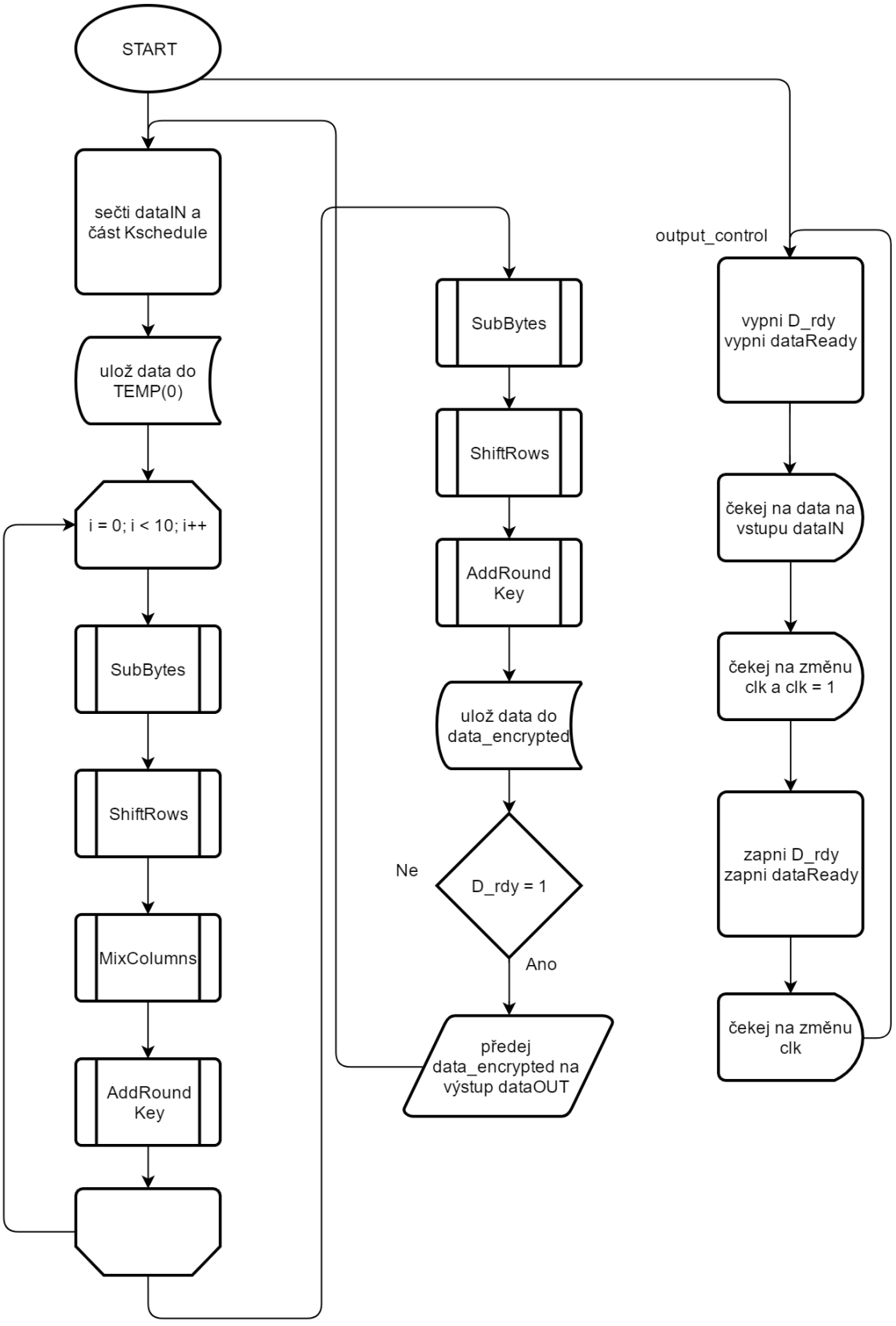
Po dokončení bloku iterací jsou provedeny tři transformace reprezentující finální iteraci a výsledek instance *AddRoundKey_cmp_FinalRound* je uložen do signálu *data_encrypted*. Odtud jsou data posílány na výstup entity, pouze když je řídící signál *D_rdy* roven 1. Obsluhu provádí proces označený jako *output_control*. Při inicializaci entity nastaví signály *D_rdy* a *DataReady* na hodnotu 0 a vyčkává na příchod dat na vstup *dataIN*. Poté vyčká na hodinový signál *clk*. Podmínka čekání *clk'event and clk = `1`* má efekt zdržení procesu jeden takt. Následně je nastavena hodnota obou signálu na 1 a opět se vyčkává na změnu *clk*. Současně se dává pokyn, aby byly zašifrované data odeslány na rozhraní. Proces se spouští znovu a oba signály vypne.

3.1.1 SubBytes

Entita obsahuje ve svém rozhraní dva 128bit signály *stateIN* a *stateOUT* reprezentující stav – vlastní data. Dále využívá další komponentu *Sbox*, která provádí vlastní transformaci bajtů. Entita *SubBytes* vykonává stejnojmennou transformaci jednoduchým příkazem *generate*, ve kterém je v 16 iteracích prováděno mapování vstupních dat na jednotlivé bajty, následně je provedena transformace pomocí komponenty *Sbox* a konečné ukládání bajtů zpět do výstupu entity.



Obrázek 9: Blokové schéma enitiy rounds_modul (šifrování)



Obrázek 10: Vývojový diagram rounds_modul (šifrování)

3.1.2 Sbox

Entita zpracovává data po bajtech – na rozhraní jsou dva signály *byteIN* a *byteOUT* o délce 8 bitů. Obsahuje pole 256 vnitřních signálů o délce jednoho bajtu (8 bitů) reprezentující S-Box.

S-Box je navržen tak, že první hexadecimální číslice bajtu reprezentuje pořadí řádku S-Boxu a druhá číslice pozici sloupce (16×16). Zadání hodnot tabulky do pole po řádcích umožňuje provádět substituci tak, že substituční hodnota, která nahradí původní hodnotu, se nachází na pozici, která odpovídá hodnotě původního bajtu.

Implementaci lze udělat minimálně 2 způsoby:

1) Implementace pomocí pole, které představuje S-Box (je použita tato implementace). Je použit signál *index* typu *integer*, do které je pomocí funkce *to_integer()* z balíčku *IEEE.NUMERIC_STD.ALL* převedena hodnota vstupního bajtu jako *unsigned integer*. Je potřeba použít *unsigned*, protože jinak se načítají do proměnné záporné hodnoty kvůli způsobu vyjádření záporného čísla pomocí dvojkového doplňku. Entita vrací signál uložený v poli na pozici *index*.
2) Implementace pomocí multiplexeru. Využívá se podmíněného přiřazení do signálu.
3) Implementace pomocí matematického vyjádření. Využívá se matematický výpočet (je definován v [9]).

První dvě jsou jednoduše implementovatelné. Zabírají sice místo v paměti (256 signálů), ale záleží o kolik je toto řešení rychlejší než matematický výpočet. Tato možnost nebyla ověřena.

3.1.3 ShiftRows

Na rozhraní entity jsou opět dva signály *stateIN* a *stateOUT* o délce 128 bitů, které reprezentují stav šifry.

První verze implementace obsahovala 16 aliasů, které dělily vstupní signál na jednotlivé bajty dle obrázku č. 1. Bajty jsou číslovány vzestupně od levého horního rohu směrem dolů a po sloupcích zleva doprava. Aliasy byly následně přeskupeny tak, aby odpovídaly transformaci ShiftRows. A opět stejným způsobem seřazeny za sebe, aby vytvořily výstupní signál.

Aliasy jsou tvořeny podle obecného zápisu:

```
alias a15-i : std_logic_vector (7 downto 0) is stateIN(i*8+7 downto i*8),
```

kde *i* nabývá hodnot 0 až 15.

Druhá verze implementace je po optimalizaci. Aliasy v paměti zabírají místo, a proto tento návrh od nich opouští. Využívá té možnosti VHDL, že lze použít část signálu bez použití aliasu, který tuto část signálu definuje. Implementace namísto aliasů vkládá do výstupního signálu *stateOUT* části vstupního signálu *stateIN*, které reprezentovaly. Kód to trochu znepřehlední, ale už nejsou vytvářeny konstrukce zabírající paměť.

Příklad části kódu:

Původní kód s použitím aliasů:

```
alias a0 : std_logic_vector (7 downto 0) is stateIN(127 downto 120);
alias a5 : std_logic_vector (7 downto 0) is stateIN(87 downto 80);
alias a10 : std_logic_vector (7 downto 0) is stateIN(47 downto 40);
alias a15 : std_logic_vector (7 downto 0) is stateIN(7 downto 0);
…
stateOUT <= a0 & a5 &  a10 & a15
```

Nový kód:

```
stateOUT <= stateIN(127 downto 120) & stateIN(87 downto 80) &
stateIN(47 downto 40) & stateIN(7 downto 0)
```

3.1.4 MixColumns

Entita má na rozhraní 2 128bit signály – *stateIN* a *stateOUT*. Obsahuje dvě vnitřní pole *state* a *state2* o velikosti 16×8 bitů, do kterých jsou ukládány mezivýpočty. Tranformace manipuluje se stavem vždy po jeho jednotlivých sloupcích a s každým bajtem jinak.

Násobení dvou polynomů reprezentující dva sloupce stavu bylo implementováno pomocí funkce *xtime* (kapitola 2.2.1), která je definovaná ve standardu AES. Využívá se vlastnosti, kdy opakovaným použití funkce lze dosáhnout výsledku násobení hodnot dvou bajtů reprezentující koeficienty polynomů. Funkce *xterm* vrací stejný výsledek jako násobení hodnotou {02}. V entitě *MixColumns* byla implementována jako funkce *xtimeFUN*, aby vracela hodnotu přímo do výpočtu.

xtimeFUN funkce má na vstupu bajt (koeficient polynomu) a vrací opět bajt. Jsou definovány proměnné *mx*, *result* a alias *lastBit*. *Mx* reprezentuje polynom $m(x)$ vyjádřený binárním číslem. V proměnné *result* jsou uloženy kroky výpočtu. Všechny proměnné jsou typu *bit_vector* o délce 9 bitů. Vstupní bajt (8 bitů) musí být taky typu *bit_vector*, protože operace *sll* (bitový posun vlevo) nepodporuje typ *std_logic_vector*. Do proměnné *result* se uloží nejprve „0“ a hodnota na vstupu. Dále se provede posun o jeden bit vlevo. Následně se testuje, jestli je 9. bit obsazený, má hodnotu 1 (v kódu bit na pozici 8 – používá se číslování od nuly). Pokud ano, provede by se operace modulo polynomem $m(x)$. Protože polynomy čísel *result* a $m(x)$ jsou v tomto případě stejného řádu, není potřeba polynomy složitě dělit a počítat zbytek. Postačí pouze exkluzivní součet a výsledek je zredukován. Z důvodu použití devítibitových proměnných, výsledek je třeba vrátit jako 8bit hodnotu typu *std_logic_vector*. Použije se funkce *to_stdlogicvector* a předá se jí 8 posledních bitů proměnné *result*.

Funkce *xtimeFUN*:

```
function xtimeFUN (byteIN : std_logic_vector (7 downto 0)) return
std_logic_vector is

    variable mx : bit_vector (8 downto 0) := "100011011";
    variable result : bit_vector (8 downto 0) ;
    alias lastBit : bit is result(8);
    begin
        result :=  "0" & to_bitvector(byteIN) sll 1;

        if lastBit = '1' then
            result := result xor mx;
      end if;

     return to_stdlogicvector(result(7 downto 0));
end xtimeFUN;
```

V bloku *nacti_state* se pomocí příkazu *generate* dělí vstupní signál *stateIN* na jednotlivé bajty a ukládá do pole *state*. V dalším bloku *matice* za použití opět cyklu *generate* je prováděna transformace bajtů. Každá iterace – celkem 4 – představuje transformaci jednoho sloupce stavu. Obsahuje 4 příkazy

```
state2(i*4+3) <= (xtermFUN(state(i*4+3))) xor (xtermFUN(state(i*4+2))
xor state(i*4+2)) xor state(i*4+1) xor state(i*4);
```

odpovídající rovnicím z kapitoly 2.3.3 pro transformaci MixColumns.

$$s'_{0,3} = (\{02\} \cdot s_{0,3}) \oplus (\{03\} \cdot s_{1,3}) \oplus s_{2,3} \oplus s_{3,3} \tag{3.1}$$

Výsledek transformace prováděný po bajtech je ukládán do druhého identického pole *state2*, a nakonec je výsledný signál *stateOUT* vytvořen poskládáním jednotlivých bajtů za sebe, stejným způsobem jako v entitě *ShiftRows*. Sestupně od levého horního rohu matice stavu dolů a po sloupcích zleva doprava.

3.1.5 AddRoundKey

Poslední entita použitá při šifrování taktéž pracuje se stavem – rozhraní obsahuje 128bit signály *stateIN* a *stateOUT*. Navíc obsahuje 128bit signál *roundKey*, který obsahuje iterační klíč příslušné iterace.

Dle specifikace jsou iterační klíče generovány pomocí expanze klíče, a to po jednotlivých 32bit slovech. Iterační klíč sestává ze skupiny 4 slov, které jsou předány transformaci AddRoundKey. V této transformaci jsou jednotlivé slova exkluzivně sčítány s jednotlivými sloupci stavu. Díky jednotnému zacházení s blokem dat v celé šifře lze zacházet s iteračními klíči jako klíči o délce 128 bitů a přímo je sčítat se stavem. Pro zjednodušení je tato entita implementována tímto způsobem. Na výstup je posílán exkluzivní součet signálu *stateIN* s *roundKey*. Tuto entitu lze vypustit a místo ní použít pouze operátor *xor*, ale pak by struktura neodpovídala specifikaci.

3.2 Dešifrování

Dešifrování dat je prováděno entitou *InvRounds_modul*, bratříček šifrovací entity. Uspořádání signálů na rozhraní a vnitřních signálů je v tabulce č. 5. Dešifrování probíhá v 9 iteracích za pomoci 4 komponent (*InvSubBytes*, *InvShiftRows*, *InvMixColumns* a *AddRoundKey)*, které provádí inverzní transformace. Inverzní operace k operaci *xor* je opět *xor*, proto není zapotřebí definovat inverzní transformaci k transformaci *AddRoundKey*.

Dešifrovací proces je implementován téměř identicky jako šifrování. Signály *TEMP* a *temp_StateOut_** mají stejný význam i použití. Obsahuje blok *transformation* s cyklem *generate*, který má na rozdíl od šifrování jiné pořadí jednotlivých transformací. Řídící proces *output_control* pracuje stejně jako ten v šifrovací entitě. Blokové schéma je na obrázku č. 11. A vývojový diagram je na obrázku č. 12.

Tabulka 5: Přehled signálů invRounds_modul (dešifrování)

Dešifrování - InvRounds_modul				
Jméno	Směr	Délka	Jméno	Délka
Rozhraní			Vnitřní signály	
dataIN	in	128	TEMP	10×128
Kschedule	in	1408	tempStateOut_InvSubBytes	10×128
dataOUT	out	128	tempStateOut_InvShiftRows	10×128
clk	in	1	tempStateOut_AddRoundKey	10×128
DataReady	out	1	tempStateOut_InvSubBytes_FR	128
			tempStateOut_InvShiftRows_FR	128
			data_decrypted	128
			D_rdy	1

Na začátku je do pole *TEMP* uložen exkluzivní součet vstupního signálu s posledním 11. iteračním klíčem (pozice *Kschedule(127 downto 0))*. Dále přichází na řadu cyklus s následujícím pořadím komponent: *InvShiftRows*, *InvSubBytes*, *AddRoundKey* a *InvMixColumns*. Protože je pořadí komponent posunuté a neodpovídá pořadí při šifrování, skončí vykonávání jako kdyby uprostřed 1. iterace šifrování (dešifrování začíná od konce). K dokončení dešifrování je potřeba ještě 3 kroků. Komponenty *InvShiftRows* a *InvSubBytes*. Poslední krok je transformace AddRoundKey, která je v tomto místě implementovaná pouze jako exkluzivní součet výstupu z předchozí komponenty *InvSubBytes* s původním klíčem (pozice *Kschedule(1407 downto 1280))*.

Následující entity *InvSubBytes*, *InvShiftRows* a *InvMixColumns* zpracovávají stavové pole a rozhraní obsahuje vždy dva 128bit signály reprezentující toto pole.

3.2.1 InvSubBytes

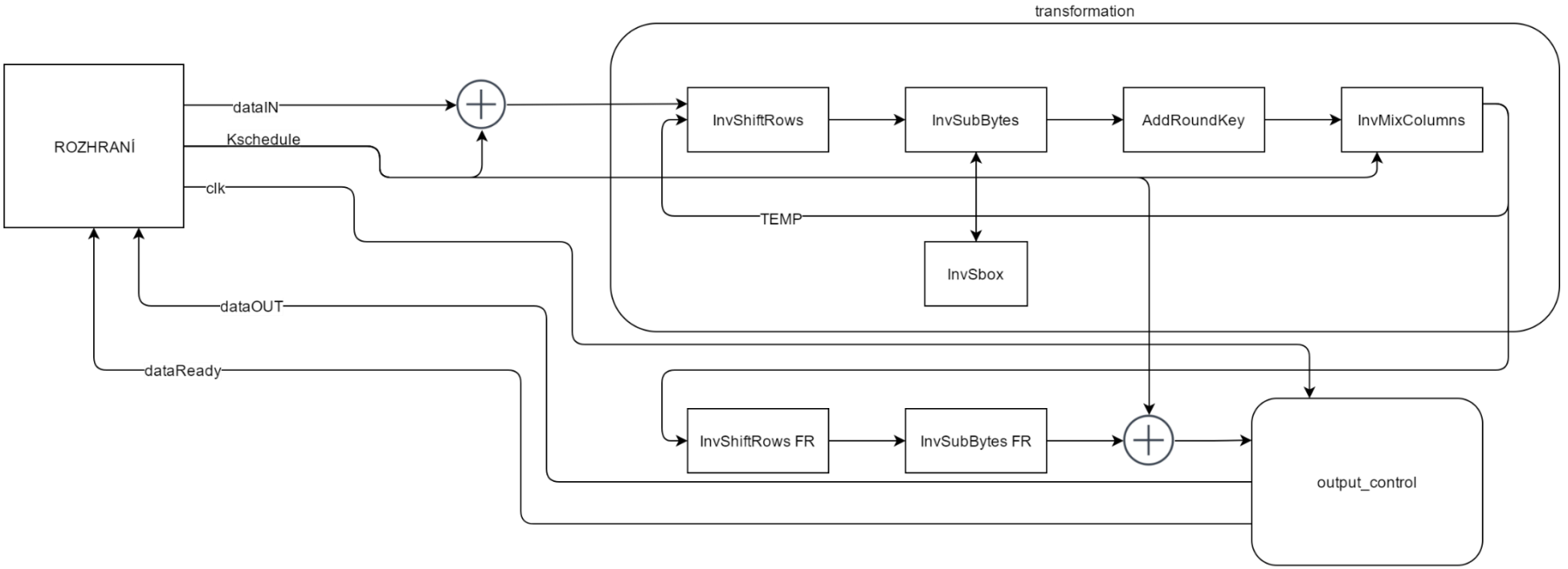
Tato entita provádí úplně stejné mapování vstupních dat na jednotlivé bajty jako její šifrovací protějšek. S rozdílem, že bajty jsou předávány komponentě *InvSbox*. Cyklem *generate* je opět vytvořeno 16 instancí, ve kterých se předávají jednotlivé bajty a vzápětí jsou transformované bajty ukládány na své místo ve výstupním signálu.

3.2.2 InvSbox

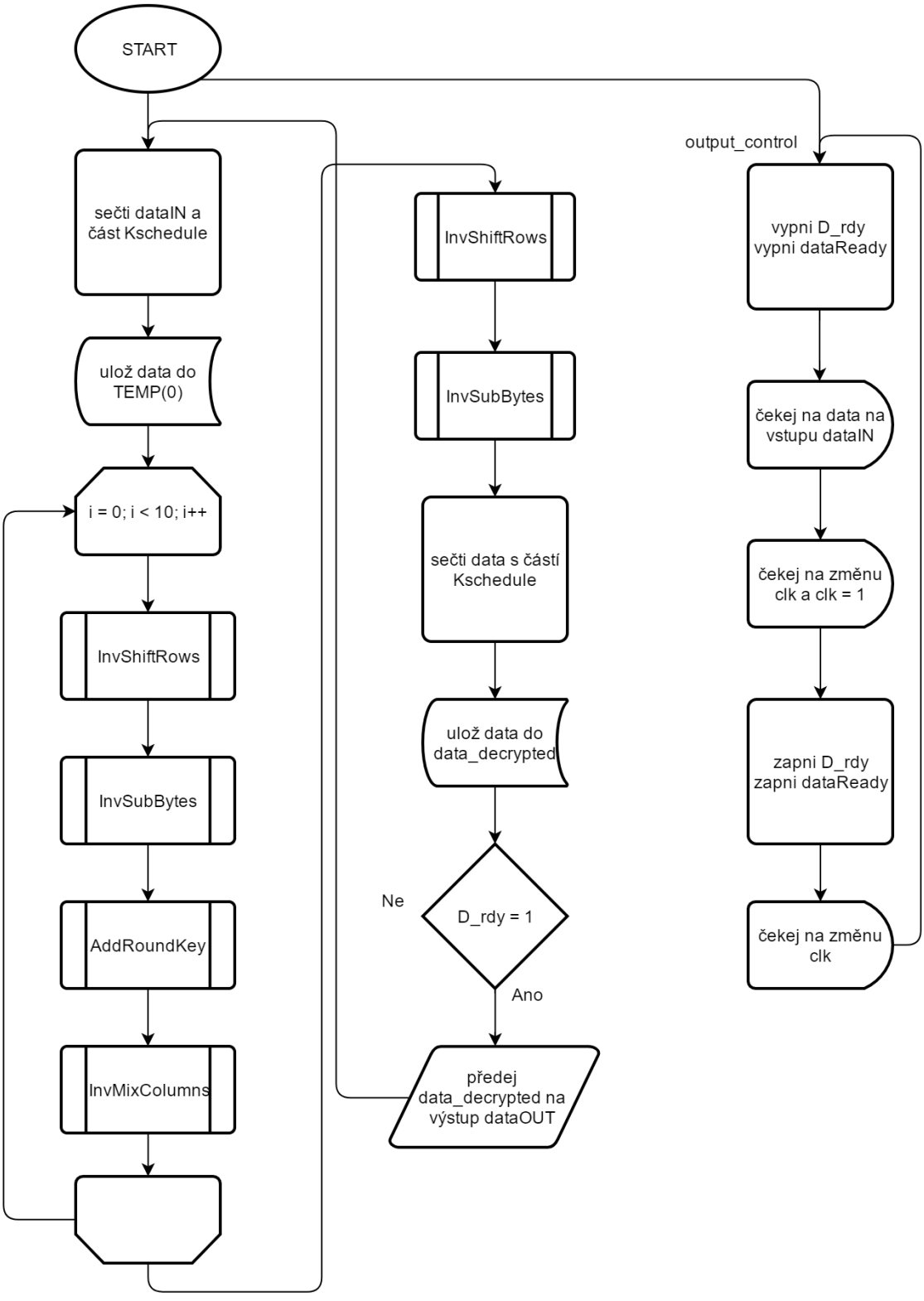
Entita je taktéž implementována stejně jako *Sbox*, jen s tím rozdílem, že obsahuje pole *Inv_S_box*, které má stejnou funkci jako pole *S_box* – inverzní transformace bajtů.

3.2.3 InvShiftRows

Entita zpracovává data stejným způsobem jako její verze v šifrování. Jednotlivé bajty ve vstupním signálu mají změněnou pozici ve výstupním signálu tak, aby výsledek odpovídal rotaci bajtů po řádcích a vytvořil tak inverzní transformaci.



Obrázek 11: Blokové schéma invRounds_modul (dešifrování)



Obrázek 12: Vývojový diagram invRounds_modul (dešifrování)

3.2.4 InvMixColumns

Implementace mění pouze blok *transformation* s cyklem. Čtyři rovnice zůstávají a jen se změnil počet opakování a kombinace funkce *xtermFUN* tak, aby se docílilo násobení požadovanou hodnotou a tím i inverzi.

V této entitě je navíc ještě pokus o optimalizaci kódu – je zde použit jen jeden vnitřní signál *state* o délce 128 bitů. Jednotlivé bajty jsou načítány do rovnic jako části vstupního signálu a následně ukládány na specifické pozice vnitřního signálu. Pro další šetření místa na FPGA čipu je možné vnitřní signál úplně vypustit a výsledky ukládat přímo do výstupu. Kód se tak ještě díky řetězení a opakování funkce *xtermFUN* stává méně přehledný. Vnitřní signál zůstal implementován z předchozích verzí, kdy byly všechny komponenty a části šifrování a dešifrování řízený synchronně pomocí vnitřních hodin *clk*.

3.3 KeyExpansion

Entita generuje dle specifikace 44 slov – 32 bitů dlouhé signály, které tvoří dohromady 11 iteračních klíčů. Tento počet vygenerovaných klíčů je pouze pro klíč šifry o délce 128 bitů. Přehled rozhraní a vnitřních signálů je v tabulce č. 6. Entita dále sestává ze dvou komponent *SubWord* a *RotWord*. Vývojový diagram expanze je na obrázku č. 13.

Tabulka 6: Přehled signálů expanze KeyExpansion

Expanze klíče - KeyExpansion				
Jméno	Směr	Délka	Jméno	Délka
Rozhraní			Vnitřní signály	
keyIN	in	128	Kschedule	44×32
scheduleKeyOut	out	1408	rcona	10×32
clk	in	1	keyFromRotWord	10×32
dataReady	out	1	keyFromSubWord	10×32
			D_rdy	1

Na začátku se načte hlavní klíč do pole *Kschedule*. 0. pozice obsahuje prvních 32 MSB bitů klíče. Generování probíhá v bloku *iterationOfGener*, který obsahuje dva cykly *generate*. Vnější cyklus má 10 iterací a zpracovává klíče po skupině o 4 slovech. Obsahuje inicializaci obou komponent, kterým je předáno první slovo skupiny. Toto první slovo prochází specifickou transformací, která sestává z průchodu oběma komponentami a sčítání se signálem *rcona* (ve specifikaci označeno *Rcon*). Stejně jako u šifrování a dešifrovaní musí být k propojení komponent použito pole signálů namísto jednoho (cyklus *generate* spouští všechny iterace v jeden okamžik, takže by došlo k přístupu ze všech iterací na jeden signál v jeden moment, a tudíž ke kolizi). Po provedení všech kroků transformace tohoto klíče se přistoupí k druhému cyklu *restKeyGener*, kde jsou vytvořeny zbylé 3 klíče skupiny operací *xor*.

Po vygenerování klíčů je potřeba je předat na výstup. To se děje v bloku *output* za pomocí dalšího cyklu *generate*. Ukládání do signálu je prováděno od konce. Slovo v poli na pozici 43 je slovem s nejnižší vahou (LSB), tudíž je ve výstupní signálu úplně na konci (vpravo) a načítání probíhá v poli sestupně a ve výstupu zprava doleva. Vše doplňuje podmínka s vnitřním signálem *D_rdy*, který je řízen z bloku *output_control* a při zapnutí odesílá na výstup data. Jinak jsou posílány samé nuly.

Řídící proces *output_control* nastavuje signály *D_rdy* a *dataReady*. Pracuje synchronně s vnitřními hodinami *clk*. Oba signály jsou v původním stavu vypnuté. Proces čeká na příjem dat na vstupu. Následně provede opět vypnutí signálů pro případ, kdyby se generování nespouštělo poprvé. Pak čeká jeden takt a zapíná oba signály. Ty jsou aktivní, dokud nedojdou nová data na vstup.

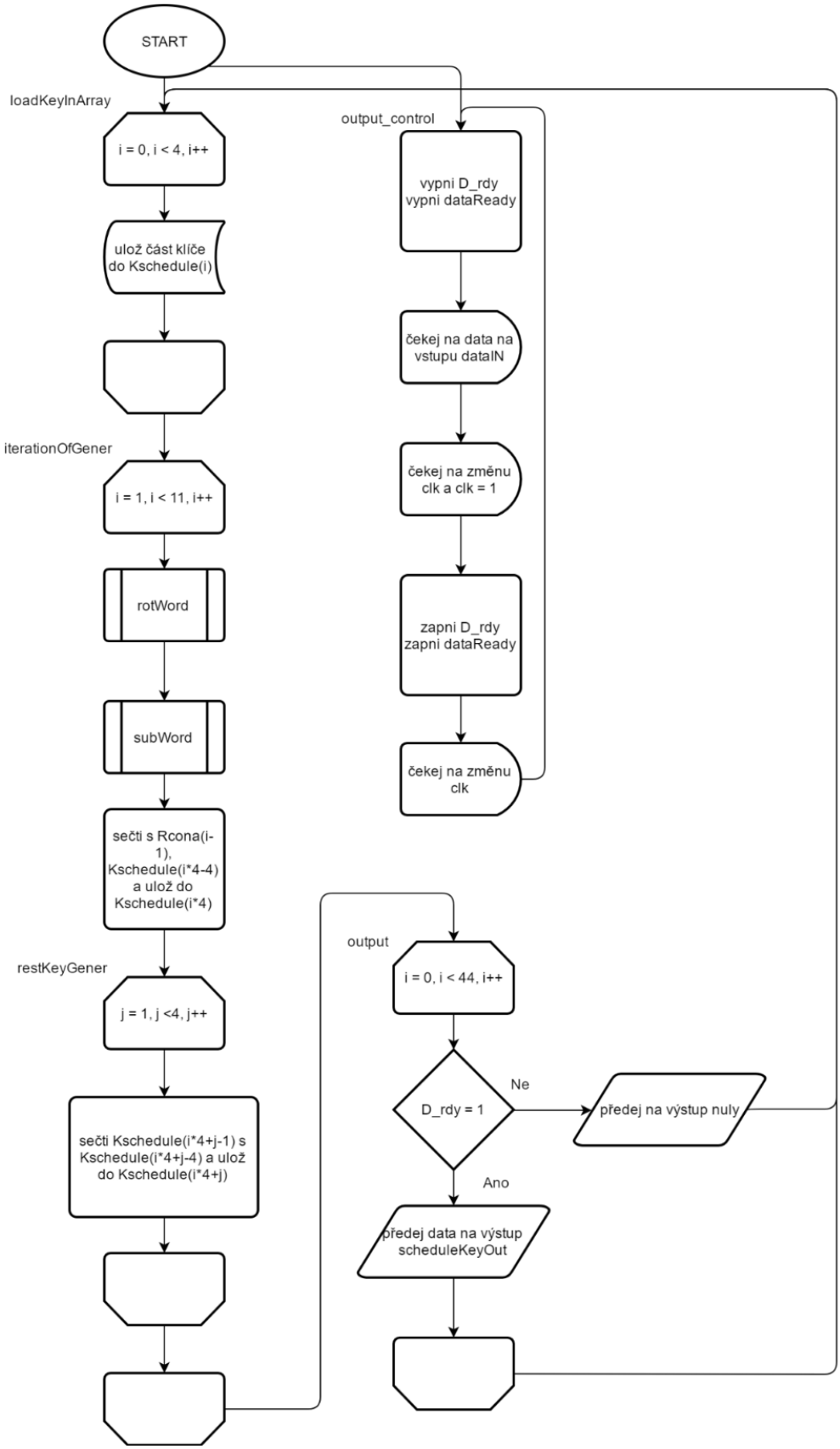
3.3.1 SubWord

Implementace je provedena se stejnou myšlenkou jako předchozí entity *SubBytes* a *InvSubBytes*. Entita využívá stejnou komponentu *Sbox* jako entita *SubBytes*. Transformace se liší v tom, že entita *SubWord* zpracovává slova – 32bit signály. Tudíž obsahuje cyklus jen se 4 iteracemi.

3.3.2 RotWord

Entita provádí ve slově (32 bitů) rotaci 1. bajtu o jednu doleva. Implementace je provedena pouhou záměnou částí vstupního signálu, které odpovídají jednotlivým bajtům. Posledních 24 bitů (3 bajty) jsou na výstup vloženy jako první a následně zbylých 8 bitů, které byly na vstupu na začátku.

```
wordOUT <= wordIN(23 downto 0) & wordIN(31 downto 24);
```



Obrázek 13: Vývojový diagram expanze KeyExpansion

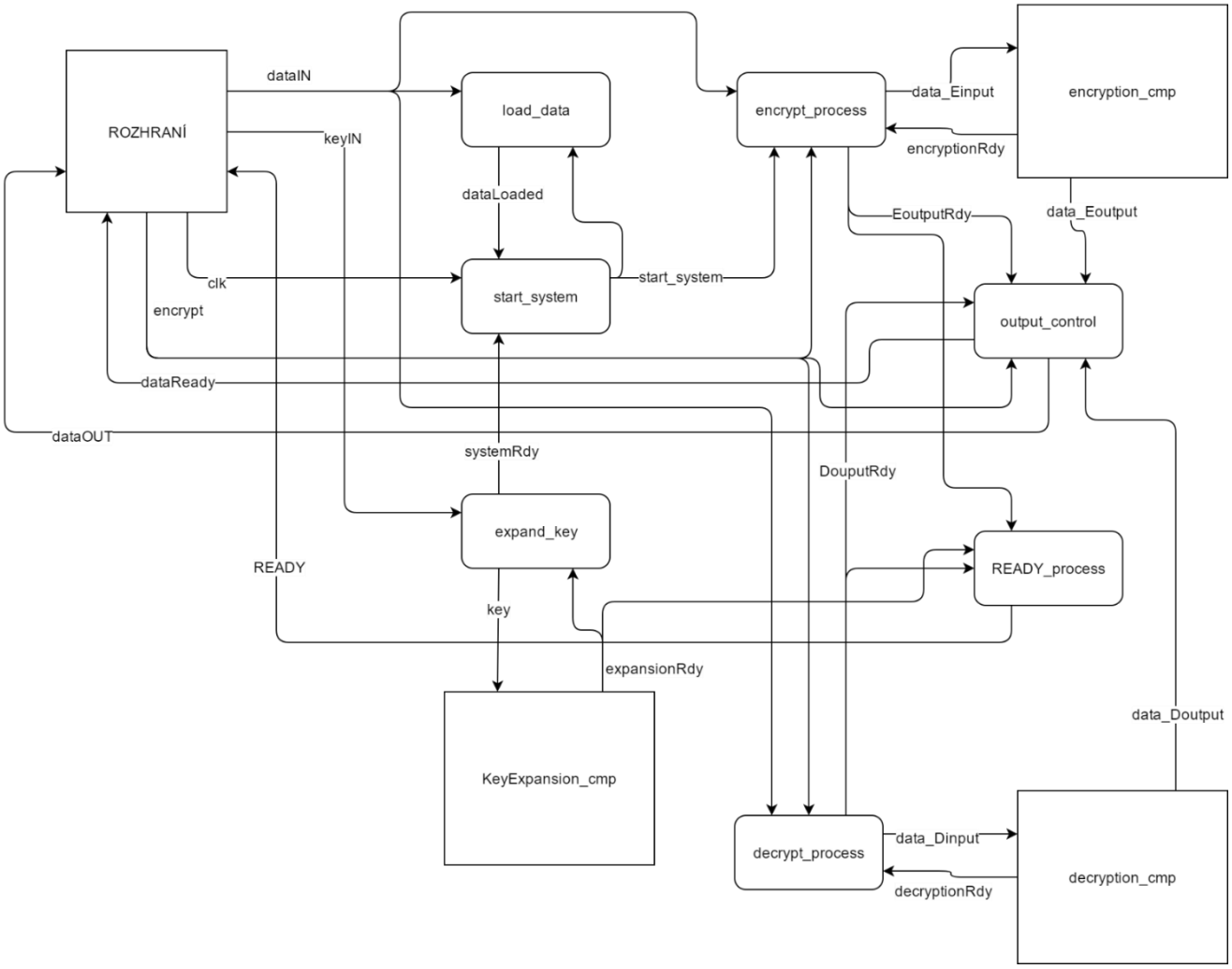
3.4 AES – řídící část

Řízení obsahuje několik 1bit řídících signálů, které slouží ke komunikaci mezi jednotlivými procesy řídící chování entity. Blokové schéma a vývojový diagram řídící části jsou na obrázcích č. 14, č. 15 a č. 16. Vývojový diagram byl kvůli své velikosti (čitelnosti) rozdělen na dvě části. Ačkoliv se zdá, že jde o úplně jiné části programu, ve VHDL běží procesy souběžně vedle sebe. Společný pro obě části diagramu je *START*, který spojuje všechny části do jednoho celku.

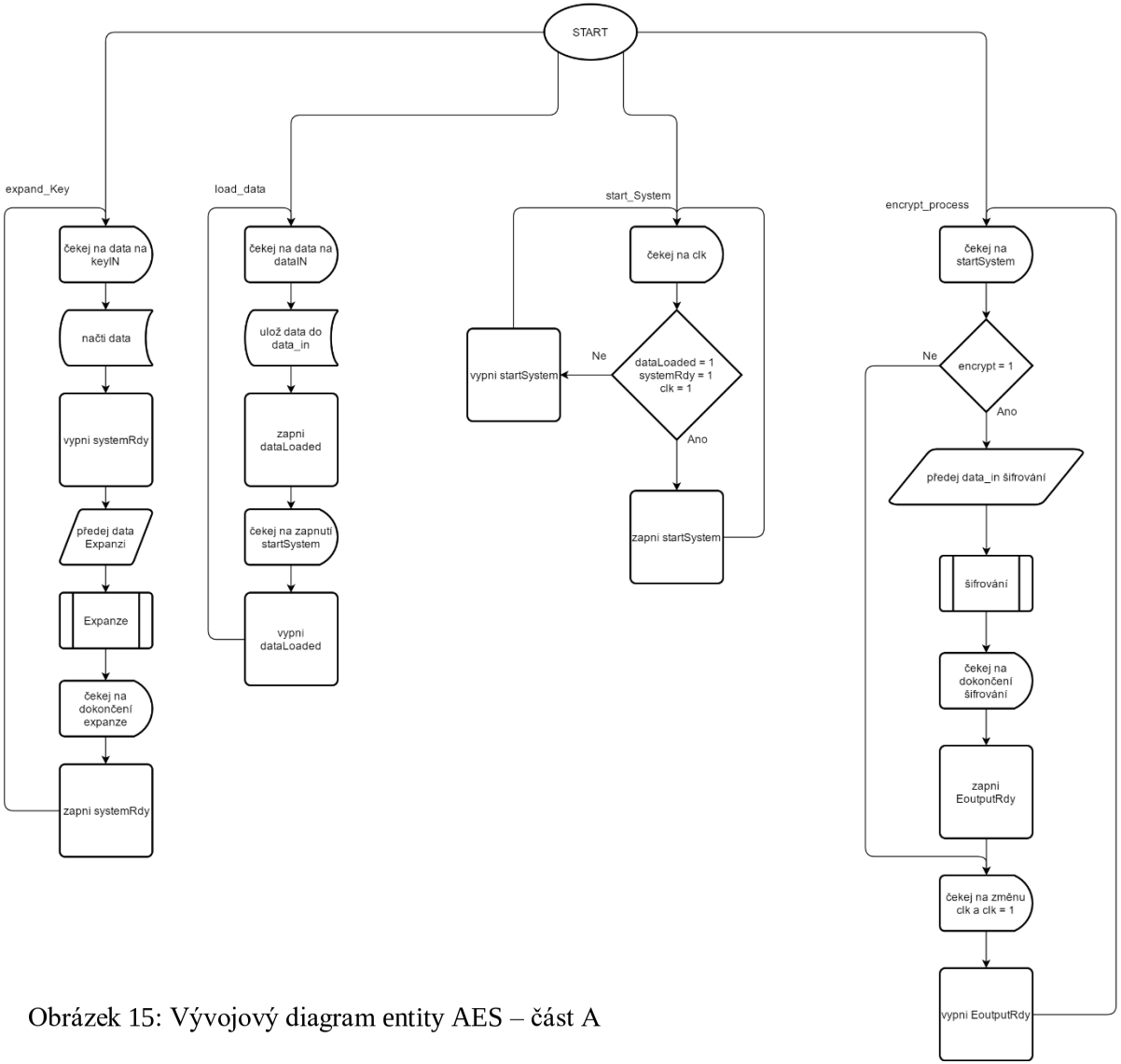
Proces *expand_key* má na starosti expanzi klíče a pozastavení systému v případě generace. Proces čeká na příchozí data v signálu *keyIN*. Po načtení dat posílá klíč na vstup komponenty *KeyExpansion* a zároveň vypíná signál *systemRdy*. Dále čeká na signál *expansionRdy* od komponenty *KeyExpansion* signalizující dokončenou expanzi. Zapne se signál *systemRdy* a proces se spouští znovu, kde čeká opět na nový šifrovací klíč.

Proces *load_data* sleduje vstupní data určené k šifrování/dešifrování. Při příchodu dat na vstup je načte do pomocného signálu *data_in* a zapne signál *dataLoaded*, který udává, že data byly načteny do paměti. Proces pak čeká na aktivaci signálu *startSystem* a následně vypíná *dataLoaded*. Proces se restartuje a čeká na nová data.

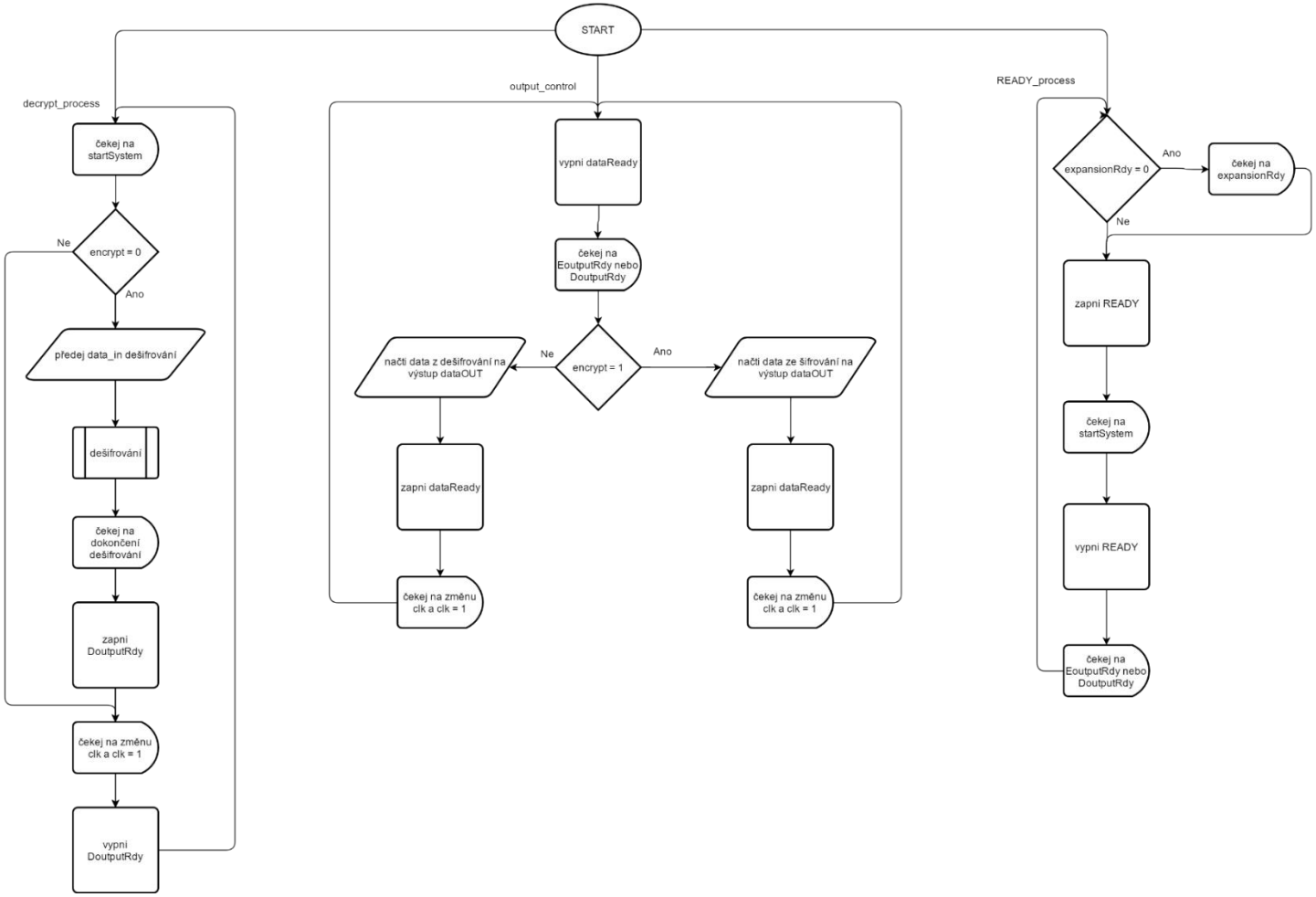
Proces *start_system* spouští šifrování nebo dešifrování. Je řízen vnitřními hodinami *clk* a testuje, jestli jsou vstupní data načteny a expanze klíče dokončena, aby nedošlo ke spuštění bez dat nebo bez iteračních klíčů. Při splnění podmínky v okamžiku, kdy je signál *clk* roven 1, zapne signál *startSystem*.



Obrázek 14: Blokové schéma entity AES



Obrázek 15: Vývojový diagram entity AES – část A



Obrázek 16: Vývojový diagram entity AES – část B

Proces *encrypt_process* řídí předávání dat mezi touto řídící entitou a šifrovací komponentou. Na začátku čeká na aktivaci *startSystem*. Následují testovací podmínka pro signál *encrypt*. Pokud je rovna 1, pokračuje se předáváním dat na komponentu – signál *data_Einput*. Dále čeká na signál *encryptionRdy,* který je napojený na signál *dataReady* komponenty. Následně je aktivován signál *EoutputRdy*. Poté se proces na takt zdrží a *EoutputRdy* se vypne. Kopií toho procesu je i *decrypt_process* s tím rozdílem, že signály obsahující v názvu „encrypt“ nebo jen „E“ jsou přejmenované. Funkci však plní totožnou.

Proces *output_control* řídí výstup entity. Proces čeká na změnu jednoho ze signálů *EoutputRdy* nebo *DoutputRdy*, potom testem stavu signálu *encrypt* rozhoduje, jestli se jednalo a šifrování nebo dešifrování. Podle toho pošle na výstup entity data z výstupu šifrovací komponenty (*data_Eoutput*) nebo z dešifrovací komponenty (*data_Doutput*). Pak nastaví signál *dataReady* na hodnotu 1 a čeká jeden takt. Tímto signalizuje nadřazené úrovni, že transformovaná data jsou validní a připravena na výstupu *dataOUT*. Po jednom taktu se proces restartuje a signál *dataReady* se vypíná.

Poslední proces *READY_process*, jak už z názvu vyplývá, obsluhuje výstupní signál *READY*, který dává nadřazené úrovni najevo, že entita je připravena zpracovávat další blok dat. Signál *READY* se na začátku aktivuje a čeká, než se spustí šifrování/dešifrování dat, to signalizuje *startSystem*. Vzápětí se *READY* vypne a setrvává, dokud proces nezaznamená aktivitu na signálech *EouputRdy* nebo *DoutputRdy*. Pak se proces restartuje a signál *READY* je opět aktivní. Na začátku je vřazena podmínka pro případ, že probíhá expanze klíče. Pokud není signál *expansionRdy* roven 1, je proces pozdržen, dokud se tak nestane. Tato podmínka ošetřuje problém pouze při inicializaci aplikace, kdy není expanze dokončena. Při budoucí změně klíče, je při rychlosti zpracování dat, velmi pravděpodobné, že v momentě, kdy bude probíhat nová expanze klíče, bude proces na pozici, kdy aktivoval signál *READY* a čeká na *startSystem*. Aplikace sice nespustí transformaci dat, ale ani nedojde k vypnutí signálu *READY*.

Implementace byla roztažena tímto způsobem z důvodu, že při pokusu řídit několik signálů najednou z různých procesů, docházelo často ke kolizi signálů. Na příklad při pokusu o zapsání výsledku na výstup entity z přímo z procesů *encrypt_process* a *decrypt_process*.

Vzorový popis činnosti aplikace: Při inicializaci jsou řídící signály vypnuty. Signál *READY* je taktéž vypnutý, a tudíž aplikace nepřijímá žádná data ke zpracování. V této fázi se čeká na přijetí šifrovacího klíče – *keyIN*. Při načtení klíče se v procesu *expand_Key* vypne signál *systemRdy* a předá se klíč expanzi. Následně se čeká na dokončení expanze (signál *expansionRdy*) a *systemRdy* se zapne. Zároveň se v procesu *READY_process* při nedokončené expanzi proces zastaví a neobnoví se, než je expanze dokončena. Potom se aktivuje *READY*. V tomto okamžiku lze předat data entitě. V procesu *load_data* se při příchozích datech aktivuje signál *dataLoaded* a ten zůstane aktivní, dokud se nespustí transformace dat. V dalším procesu *start_system* se přijde v platnost kontrolní podmínka a aktivuje se signál *startSystem*. Při aktivaci tohoto signálu se spustí procesy *encrypt_process* a *decrypt_process*. Zároveň se vypne *READY*. V tomto příkladu se bude šifrovat, takže dešifrovací proces projde naprázdno a šifrovacím procesu se předají data na šifrovací komponentu a čeká se na její dokončení. Po dokončení aktivuje signál *EoutputRdy* a po jednom taktu se opět vypne. Současně při aktivaci se spustí proces *output_control*, který podle signálu *encrypt* vybere, ze které komponenty se pošlou data na výstup. Současně se aktivuje *dataReady* na jeden takt signalizující připravené data na

výstupu. Současně se aktivuje *READY*. Entita dokončila transformaci a signalizuje, že je připravena a očekává další data. Nový proces opět pokračuje v procesu *load_data*.

V kapitole 3.6.1 je na obrázku č. 19 vyobrazen průběh řídících signálů.

3.5 NetCOPE

Tato kapitola popisuje propojení entity s uživatelskou aplikací firmwaru NetCOPE běžící přímo na kartě. Hierarchie celého NetCOPE je rozdělena do několika složek. Uživatelská aplikace je uložena ve složce *\applications\<síťová karta>*, v tomto případě byla použita karta *nic_10g8*. Složka obsahuje další 3 podadresáře. *Build* obsahuje výstupní soubory po syntetizaci a implementaci – soubor *fpga.bit*. Firmware pro kartu musí být soubor **.msc* pomocí příkazu

```
promgen -p mcs -data_width 16 -w -u 0 fpga.bit -o fpga.msc
```

Podsložka *sim* obsahuje zdrojové soubory pro provedení simulace. Aktuálně obsahuje zdrojový soubor *testbench.vhd*. V poslední podložce *src* se nachází zdrojové a konfigurační soubory aplikace.

Ve složce *config* je umístěn *config.tcl*, který upravuje část nastavení NetCOPE. Nejdůležitější z nich je frekvence celého systému. Ta se nastavuje pomocí děličky, která dělí výchozí frekvenci – 1400 MHz. Výchozí nastavení děličky je 17.5, to odpovídá frekvenci 80 MHz. Dělička *CLK0_DIV* je typu reálné číslo, tudíž je potřeba zapisovat v případě celé hodnoty i desetinnou část (např. 11.0).

Složka *core* obsahuje zdrojová data uživatelské aplikace. Nachází se zde zdrojový soubor *application_core.vhd*, který slouží jako vstupní bod pro uživatelské aplikace. V této implementaci jsou zde uloženy zbylé soubory aplikace AES a *application_core.vhd* je upraven tak, aby do sebe napojil entitu AES jako komponentu. Bude popsáno v podkapitole.

Ve složce *src* je dále soubor *Modules.tcl*, ve kterém jsou uloženy cesty ke všem zdrojovým souborům uživatelské aplikace. Pokud jsou soubory uloženy ve složce *core*, tak se cesta k souboru zadává podle příkladu, který je uveden v souboru *Modules.tcl*.

Poslední za zmínku stoji *Makefile*, který je určen pro linux operační systémy (v tomto případě CentOS). Příkazem *make sim* se spouští simulace s testbenchem, který je v nadřazené složce ve složce *sim*. Příkaz *make* spouští syntézu a následné generovaní bitového zdrojového kódu pro čip. Tento proces je vykonáván pouze v příkazovém řádku za pomocí tcl příkazů, kterými lze ovládat Vivado. Příkaz *make clean* vymaže již soubory, vytvořené při syntéze.

Zbytek souborů nebyl upravován, a tudíž není jejich význam v téhle práci uveden. Pro případný zájem si prosím funkci přečtěte v dokumentaci NetCOPE, která se nachází na rootu ResourceCD v složce *documentation*.

3.5.1 Application_core

Tato kapitola obsahuje popis provedených změn v souboru *application_core.vhd*. Definice signálů a jiných částí bylo ponecháno. Původní soubor popisuje, jakým způsobem se zapisuje do registrů FPGA čipu a následně jejich opětovné čtení. Myšlenka byla převzatá a následně upravena dle potřeby komponenty AES od vývojářů NetCOPE. Zásluhy a autorství patří jim.

Komunikace uživatelské aplikace umístěné na čipu s hostujícím počítačem probíhá pomocí sběrnice MI32. Ta slouží k propojení jednotlivých modulů celého NetCOPE – řídící síť, před kterou se posílají příkazy k jednotlivým modulům. Pracuje s 32bit adresami a jedná se o sběrnici s nízko rychlostní sběrnici. Proto není vhodná pro přenos velkých objemů dat. Je vhodnější použít modul DMA, který pracuje s RAM pamětí hostujícího počítače pomocí protokolů FrameLink, FrameLinkUnaligned (FLU) a FLU Plus Buses. V této implementaci se používá přenos pomocí sběrnice MI32, protože ovládání je jednodušší.

Na začátku bylo potřeba definovat komponentu *AES* včetně několika skupinek signálů, které reprezentují registry čipu, a pak řada 1bit signálů určené k řízení. Signály, který byly dodatečně definovány, jsou vypsány v následující tabulce č.7.

Tabulka 7: Přehled signálů Application_core

Přehled signálů					
Jméno	Délka	Použití	Jméno	Délka	Použití
reg_wordIN0	32	vstupní vektor	we_wordIN0	1	značky povolující vlastní zápis do registrů
reg_wordIN1	32		we_wordIN1	1	
reg_wordIN2	32		we_wordIN2	1	
reg_wordIN3	32		we_wordIN3	1	
reg_wordOUT0	32	výstupní vektor	we_key0	1	
reg_wordOUT1	32		we_key1	1	
reg_wordOUT2	32		we_key2	1	
reg_wordOUT3	32		we_key3	1	
reg_key0	32	šifrovací klíč	dl_wordIN0	1	značky signalizující načtení dat do registrů
reg_key1	32		dl_wordIN1	1	
reg_key2	32		dl_wordIN2	1	
reg_key3	32		dl_wordIN3	1	
regSel_wordIN0	1	značky signalizující aktuálně vybraný registr pro zápis	dl_key0	1	
regSel_wordIN1	1		dl_key1	1	
regSel_wordIN2	1		dl_key2	1	
regSel_wordIN3	1		dl_key3	1	
regSel_FLAGS	1		aes_dataIN	128	rozhraní komponenty AES
			aes_dataOUT	128	
regSel_key0	1		aes_key	128	
regSel_key1	1		aes_encrypt	1	
regSel_key2	1		aes_dataReady	1	
regSel_key3	1		aes_READY	1	
			RegAes_FLAGS	32	značky pro komunikaci s hostem

Pracuje se s 32bit adresami, tudíž registry (signály) musí být délky 32 bitů. Ergo data a klíč šifry potřebuje 4 registry. Číslování začíná od LSB části – číslování jde zprava do leva. Skupina signálů *reg_wordIN** reprezentuje registry, do nichž je uložen blok vstupních dat (128 bitů). Analogicky signály *reg_wordOUT** označují výstupní blok dat. Skupina se signály *reg_key** slouží pro načtení šifrovacího klíče. Další větší skupinou jsou signály *regSel_**, které se podílejí na řízení aplikace. Používají se k určení, se kterým registrem se aktuálně pracuje v transakci čtení/zápis. Signály *we_** slouží k řízení zápisu do jednotlivých registrů. A signály *dl_** uchovávají informaci o zapsání dat do registrů.

Umožňuje to sledovat, do kterých registrů na vstupu již bylo zapsáno. Je potřeba před spuštěním AES aplikace počkat na všechny 4 části příchozích dat – klíč nebo vstupní data. Signály *aes_** slouží k připojení na komponentu *AES*.

Posledním definovaným signálem je *RegAes_FLAGS* o délce 32 bitů. Slouží pro komunikaci s hostitelským počítačem pomocí značek, které byly definovány. Využity jsou momentálně poslední 3 bity signálu. Číslování začíná od LSB bitu. Bit na pozici 0 je propojen se signálem *encrypt* komponenty *AES*. Nastavuje se pomocí něj, zda bude komponenta šifrovat (log. 1) nebo dešifrovat (log. 0). Výchozí nastavení je log. 1. Bit na pozici 1 je propojen s komponentou přes signál *dataREADY* a bit na pozici 2 se signálem *READY*. Obě pozice jsou pouze pro čtení a nastavovat lze jen bit na pozici 0. Myšlenka komunikace je pak taková, že aplikace na hostitelském počítači se ptá dokola (načítá stále tento registr) a zjišťuje značky, které jsou nastavené. Jednoduchý příklad, aplikace načítá data z registru do té doby, než se objeví na 3. bitu log. 1 – komponenta *AES* je připravena ke zpracování dat. Začne přenos dat do registrů. Probíhá další smyčka, při které se načítá registr se značkami a až se na 2. bitu objeví log. 1, znamená to, že na výstupu komponenty jsou připraveny validní data a začne přenos dat z karty do počítače. Cyklus je dokončen a začíná znovu tím, že aplikace načítá opět značky a čeká na 3. bit. Šifrování lze přepnout kdykoliv odesláním log. 0 do registru na 0. pozici. Podmínky, které řídí tuto implementaci, bohužel vnáší do systému zpoždění několik taktů.

Jako první byly implementovány dekodéry adres. Zvlášť pro čtení a zápis. Dekodér pro psaní je jednoduší. Obsahuje podmíněné přiřazení do signálu. Celý blok sleduje signál *MI_ADDR* ve které je uložena adresa registru. Blok kontroluje jen část adresy – rozpětí přidělených adres není natolik velký, aby bylo potřeba provádět kontrolu všech 32 bitů. Konkrétně se sledují bity na pozici 2–5 zprava (LSB). Následně se provede přiřazení dat na výstup sběrnice, signál *MI_DRD*, z registru, jemuž odpovídá konkrétní adresa.

```
MI_DRD <= reg_wordOUT0              when "0100",
          reg_wordOUT1              when "0101",
…
```

Seznam adres je uveden v tabulce č. 8. Registry určené ke čtení jsou signály se jménem *reg_wordOUT** a *RegAes_FLAGS*.

Tabulka 8: Přehled adres registrů

Přehled adres registrů	
Jméno	Adresa (HEX)
reg_wordIN0	0200_0000
reg_wordIN1	0200_0004
reg_wordIN2	0200_0008
reg_wordIN3	0200_000C
reg_wordOUT0	0200_0010
reg_wordOUT1	0200_0014
reg_wordOUT2	0200_0018
reg_wordOUT3	0200_001C
reg_key0	0200_0030
reg_key1	0200_0034
reg_key2	0200_0038
reg_key3	0200_003C
RegAes_FLAGS	0200_0020

Dekodér určený pro zápis do registru je už složitější. Realizováno pomocí procesu, který má v citlivostním seznamu signál *MI_ADDR* – při každé změně adresy se proces spustí a provede. Na začátku se vypnou všechny signály *regSel_**, které určují, který signál (registr) je právě vybrán. Podobným blokem (*case*) jako v předchozím dekodéru se určí podle bitů adresy na pozici 2–5 registr pro zápis tak, že se příslušný signál *regSel_** aktivuje. V tomto místě se víceméně určují adresy jednotlivých registrů. Blok procesu je tímto uzavřen.

```
case MI_ADDR(5 downto 2) is
         when "0000"    => regSel_wordIN0 <= '1';
         when "0001"    => regSel_wordIN1 <= '1';
…
```

V dalším bloku příkazů se povoluje zápis do jednotlivých registrů. Informaci o povolení udržují signály *we_**. Povolení se ukládá jako konjunkce signálů *regSel_** a *MI_WR*, který je aktivní v případě zapisovací transakce. Pomocí těchto signálů je v dalším procesu rozhodnuto, do kterého registru zapsat data. Protože v tomto okamžiku je jasné, který signál byl vybrán adresou *MI_ADDR* a že se provádí zápis dat (*MI_WR*). Následující proces se spouští s vnitřními hodinami *core_clk*, dále podmínka určuje, že vykonávání procesu se provádí při náběžné hraně hodin. Nyní je to několik vnořených podmínek. První, testuje, jestli je aktivní reset aplikace (signál *mi_reset_i*). Pokud ano, veškeré vstupní registry i s řídícími signály *dl_**. Proces se ukončí a začne nanovo. V případě, že reset nebyl aktivován, se přechází k podmínkám řídící zápis. Každý registr

má svou kontrolní podmínku, které se musí všechny projít. Kontroluje se již zmíněný signál *we_** přidružený konkrétnímu registru, který vypovídá o tom, že byl vybrán zrovna tento registr a byl poslán požadavek pro zápis. Kromě samotného zápisu jsou aktivovány řídící signály *dl_**, které signalizují zápis do registru. Zdroj dat je v signálu *MI_DWR*. Jedinou výjimkou je registr *RegAes_FLAGS*, který má pro zápis určený momentálně jen jeden bit na pozici 0. Následují další dvě podmínky, které restartují (vypínají) signály *dl_**. Spouští se tehdy, když všechny 4 registry byly naplněny daty. Tyto bloky jsou umístěny tady, a ne v místě, kde se předávají vstupní data do komponenty *AES*, protože tam to způsobuje kolizi řízení těchto signálů. Podmínky se spouští se zpožděním jednoho taktu, protože při zápisu posledního registru je sice vykonán příkaz, který zapne poslední *dl_** signál dané skupiny, jenže veškeré přiřazení do signálu se provádí až na konci procesu. Tudíž při kontrole podmínky nebylo uvedeno v platnost přiřazení do signálu. To se projeví až v dalším spuštění procesu.

Další dva procesy v aplikaci se věnují vkládání vstupních dat do komponenty. Oba dva řídí hodiny *core_clk*. První vkládá do komponenty klíč. Vždy při náběžné hraně hodin kontroluje, zda byly načteny všechny části klíče (pomocí *dl_key**) a pak všechny části spojí a uloží do signálu *aes_key*. Druhý proces probíhá stejně až s tím rozdílem, že se zde pracuje blokem dat určených ke zpracování. Data v podmínce se předávají do signálu *aes_dataIN* a v podmínce je kromě *dl_wordIN** signálu ještě *aes_READY*, který předá data pouze tehdy, jsou-li vstupní data načteny a komponenta *AES* je připravena ke zpracování nových dat.

Poslední proces rozděluje výstupní data z komponenty do jednotlivých registrů a to tak, že při každé náběžné hraně hodin kontroluje, zda signál *aes_dataREADY* je aktivní. Potom při aktivaci rozdělí jednotlivé části výstupu *aes_dataOUT* do příslušných registrů. Číslováno opět vzestupně zprava doleva. Současně s podmínkou aktualizuje značky v *regAES_FLAGS* – signály *aes_READY* a *aes_dataREADY*. Každé takové přiřazení řídícího signálu v procesu má za následek postupné zpožďování celé aplikace, neboť veškeré přiřazení do signálu je prováděno vždy na konci procesu.

Souhrn všech kroků. Inicializace aplikace – veškeré signály jsou ve výchozí pozici a procesy bez citlivostního seznamu se spouští. *RegAES_FLAGS* obsahuje pouze bit signalizující šifrování, příznak READY je vypnutý. Následně se nahraje klíč na patřičné adresy. Pořadí nahrávání částí je irelevantní, ale musí splňovat podmínku, že jednotlivé částí musí být nahrány do registrů odpovídající poloze částí (prvních 32 bitů klíče musí být nutně v registru, který reprezentuje prvních 32 bitů – *reg_key3*). Probíhá kontrola adresy v dekodéru, než se najde shoda. Aktivuje se příslušný signál *regSel_key**, dále se povolí zápis do registru (*we_key**). Následně se prochází proces, při kterém se najde registr, jemuž přísluší signál *we_key** a provede se zápis do registru a aktivace *dl_key**. Současně probíhá vyhodnocování řady jiných podmínek, které aktuálně nejsou splněny. Po načtení poslední části registru se vyhodnotí podmínka v procesu řídící zápis klíče do komponenty jako pravdivá a registry se spojí a výsledný klíč se předá komponentě. O takt později se v jiném procesu vyhodnotí taktéž podmínka, že všechny části byly úspěšně načteny a tyto signály *dl_key** vymaže.

Probíhá expanze klíče a o několik taktů později se aktivuje příznak *READY*. Hostující počítač, který mezitím dokola načítal signál se značkami obdrží tuto skutečnost a začne nahrávat data ke zpracováním obdobným způsobem. Pravidla pro načítání jsou stejná. Při načtení poslední části se opět vyhodnotí podmínka, která navíc sleduje ještě

příznak *READY* a pokud jsou všechny signály aktivní, předá data komponentě. Jiná podmínka opět se zpožděním jednoho taktu vymaže příslušné signály *dl_wordIN**. Hostitelský počítač po přenosu dat opět kontroluje značky a čeká na příznak *dataREADY*. Mezitím probíhá transformace dat a proces, jenž řídí dělení výstupních dat do registrů, čeká taktéž na příznak *dataREADY*. Jakmile jej obdrží, rozdělí dle definice výstupní data do jednotlivých registrů. Příznaky *dataREADY* a *READY* však řídí nezávisle na dělní výstupních dat. Nastaví příznak *dataREADY* a hostitelský počítač začne s přenosem dat ven z karty. Tentokrát budou adresy odpovídat v dekodéru pro čtení dat (kontrola probíhá ve všech dekodérech) a data z registru odpovídající adrese pošle na sběrnici. Po dokončení přenosu hostitelský počítač začne opět sledovat značky a čeká na příznak *READY*, který během přenosu přešel do aktivního stavu. Proces nahrávání dat se opakuje.

Směr transformace dat (šifrování/dešifrování) lze kdykoliv měnit za běhu. Stačí na příslušnou adresu zapsat celý 32bit signál s posledním bitem odpovídající požadavku. Aplikace pro zápis do registru značek sleduje jen poslední bit, takže je irelevantní, co se posílá ve zbytku signálu. Pochopitelně pokud se změna směru transformace má uplatnit pro aktuálně zasílaná data, je potřeba tento požadavek zaslat kdykoli během přenosu, nejpozději však před poslední částí dat. Jinak se změna nestihne projevit a nastane tak pro všechny další příchozí data.

3.6 Testování aplikace

Tato kapitola popisuje chování entity *AES*, její části a uživatelskou aplikaci *application_core* v simulačním prostředí Vivada a vyobrazuje signály ve *waveformu*. Pro testování se používá speciálně napsaný program *testbench* (dále jen TB), který je napsán tak, aby simuloval vnější prostředí pro aplikaci a zjistil, jakým způsobem se bude aplikace chovat v konečné fázi.

TB jsou dva, jeden (TB1) pro entitu *AES*, který byl použit při jejím vývoji (Notebook, OS WIN10 x64, Vivado 2016.3). Druhý TB (TB2) je od NetCOPE – prováděno na syntézním serveru fakulty (OS CentOS x64, Vivado 2013.4). Implementace TB1 obsahuje jen napojení rozhraní *AES* a generátor hodiny. TB2 obsahuje pouze modifikaci již existujícího TB v NetCOPE – práce s registry přes rozhraní MI32.

3.6.1 TB1

Obrázek č. 17 vyobrazuje záběr z TB1. Lze v něm vidět data vnitřních signálů. Navíc je na konci vyobrazena první iterace. Začátek šifrování začíná v čase 300 ns a končí v čase 700 ns. Samotné zpracování dat se dokončí během 1 taktu a celý cyklus (perioda s jakou se předávají vstupní data) trvá 2 takty. Na začátku je zobrazené rozhraní, vstupní data (*dataIN*) jsou předána v čase 300 ns a na výstupu se objeví v čase 500 ns spolu s impulzem na signálu *DataReady*.

Pro testování byly použity data a šifrovací klíč podle [9], aby bylo možné sledovat jednotlivé kroky šifrování. Pro dešifrování byl použit jako vstup výsledek šifrování.

Vstupní data: 32 43 f6 a8 88 5a 30 8d 31 31 98 a2 e0 37 07 34

Šifrovací klíč: 2b 7e 15 16 28 ae d2 a6 ab f7 15 88 09 cf 4f 3c

Mezitím se do pole *TEMP* uložily výsledky ze všech iterací. Pozice 0 obsahuje výsledek předzpracování, kdy se k datům přičítá pouze šifrovací klíč. Každá pozice odpovídá startovacímu vektoru o číslu větší iteraci – podle [9] (strana 33 a 34). Modře označené komponenty na konci obrázku zobrazují postupnou transformaci dat v iteraci označené číslem 1 v [9] na straně 33.

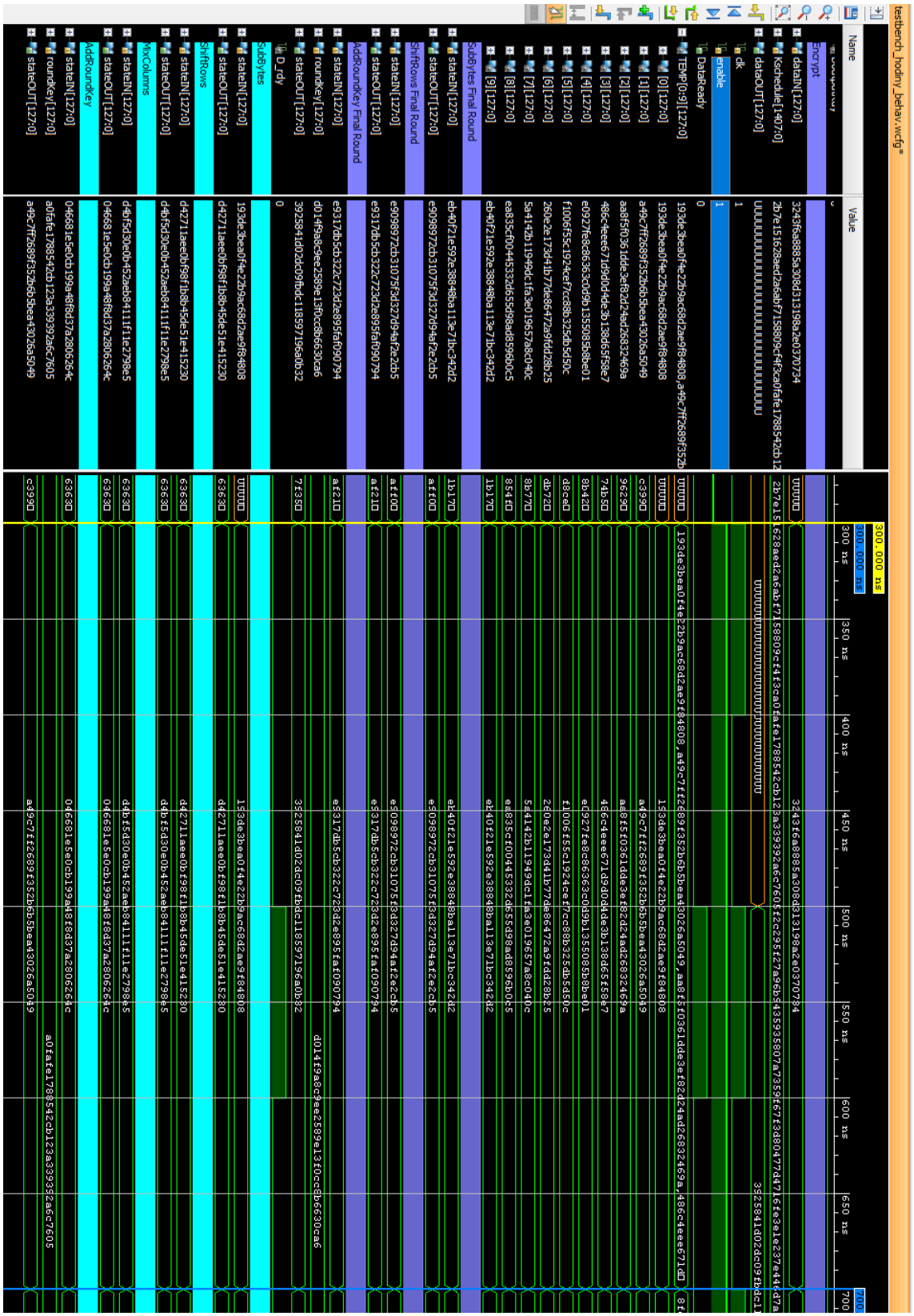
Na obrázku č. 18 je druhý směr – dešifrování. Popis je podobný jako u šifrování. Na začátku je zobrazené rozhraní, dešifrování trvá stejně jako šifrování (čas 300 ns až 700 ns). Dešifrování začíná transformací *AddRoundKey*, které je předán „poslední[11]“ iterační klíč (iterační klíče jsou použity v opačném pořadí) do pole *TEMP* na pozici 0. Jednotlivé transformace jsou v iteraci posunuty, takže poslední iterace (vektor na pozici 9 v *TEMP*) končí inverzí transformace MixColumns. Pro šifrovací směr je to v [9] na straně 33 iterace číslo 1 a 3. sloupeček. Následují inverzní transformace ShiftRows a SubBytes a na výstupu je pak poslán součet výstupu ze *InvSubBytes_cmp_FinalRound* s klíčem. Dále komponenty označeny modře zobrazují průchod 1. iterací dešifrování.

Řízení aplikace je vyobrazeno na obrázku č. 19. Popisuje řízení dešifrování. Předávání vstupních dat mezi nadřazenou entitou a komponentou je zpožděno o jeden takt. Vstupní data jsou do nadřazené entity předány v čase 100 ns a do komponenty pak v čase 300 ns. Výsledek je předán z nadřazené entity okamžitě, jakmile se objeví na výstupu komponenty. Detailnější popis je následovný. Jako hlavní spouštěč lze považovat signál *READY*. V okamžiku jeho aktivace se načítají data do nadřazené entity. Společně s daty je aktivován signál *dataLoaded*. O takt později se spustí dešifrování – signál *startSystem* a s touto událostí jsou vstupní data předány dešifrovací komponentě a signál *READY* je deaktivován. Během půl taktu se deaktivuje *startSystem*. Za dalšího půl taktu (1 takt po spuštění – 500 ns) komponenta signalizuje signálem *decryptionRdy* dokončení transformace a validní data na svém výstupu. Validní data jsou dále signalizována uvnitř entity signálem *DoutputRdy* a pomocí tohoto signálu se řídí entita *AES*, včetně řízení signálu na výstupu – *dataReady*. Tento signál je aktivní po celou dobu, kdy jsou dostupné na výstupu data. Na rozdíl od signálů *DoutputRdy*, které se automaticky po jednom taktu deaktivují. Společně se aktivuje *READY* a celý proces se opakuje s periodou 2 takty.

Transformace dat, ať už jakýmkoliv směrem, lze provést během jednoho taktu. Další takt je režií této implementace. Vhodnou úpravou, respektive jinou implementací řídící části, by mělo být možné zkrátit délku výsledného procesu na jeden takt.

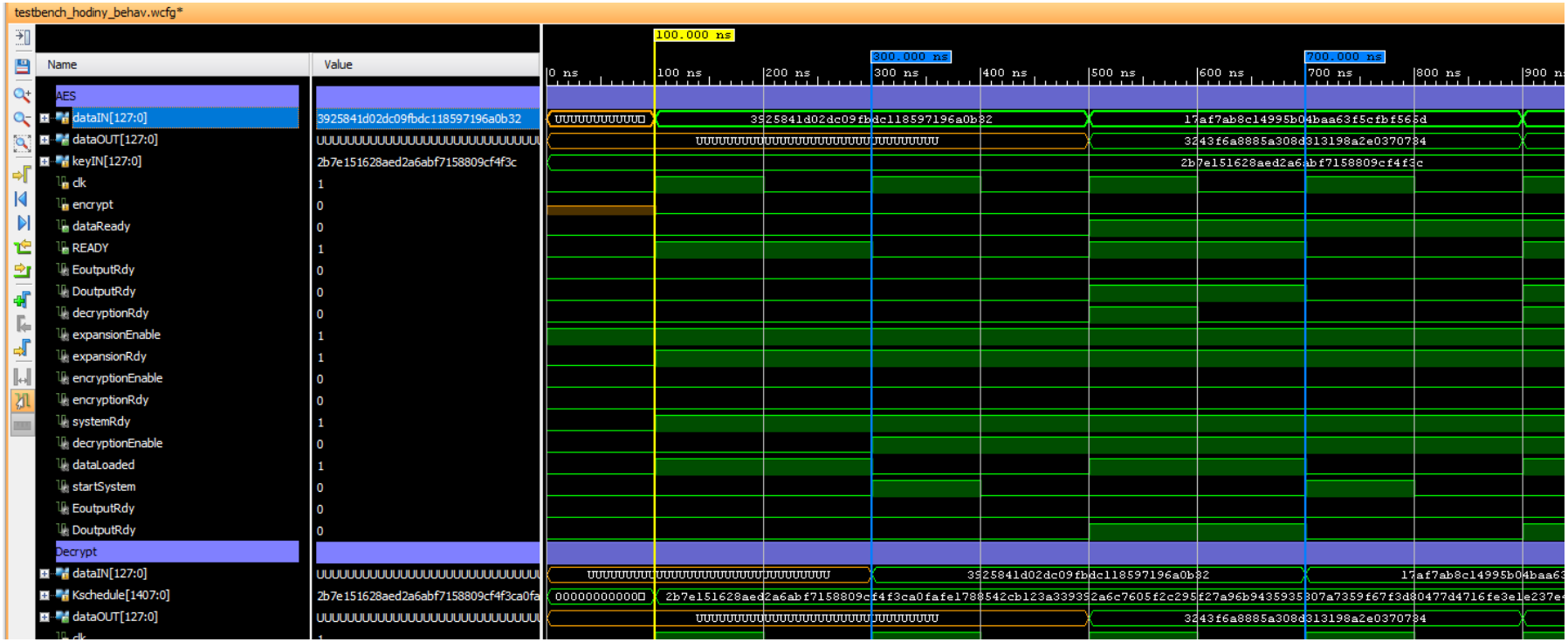
U šifrování a dešifrování lze vidět, že pracují správně a poskytují validní data.

[11] Poslední klíč je myšlen ten, který byl použit při šifrování jako poslední



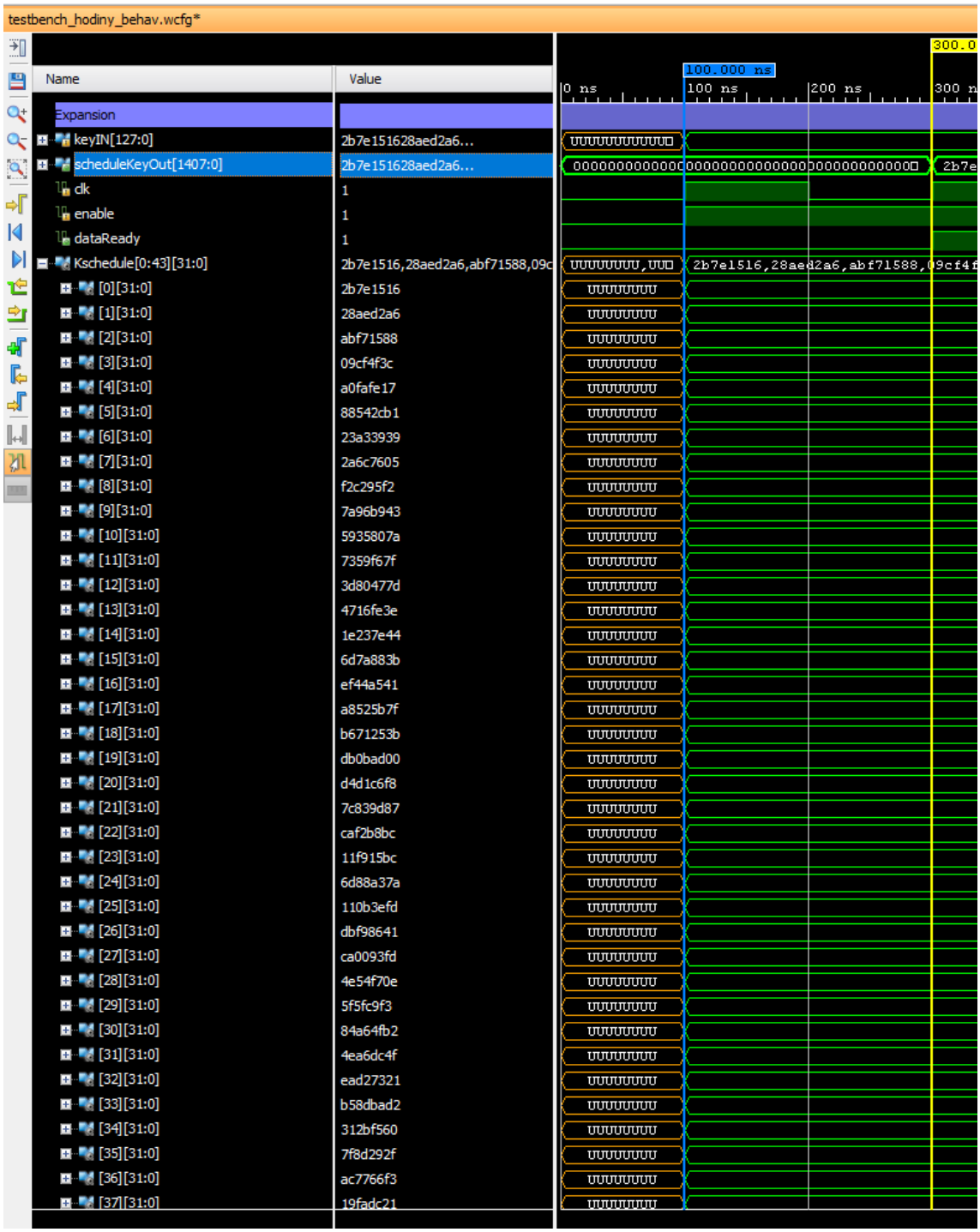
Obrázek 17: Výpis průběhů signálů entity rounds_modul (šifrování)

Obrázek 18: Výpis průběhů signálů entity invRounds_modul (dešifrování)



Obrázek 19: Výpis průběhů řídících signálů entity AES

Expanze klíče je vyobrazena na obrázku č. 20. Na začátku je zobrazeno rozhraní a pak vlastní expandovaný klíč jako pole slov. Pořadí slov v poli *Kschedule* je totožné s [9] (strana 27). Expanze trvá jeden takt. Předání dat je v čase 100 ns a v čase 300 ns se aktivuje signál *dataREADY* a expandovaný klíč je předán na výstup, na kterém byly do této doby samé nuly. Implementace expanze byla úspěšná – poskytuje validní slova. Na obrázku č. 20 lze vidět prvních 30 slov.



Obrázek 20: Výpis průběhů signálů expanze KeyExpansion

3.6.2 TB2

Protože TB provedený na serveru obsahuje taktéž entitu *AES*, která je stejná, tak bude v této kapitole ukázán pouze záběr z *application_core* – předávání dat dovnitř a ven. Na obrázcích č. 21 a č. 22 je zobrazeno jen několik vybraných signálu, na kterých probíhají změny a jsou něčím zajímavé.

Při testování byl použit stejný šifrovací klíč jako u TB1, ale jiná vstupní data. Dešifrovaná data (bylo testováno dešifrování) byla ověřena programem na stránkách http://aes.online-domain-tools.com/.

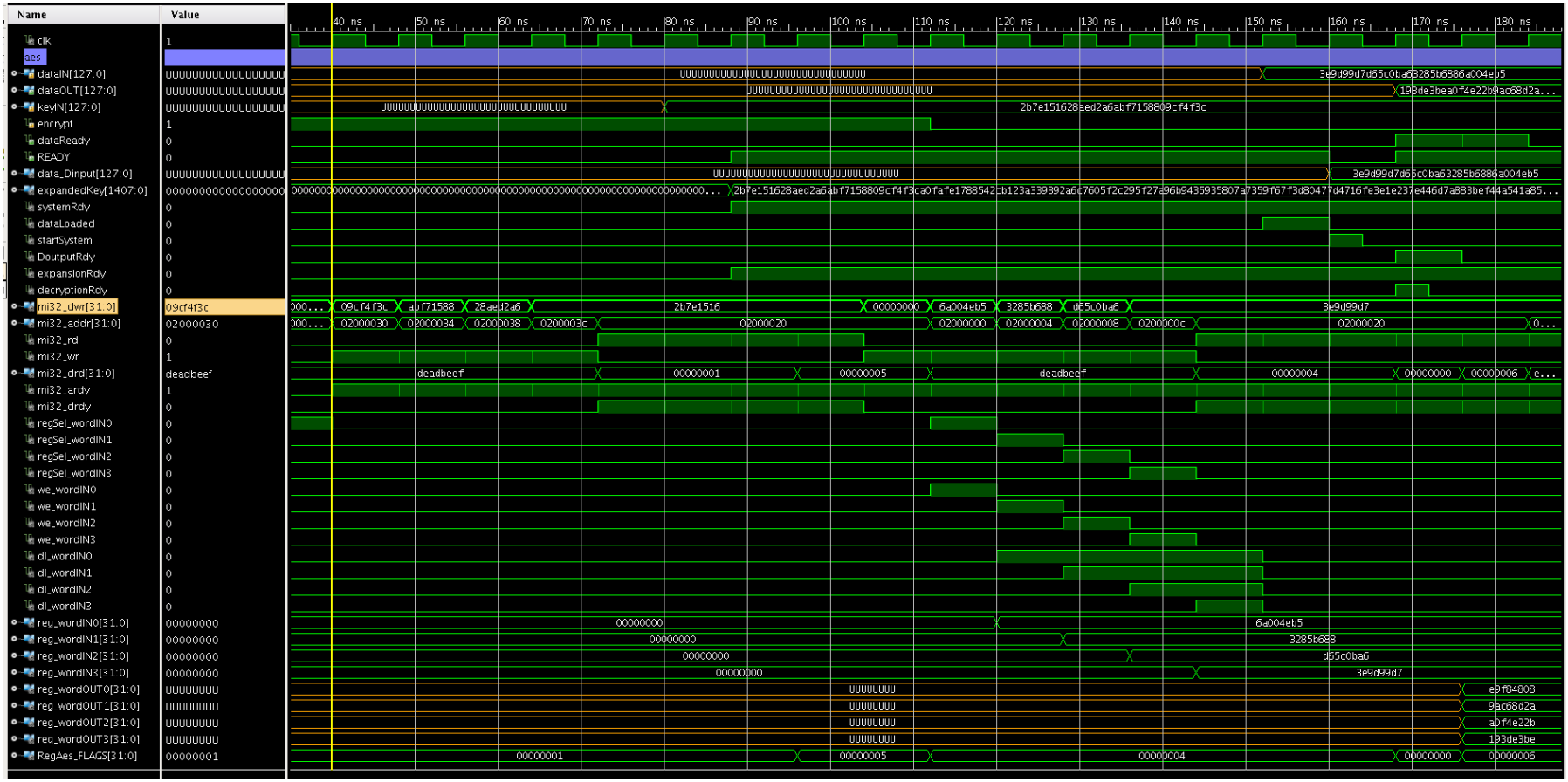
Vstupní data: 3e 9d 99 d7 d6 5c 0b a6 32 85 b6 88 6a 00 4e b5

Očekávaná výstupní data: 19 3d e3 be a0 f4 e2 2b 9a c6 8d 2a e9 f8 48 08

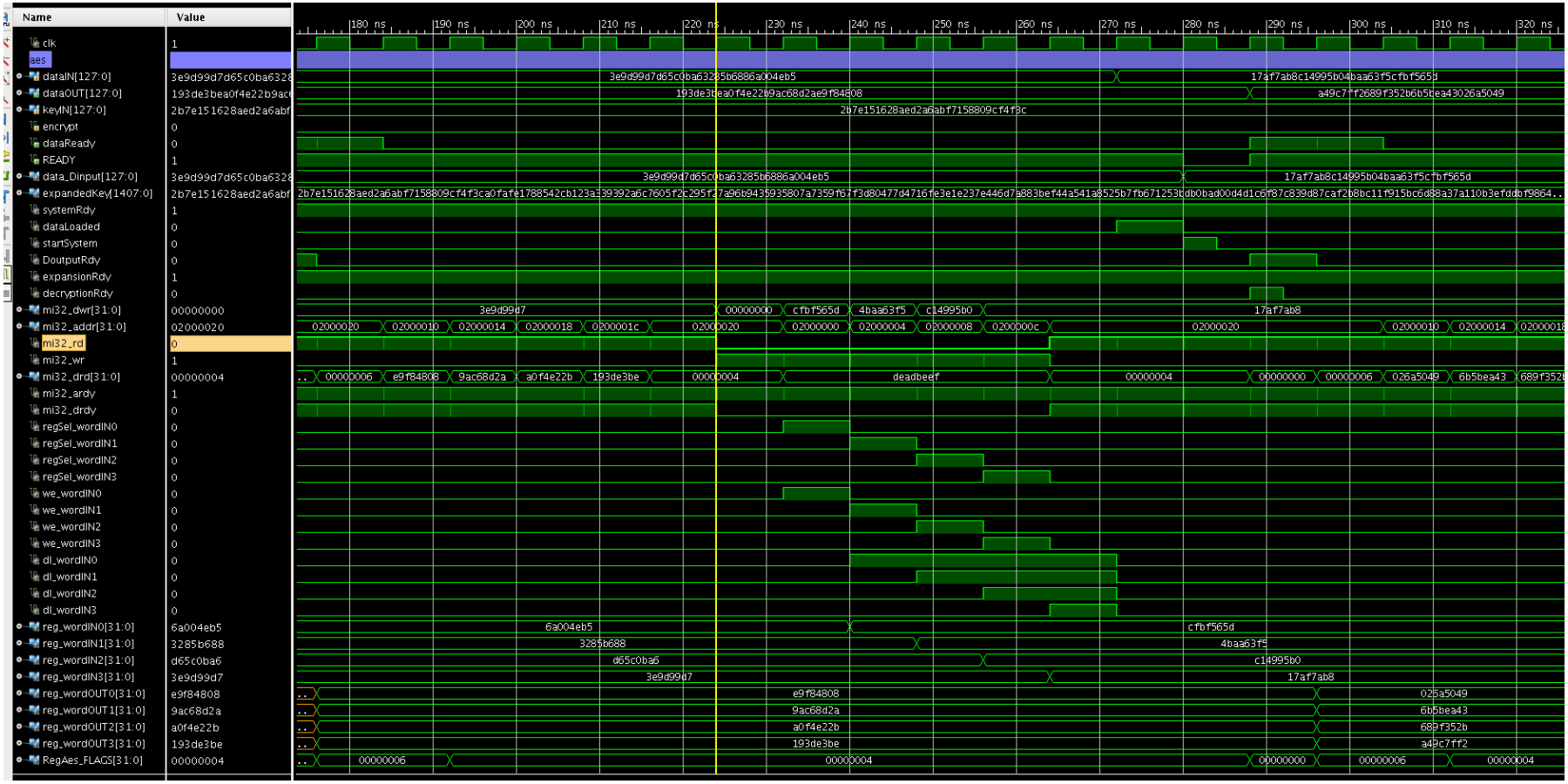
V čase 40 ns probíhá načítání šifrovacího klíče. V signálu *mi32_addr* jsou postupně předány adresy registrů klíče a v signálu hned na ním (*mi32_dwr*) jsou předány části šifrovacího klíče. O takt později, po načtení poslední části, je klíč předán expanzi – *keyIN*. O další takt později (v čase 88 ns) je dokončena expanze (spuštěn signál *READY*) a opět se zpožděním jednoho taktu je v signálu *RegAes_FLAGS* hodnota „5“, která mimo jiné signalizuje aktivní signál *READY*. Mezitím byl po dokončení načítání klíče opakovaně načítán tento registr značek a jakmile byla zjištěna přítomnost značky *READY*, začal přenos dat se zpožděním jednoho taktu – čas 104 ns. V tomto čase jsou aktivní příznaky pro zápis na sběrnici MI32. V adrese se postupně střídají adresy jednotlivých registrů vstupních dat. S adresami lze vidět, jak se v dekódují v dekodéru adres a následně povolení zápisu do registru, aktivace signalizace pro načtená data a pak data v jednotlivých registrech. Signály *regSel_wordIN*, *we_wordIN*, *dl_wordIN* a *reg_wordIN*. Po načtení posledního registru se po jednom taktu v čase 152 ns spustí transformace dat. Signály *dataIN* a *dataLoaded*. Průběh už je pak stejný jak u TB1. Se signálem *READY* se mění i signál značek.

Při načtení posledního registru je okamžitě načítán registr značek a kontroluje se příznak na *dataReady*. V čase 176 ns se data na výstupu dostanou na výstupní registry, to je zpožděno oproti výstupu entity *AES* o jeden takt. V tomtéž čase je také detekován příznak *dataReady* a o takt později (184 ns) je započat přenos dat ven z registrů *reg_wordOUT*. Tento přenos je dokončen v čase 208 ns. Po dokončení je okamžitě kontrolován registr značek na příznak *READY*, který měl dost času se aktivovat, zatímco probíhal přenos výstupních dat. A proto okamžitě v dalším taktu (232 ns) začíná přenos nových vstupních dat. Dál už proces probíhá stejně.

Tento test měl za úkol otestovat už pouze celkovou funkčnost entity (šifrování bylo ověřeno v TB1) – dekodéry adres, předávání dat registrům a celkovou automatizaci. To bylo ověřeno (obrázky č. 18 a 19) a je považováno za funkční.



Obrázek 21: Výpis průběhů signálů v Application_core – část A



Obrázek 22: Výpis průběhů signálů v Application_core – část B

3.7 Výsledky implementace

Tato podkapitola pojednává o výsledcích syntézy (rychlost zpracování a využití zdrojů čipu FPGA). Správnost zpracování dat byla ověřena v předchozí podkapitole 3.6.

Syntéza kódu VHDL je proces, při kterém se kompiluje zdrojový kód a ověřuje se, zda je možné použité konstrukce použít na fyzickém čipu. Ne všechny konstrukce a příkazy, které jazyk obsahuje, lze použít mimo simulační prostředí (např. práce se soubory, některé příkazy *wait*).

Stejně jako u simulace byly provedeny 2 syntézy. Jedna (SYN1) byla provedena při vývoji na notebooku a druhá (SYN2) byla provedena v rámci generování firmwaru na syntetizačním serveru.

3.7.1 SYN1

Syntéza byla provedena ve Vivado 2016.3 64bit, stejně jako TB1. Na desce Artix-7 AC701 Evaluation Platform, xc7a200tfbg676-2. 400 dostupných I/O pinů, 269200 registrů a 134600 LUT záznamů.

V této fázi byl syntetizován pouze kód entity *AES* bez NetCOPE. Nejvyšší frekvence, které bylo dosaženo 476 MHz (2,1 ns). Nejhůře dopadla část, kdy se přenáší výstupní data z komponent pro šifrování nebo dešifrování na výstup celé entity. Časová rezerva (WNS) je 0.057 ns. Při těchto parametrech je teoretická rychlost entity (frekvence 476 MHz, doba trvání šifrování 2 takty, délka dat 128 bitů) 30,464 Gbit/s (vzorec 3.1). Pokud by se podařilo upravit řízení entity tak, aby celý cyklus transformace trval jen jeden takt, byla by teoretická rychlost dvojnásobná.

$$R = \frac{f \cdot d}{t} \tag{3.1}$$

kde

R je rychlost zpracování dat v [bit/s]

f je pracovní frekvence čipu v [Hz]

d je délka zpracovávaných dat v [bit]

t je počet taktů cyklu[12] zpracování dat [-]

Využití zdrojů je v tabulce č. 9. Nejvíc zabírá tabulku LUT implementace entity *KeyExpansion*. Celkem je využito přibližně 14 % LUT.

[12] Délka cyklu je chápána jako počet taktů, které uběhnou mezi jednotlivými předáními vstupních dat.

Tabulka 9: Využití zdrojů čipu Artix-7 (Vivado 2016.3)

Využití čipu Artix 7					
Celek/část	Záznamy LUT (134600)	Registry (269200)	F7 Multiplexery (67300)	F8 Multiplexery (33650)	I/O piny (400)
Expanze	17468	1	536	124	0
Šifrování	156	128	2560	1280	0
Dešifrování	374	128	2304	1024	0
AES	18383 (14%)	1025 (0,4%)	5400 (8%)	1428 (7%)	388 (97%)

3.7.2 SYN2

Tato syntéza byla provedena na syntézním serveru VUT ústavu UTKO, Vivado 2013.4 64bit. Čip Virtex-7, xc7vx690tffq1157-2, 600 dostupných I/O pinů, 866400 registrů a 433200 LUT záznamů.

V tomto kroku byla provedena syntéza včetně implementace a generování firmwaru. Hlášení (reports) byla vygenerována při různých krocích vytváření výsledného kódu. První hlášení bylo vygenerováno po dokončení syntézy a druhé hlášení bylo vygenerováno po dokončení implementace (o jeden vývojový krok po syntéze), viz tabulka č. 10.

Tabulka 10: Využití zdrojů čipu Virtex-7 pro NetCOPE

Využití čipu				
Krok	Záznamy LUT (433200)	Registry (866400)	F7 Muliplexery (216600)	F8 Multiplexery (108300)
Syntéza	127441 (29,4%)	117256 (13,5%)	5435 (2,5%)	615 (0,5%)
Implementace	123880 (28,6%)	115325 (13,3%)	5434 (2,5%)	615 (0,5%)

Základní frekvence je 80 MHz. Cyklus transformace dat z TB2 trvá 120 ns, to je 15 taktů. Při těchto frekvencích je teoretická maximální rychlost transformace 682 Mbit/s. Pokud by se podařilo vylepšit řízení aplikace a ušetřit 2 nebo 3 takty, rychlost by se zvýšila cca od 200 Mbit/s. Pro přenos dat by bylo stejně vhodnější použít DMA modul než sběrnici MI32.

Byla provedena série několika syntéz a implementací, které ale nebyly testovány na hardwaru. Podle hlášení o časování (Timing Report) je při frekvenci 160 MHz časová rezerva pouhých 0,002 ns. Při této frekvenci je teoretická maximální rychlost transformace dat 1365 Mbit/s.

Pro srovnání se syntézou v podkapitole 3.7.1 SYN1 byla provedena syntéza pouze aplikace *AES* ve Vivadu na syntézním serveru s čipem Virtex-7. Dosáhlo se maximální frekvence 689 MHz (1,45 ns). Pro tuto frekvenci dosahuje samotná aplikace *AES* rychlosti přibližně 44 Gbit/s.

Využití čipu Virtex 7 entitou *AES* je v tabulce č 11. Hodnoty využitých zdrojů se liší a proto byla vyhotovena tabulka č.12, ve které jsou výsledky využití čipu Artix 7. Tyto výsledky by měli odpovídat tabulce č. 9. Bohužel tomu tak není, pravděpodobně to bude způsobeno novější verzí Vivada v případě tabulky č. 9.

Tabulka 11: Využití čipu Virtex 7 (Vivado 2013.4)

Využití čipu Virtex 7					
Celek/část	Záznamy LUT (433200)	Registry (866400)	F7 Multiplexery (216600)	F8 Multiplexery (108300)	I/O piny (600)
Šifrování	6890	128	2448	1152	0
Dešifrování	2797	128	181	0	0
Expanze	11374	1	536	124	0
AES	21446 (5%)	1025 (0,1%)	3165 (1%)	1276 (1%)	388 (65%)

Tabulka 12: Využití čipu Artix 7 (Vivado 2013.4)

Využití čipu Artix 7					
Celek/část	Záznamy LUT (134600)	Registry (269200)	F7 Multiplexery (67300)	F8 Multiplexery (33650)	I/O piny (400)
Šifrování	6890	128	2448	1152	0
Dešifrování	2797	128	181	0	0
Expanze	11374	1	536	124	0
AES	21446 (16%)	1025 (0,4%)	3165 (5%)	1276 (4%)	388 (97%)

4 ZÁVĚR

Implementace autorova řešení v této práci je validní a funkční a zadání práce tak bylo splněno. Navržený program je schopný přijmout data z hostitelského počítače, zpracovat je a odeslat je zpět do počítače. Program dokáže zpracovávat data v obou směrech – šifrování a dešifrování. Dokáže taktéž expanzi šifrovacího klíče.

Program včetně NetCOPE nyní běží na základní frekvenci 80 MHz, při které je jeho teoretická rychlost zpracování dat 682 Mbit/s. Tato frekvence je limitovaná samotnou platformou NetCOPE, která by teoreticky mohla běžet na frekvenci 160 MHz. Rychlost by tak byla teoreticky 1365 Mbit/s. Tato frekvence nebyla testována na kartě, vychází z výsledků implementace kódu (hlášení Timing Report). Samotná entita *AES* dokáže teoreticky běžet (opět hlášení Timing Report) na čipu na frekvenci 689 MHz s rychlostí zpracování dat teoreticky 44 Gbit/s. Na čipu zabírá entita 5 % jeho prostředků a společně s NetCOPE potom 29 %.

Na čipu Artix (levnější varianta) dokázala entita *AES* běžet na teoretické frekvenci 476 MHz s teoretickou rychlostí 30 Gbit/s. Využití prostředků čipu je přibližně 16 %.

Řešení má i své nedostatky. Kód programu by bylo vhodné ještě více optimalizovat a využít tak více možností čipu FPGA. Dále program nedokáže pracovat s kartou víc, než je zpracování dat a jejich přenos přes sběrnici MI32, která je pomalá (přenos dat tam a zpět zabírá z celkové doby zpracování dat víc jak 50 %). V dalším kroku vývoje by bylo určitě použití pro přenos dat modulu DMA, dále také zpracování dat na síťovém rozhraní. To program také nedokáže.

Implementaci samotné šifry je dále potřeba rozšířit o podporu zbylých dvou délek šifrovacích klíčů (192 a 256 bitů) a také operačních režimů šifry. Řízení šifry by šlo upravit a dosavadní myšlenku zpracování dat změnit a docílit tak zrychlení logiky předávání dat uvnitř entity *AES* o jeden takt. Aktuální implementace řídící logiky (rozhodovací podmínky) zdržuje celý proces (rozhodování kdy jsou data validní, aktivace příslušných signálů atd.). Tento jev lze pozorovat na obrázku č. 21 v čase 160 ns. Ještě více dojde k ušetření času v *application_core*, kde se při načítání a spouštění transformace dat kontroluje několik signálů a proces se tak zpomalil minimálně o 2 takty. Tento nežádoucí jev lze upravit jiným návrhem vyhodnocovacích podmínek.

Za vhodné autor také považuje se pokusit o zvýšení propustnosti zpracovávaných dat tím, že by se spustilo více instancí entity *AES* najednou.

LITERATURA

[1] Burda, K. *Aplikovaná kryptografie*. Brno: Vutium, 11.04.20.13. ISBN/EAN: 9788021446120

[2] Dworkin, M. NIST SP 800-38A, Recommendation for Block Cipher Modes of Operation, <http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf>, 2001 [cit. 1.6.2017]

[3] Lipmaa, H., Rogaway, P., Wagner, D., Comments to NIST concerning AES Modes of Operations: CTR-Mode Encryption, <http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ctr/ctr-spec.pdf>, [cit. 1.6.2017]

[4] jgoedde, Xilinx: What is an FPGA? Field Programmable Gate Array, <https://www.xilinx.com/training/fpga/fpga-field-programmable-gate-array.htm>, 11/7/2012 [cit. 2.12.2016]

[5] Pinker, J. Poupa, M. *Číslicové systémy a jazyk VHDL*. Praha: BEN, 2006. ISBN: 80-7300-198-5.

[6] Xilinx, Synthesis and Simulation Design Guide < https://www.xilinx.com/support/documentation/sw_manuals/xilinx12_3/sim.pdf >, [cit. 3.12.2016]

[7] Liberouter, < https://www.liberouter.org >, [cit. 3.12.2016]

[8] Liberouter – NetCOPE, < https://www.liberouter.org/technologies/netcope/>, [cit. 3.12.201]

[9] FIPS-197, National Institute of Standards and Technology, Computer Security Resource Center, November 2011, <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf> [cit. 10.10.2016]

[10] Liberouter – COMBO80G, < https://www.liberouter.org/combo-80g/>, [cit. 14.12.2016]

SEZNAM POUŽITÝCH ZKRATEK

a – bajt
a(x) – transformační polynom MixColumns
$a^{-1}(x)$ – inverzní transformační polynom InvMixColumns
AES – Advanced Encryption Standard
ASIC – Application Specific Integrated Circuit
b – bit
c – index sloupce
CBC – Cipher Block Chaining
CTR – Counter mode
d – délka zpracovávaných dat [bit]
DMA – Direct Memory Access
DSA – Digital Signature Algorithm
ECB – Electronic Codebook
f – pracovní frekvence čipu FPGA [Hz]
FIPS PUB – Federal Information Processing Standards Publication
FPGA – Field Programmable Gate Array
HDL – Hardware Description Language
ITL – Information Technology Laboratory
LSB – Least Significant Bit
LUT – Look-up Table
m(x) – redukční polynom
MI32 – Memory Interface bus (32bit)
MSB – Most Significant Bit
NIST – National Institude of Standards and Technology
OS – Operační Systém
R – rychlost zpracování dat [bit/s]
r – index řádku
RSA – Rivest, Shamir, Adleman
$s_{r,c}$ – bajt ve stavovém poli
SYN - Syntéza
t – počet taktů nutných pro zpracování dat [-]
TB - Testbench
VHDL – VHSIC Hardware Description Language
VHSIC – Very High Speed Integrated Circuit
w – slovo (32 bitů)