Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra číslicového návrhu

Bakalářská práce

Implementace AES algoritmu pro FPGA

Tomáš Zimmerhakl

Vedoucí práce: Ing. Filip Štěpánek

13. prosince 2015

Poděkování

Děkuji Ing. Filipu Štěpánkovi za odborné rady a čas, který mi při tvorbě práce věnoval.
Děkuji Dr.-Ing. Martinu Novotnému za jeho nápad, bez něhož by tato práce nevznikla.
Děkuji Ing. Pavlu Zimmerhaklovi za důkladnou kontrolu práce.
Děkuji Tomáši Zajíčkovi za stylistické úpravy a jazykové konzultace.
Děkuji svojí rodině za podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 13. prosince 2015 .....

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Zimmerhakl, Tomáš.: *Implementace AES algoritmu pro FPGA*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

Tato bakalářská práce se zabývá návrhem AES algoritmu pro programovatelné obvody (FPGA). Nejprve je předveden základní návrh, z něhož je následně odvozeno 5 spolehlivostních variant. Ty zajišťují odolnost algoritmu vůči poruchám. Odolnost návrhu je dosažena použitím časové, prostorové a informační redundance. Verze AES algoritmu jsou implementovány v jazyku VHDL. Realizované varianty byly zhodnoceny s ohledem na obsazenou plochu na čipu a dobu šifrování. Výsledkem práce je sada realizovaných a otestovaných variant AES algoritmu, která poslouží k výzkumu v oblasti spolehlivosti a útoků postranními kanály na šifru AES.

Klíčová slova AES, FPGA, VHDL, odolnost vůči poruchám, spolehlivost, simulace, ModelSim, Quartus II

Abstract

This thesis describes the design of the AES algorithm for the field-programmable gate array (FPGA). First the basic design is proposed and then five fault-tolerant variants are derived from the basic design. These variants add fault-resistance to the design. The fault-tolerance of the design is achieved by using the time, area and the information redundancy. All the AES variants are implemented using the VHDL language. Then the comparison was created that compares the variants in the scope of area consumption and time of encryption terms. The output of the thesis is a set of implemented and tested variants of the AES algorithm that can be used for the research of reliability and side channel attacks.

Keywords AES, FPGA, VHDL, fault tolerance, reliability, simulation, ModelSim, Quartus II

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V této práci se zabývám implementací různých variant šifrovacích AES algoritmů pro FPGA. Algoritmus AES je dnes hojně využíván v průmyslových standardech a komerčních systémech. Vzhledem k tomu, že FPGA umožňují velmi rychlé datové přenosy, nachází využití především v oblasti síťové komunikace. AES je součástí internetového bezpečnostního standardu IPsec, SSH nebo šifrovacího standardu WiFi IEEE 802.11. Šifrování algoritmem AES se mimo jiné používá v programovatelných hradlových polích od společnosti Altera, kde chrání uživatelské aplikace a jejich data před neoprávněným přístupem.

Při návrhu řešení v rámci této práce je kladen důraz na spolehlivý provoz variant, implementuji je tedy s použitím prostředků, které zajistí možnou detekci/eliminaci poruch vzniklých například nepříznivým vlivem prostředí. Užití podobných řešení se může vyskytovat v systémech, které vyžadují tzv. princip CIA (data confidentiality, integrity, availability) – tedy zdůrazňují diskrétnost, integritu a dostupnost zpracovávaných dat. Zároveň se však v těchto systémech musí předejít nežádoucímu selhání systému z důvodu nepříznivého operačního prostředí. Příklady takových systémů může nabídnout například letecký či automobilový průmysl, můžeme se s nimi setkat v kolejové dopravě nebo v jiných systémech řízení dopravy.

Pro potřeby výzkumu spolehlivostních a vůči útokům odolných systémů je potřeba vytvořit různé varianty šifrovacích algoritmů (AES) pro FPGA, které by byly implementovány za použití spolehlivostních metod (tedy užitím redundance). Výsledné algoritmy navrhuji pro vývojovou/měřící desku Evariste II, která se v rámci výzkumu používá k měření spotřeby FPGA za účelem tzv. útoku postranními kanály.

Cílem této práce je připravit podklady k měření energetické spotřeby šifrovacího algoritmu pro FPGA, aby bylo možné odměřit spotřebu různých variant toho samého algoritmu s důrazem na různé parametry implementace. V rámci textu uvádím postup návrhu jak základních tak spolehlivostních variant algoritmu, jejich realizace a výsledné požadavky na prostředky FPGA (plocha

a hodinová frekvence). Ty mohou být důležité v rámci možného budoucího měření, které ovšem již není součástí této práce.

Práce je rozdělena do čtyř kapitol – analýza, návrh, realizace a testování. V první kapitole seznamuji čtenáře se vznikem standardu AES a jeho historií. Následuje detailní rozbor algoritmu a popis jeho fungování. V závěru první kapitoly nabízím několik příkladů, kde se lze setkat s aplikací AES algoritmu v obvodech FPGA v praxi.

V návrhové části práce předkládám návrh dvou základních a pěti spolehlivostních variant. U každé ze spolehlivostních variant jsou uvedeny její základní charakteristiky a za jakým účelem je navržena. Pro každou z variant jsem vytvořil schémata znázorňující datové cesty a řadiče, z nichž jsem při následné realizaci vycházel. Na návrhovou část navazuji kapitolou realizace, která mimo jiné ukazuje, jakým způsobem se navržená schémata zapisují v jazyku VHDL. Jak při návrhu, tak při realizaci jsem použil rozhraní dodané vedoucím práce, aby bylo možné šifru v budoucnu podrobit útoku postranními kanály. Neopomíjím ani některé problémy, na které jsem při implementaci narazil a jejich následné řešení. V této kapitole se též nachází porovnání všech implementovaných variant s ohledem na obsazenou plochu čipu a čas potřebný k zašifrování dat.

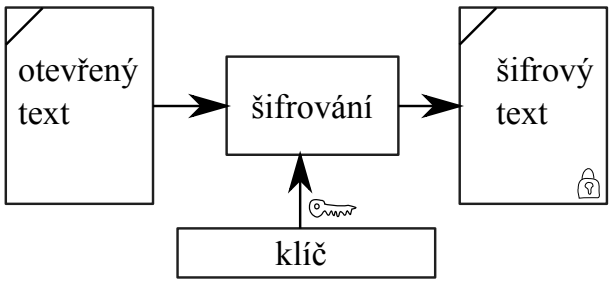
V závěrečné kapitole testování dokládám, že realizované varianty jsou řádně otestovány a fungují v souladu se specifikací. V první části kapitoly vysvětluji, jakým způsobem jsem testy prováděl a ve druhé části popisuji, co přesně jednotlivé testy kontrolují a jak dopadly.

Kapitola 1

Analýza

1.1 Úvod do šifrování

Poptávka po šifrování je na Zemi od nepaměti. Už Julius Caesar si v 1. století před Kristem byl dobře vědom toho, jaké následky by mělo, kdyby jeho zprávy padly do rukou nepřátel. Historie šifer je plná okamžiků, které ovlivnily světové dějiny. Války, převraty, komunikace v nejvyšších patrech politiky – tam všude se uplatnilo a dodnes uplatňuje šifrování. Kromě zvýšené bezpečnosti vedl boj mezi luštiteli a tvůrci šifer také k mnoha vědeckým objevům. Vynaložené úsilí se stalo přínosem pro mnoho oborů lidské činnosti a korunováno bylo vznikem programovatelného počítače [1]. A právě existence počítačů uspíšila v 70. letech 20. století poptávku po nových dokonalých šifrovacích algoritmech. Americká vláda potřebovala zajistit bezpečnost při přenosu dat, a tak byla v roce 1973 vyhlášena soutěž na vytvoření šifrovacího standardu.



Obrázek 1.1: Obecný princip šifrování

Ještě než text přejde k jednotlivým šifrám, je načase vymezit pár nejzákladnějších pojmů, se kterými se čtenář v této práci setká. Obecný princip šifrování je znázorněn na obrázku 1.1. Existují šifry symetrické a asymetrické. Prvně jmenované používají stejný klíč při šifrování i při dešifrování, a proto je

nutné klíč držet v tajnosti. Asymetrická šifra použije k šifrování veřejný klíč, který může vlastnit každý a není tajný. K dešifrování je použit soukromý klíč, který zná pouze majitel klíče a pouze on může zprávu rozluštit [2], [3]. Další možné dělení je na blokové a proudové šifry. Zatímco bloková zpracovává text po blocích, proudová šifra šifruje text po znacích.

Roku 1976 byl americkými úřady přijat první veřejný kryptografický algoritmus DES (Data Encryption Standard). Již v době svého vzniku se spekulovalo o jeho bezpečnosti, nicméně na prolomení hrubou silou si svět počkal ještě 20 let. O prolomení algoritmu DES hrubou silou se mimo jiných úspěšně pokoušel tým okolo stroje COPACOBANA. Ten se skládá ze 120 FPGA a je optimalizován pro lámání šifer, průměrně dokáže najít klíč k šifře za 6,4 dne [4]. Postupem doby bylo nezbytné přijít s bezpečným algoritmem, který umožní přenos citlivých dat (bankovní transakce, soukromá korespondence, ...) po síti. Tím se stal na konci 90. let Advanced Encryption Standard (AES), který je obecně považován za jednu z nejbezpečnějších blokových šifer.

Jádrem této práce je realizace algoritmu AES na FPGA, proto bude v následujících sekcích věnována pozornost už jen tomuto algoritmu.

1.2 Advanced Encryption Standard – AES

Tato kapitola stručně představuje historii AES a okolnosti vzniku nového šifrovacího standardu. V následující části je detailně rozebrána vnitřní struktura algoritmu a je zde patrné, jak konkrétně algoritmus AES šifrování provádí.

Advanced Encryption Standard (AES) je šifrovací standard publikovaný americkým National Institute of Standards and Technology (Národní úřad pro standardizaci a technologie) [5]. Zatímco standard AES vytvořil americký úřad NIST, konkrétní algoritmus, který je zodpovědný za šifrování se nazývá Rijndael. Vytvořili ho belgičtí autoři Joan Daemen a Vincent Rijmen. Mohlo by se zdát, že pojmy AES a Rijndael jsou vlastně totéž, avšak striktně vzato, není tomu tak. Zjednodušeně řečeno AES je standard, zatímco Rijndael je algoritmus, tedy konkrétní implementace daného standardu. Navíc AES striktně omezuje délku bloku na 128 bitů (tedy 16 bytů), což u Rijndaelu neplatí. Algoritmus Rijndael umí šifrovat/dešifrovat i bloky o velikostech 24 a 32 bytů. Pokud se ale dnes řekne AES, automaticky je tím myšlen algoritmus Rijndael.

1.2.1 Stručná historie AES

Po prolomení algoritmu DES (Data Encryption Standard) hrubou silou v roce 1997 [6] začal americký úřad NIST připravovat nový standard, který bude splňovat ty nejpřísnější podmínky pro šifrování dokumentů amerických bezpečnostních složek.

Algoritmus DES byl sice již v roce 1997 narychlo nahrazený svou modernizovanou verzí 3DES (TripleDES). Ten provádí šifrování třikrát za sebou pomocí standardního DES. Přestože se díky tomu prolomení hrubou silou

stalo opět nemožné, šlo pouze o narychlo použité řešení, které trpělo několika problémy. Zaprvé 3DES není příliš efektivní v softwarové implementaci, přesněji řečeno byl přesně třikrát pomalejší než DES. Další nevýhodu v některých aplikacích způsobovala malá délka bloku – 8 bytů. Všechny tyto okolnosti vedly v lednu roku 1997 NIST k oznámení veřejné soutěže na realizaci nového a bezpečnějšího standardu – AES.

Výběrové řízení bylo vyhlášeno 12.9.1997 a do srpna roku 1998 se přihlásilo 15 kandidátů. Mezi nezbytné požadavky na novou šifru patřily podle [7] následující:

- bloková šifra s délkou bloku 16 bytů
- podpora 3 délek klíčů: 16, 24 a 32 bytů
- možnost efektivní realizace jak v softwaru tak v hardwaru

Na Druhé konferenci k AES konané v Římě v březnu roku 1999 bylo představeno několik studií, které podrobně zkoumaly a testovaly kandidátní algoritmy [7]. Už tam bylo zřejmé, že pro nesplnění bezpečnostních požadavků nebude do nejužšího výběru pokračovat hned 5 algoritmů. Mimo jiné byly představeny studie, které zkoumaly, jak vhodné jsou jednotlivé algoritmy k nasazení do čipových karet. Ukázalo se, že mnoho z nich je pro svoji složitost nemožné do karet nasadit a jako nejvhodnější pro tuto platformu se ukázal Rijndael.

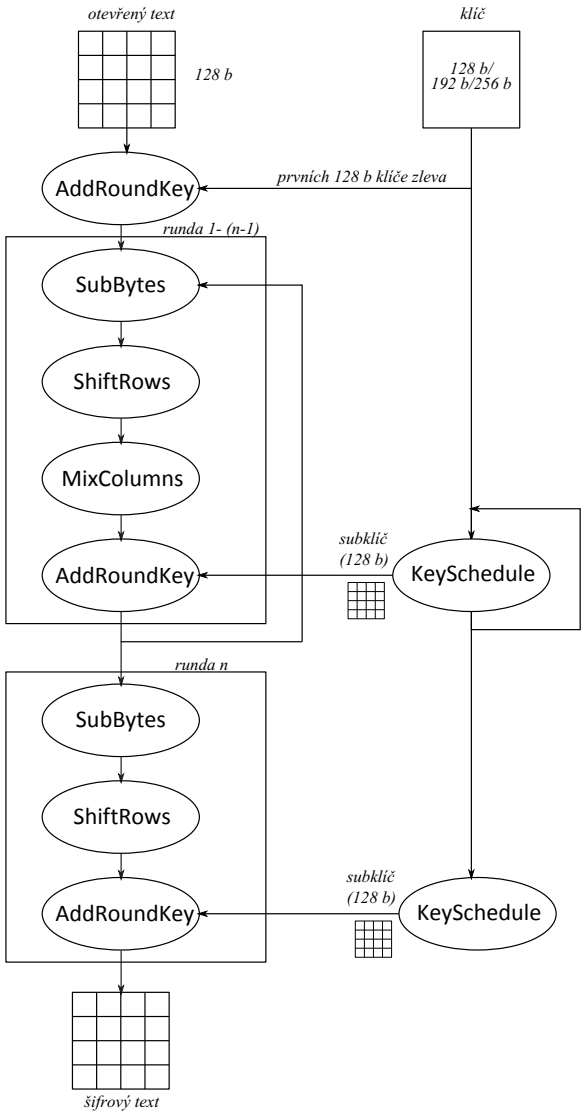
Po dalších 5 měsících vybral Národní úřad pro standardizaci 5 finalistů. Do nejužšího výběru se dostaly algoritmy Mars, RC6, Rijndael, Serpent a Twofish. V porovnání s ostatními finalisty byly bezpečnější, rychlé v hardwaru nebo levnější. V dubnu roku 2000 se v New Yorku konala Třetí konference k AES, kde bylo opět představeno několik výzkumů podrobně rozebírajících kandidáty. Jedna přednáška se také věnovala hardwarovým implementacím na FPGA a ASIC (zákaznické obvody určené pro jednu specifickou aplikaci). V této oblasti za svými konkurenty mírně zaostávaly RC6 a Mars. V porovnání s nimi byla implementace v hardwaru složitější a tudíž dražší. Na závěr konference byli účastníci dotazováni, jakého mají favorita. Výsledky byly nejpříznivější pro Rijndael.

Uzavření soutěže přišlo 2. 8. 2000, kdy byl oznámen vítěz – Rijndael. Podle závěrečné zprávy publikované NIST [8] je Rijndael vhodný jak pro softwarovou, tak pro hardwarovou implementaci. Má velmi nízké požadavky na paměť, což ho činí vhodným k použití v prostředích s omezenými výpočetními prostředky. Konečně vnitřní struktura algoritmu je potenciálně velmi vhodná k paralelnímu zpracování na úrovni instrukcí.

1.2.2 Vnitřní struktura algoritmu AES

Následující oddíl vysvětluje, jakým způsobem funguje obecná šifra AES. Implementační detaily nebo specifika hardwarové realizace jednotlivých částí

popíše blíže kapitola 3.2. Kapitola se zabývá pouze procesem šifrování, opačný proces není náplní této práce. Dešifrování je srozumitelně popsáno v [9].



Obrázek 1.2: Blokový diagram AES šifry

Standard AES je bloková šifra s délkou bloku 16 bytů. Šifrovací algoritmus se skládá z takzvaných rund. Runda je posloupnost několika operací, které určitým způsobem transformují otevřený text. Tyto operace uvnitř rundy se nazývají *SubBytes* (substituce bytů), *ShiftRows* (rotace řádků), *MixColumns* (substituce sloupců) a *AddRoundKey* (přičtení rundovního klíče). O dodávku správného klíče v rámci jedné rundy se stará blok *KeySchedule*. Všechny výše popsané operace znázorňuje obrázek 1.2.

Operace uvnitř rundy se také označují pojmem moduly. V [9] je označují jako vrstvy, v [7] používají termín transformace nebo kroky a v [8] používají termín operace. Jak je vidět, terminologie je poměrně roztříštěná, tento text se drží pojmů modul nebo operace. V závorkách za anglickými názvy operací je uveden možný český překlad, který byl převzat z [10], ale aby se předešlo případným nedorozuměním, bude se text nadále držet originálních anglických pojmenování. Detailním popisem všech modulů se zabývá kapitola 1.2.3.

Je-li k zašifrování textu o délce 16 B použit klíč o délce 16 B, skládá se proces šifrování z celkem 10 rund. To ale neplatí vždy. Počet rund, ze kterých se šifrovací proces skládá, je závislý na délce klíče, který je použit. V případě, že je použit klíč s délkou 24 B, provádí se 12 rund. V případě nejdelšího klíče (32 B) se provádí 14 rund [11]. Delší klíč účinně zabraňuje útokům, které byly aplikované na předchůdce DES (Data Encryption Standard) a vylučuje útok hrubou silou v rozumném čase. Pokud by byl k útoku použit počítač, který dokáže testovat 10^{12} klíčů za sekundu, pak by výpočet trval $1,84 \cdot 10^{57}$ let.

Bez ohledu na délku klíče je práce s klíčem v průběhu šifrování v principu stále stejná. Pokud označíme n jako počet rund, pak bude klíč v průběhu jednoho šifrování použit přesně $(n + 1)krát$. Ovšem pokaždé, když se v algoritmu pracuje s klíčem, je použita jiná hodnota. Odvozování správné hodnoty klíče v každé rundě má na starosti modul *KeySchedule*, který v každé rundě poskytuje subklíč o délce 16 bytů (stejně dlouhý jako blok otevřeného textu). Proto jediný rozdíl při implementaci šifrovacích algoritmů AES-128, AES-192 a AES-256[1] spočívá v počtu rund a realizaci modulu *KeySchedule*, který musí produkovat bez ohledu na délku klíče vždy stejný šestnáctibytový subklíč.

Při šifrování se s otevřeným textem a s klíčem pracuje ve formě matic. Jakým způsobem se text do matice uspořádá je blíže vysvětleno na začátku sekce 1.2.3. Nechť n je počet rund. Pak lze ve stručnosti shrnout šifrovací algoritmus do následujících bodů:

1. Ještě před první rundou se mezi maticí otevřeného textu a maticí klíče provede operace *xor*[2]. Vzniká nová matice o stejné velikosti.

2. Poté vstupuje otevřený text do rundy, kde je postupně transformován operacemi v tomto pořadí:

   a) *SubBytes*.

   b) *ShiftRows*.

   c) *MixColumns*.

   d) *AddRoundKey*. Tato operace je shodná jako v bodě 1.

[1] Číslo za názvem algoritmu 128, 192 nebo 256 označuje délku klíče v bitech.

[2] Xor je binární operace s bity. Výstupem operace a xor b je 1, pokud je právě jeden z bitů roven 1 a druhý 0. Jinak je výstupem 0.

3. Celý postup v bodě 2 se opakuje $(n-1)krát$.

4. V poslední n-té rundě je operace MixColumns vynechána.

1.2.3 Moduly algoritmu AES

Hlavními zdroji, ze kterých bylo při psaní této kapitoly čerpáno, jsou [7] a [9]. Před začátkem šifrovacího procesu se otevřený text v binární podobě rozdělí do bloků po 16 bytech, každý blok se ještě rozdělí po bytech, které jsou indexovány zleva doprava po řadě od A_0 až po A_{15}. Byty se uspořádají do matice o velikosti 4×4 a to po sloupcích. Uspořádání popisuje tabulka 1.1.

Tabulka 1.1: Rozdělení otevřeného textu do matice

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

Tedy v prvním sloupci zleva jsou shora dolů byty A_0 až A_3, v posledním pravém sloupci jsou byty A_{12} až A_{15}. Podle stejného principu je do matice 4×4 uspořádán i klíč. Jestliže je otevřený text uspořádán do mřížky, je šifrování převedeno na úpravy matice, jejíž prvky jsou 8bitové vektory.

Pro všechny níže popsané moduly s výjimkou KeySchedule je společné, že do nich vstupuje 16 bytů dat a zase z nich 16 bytů dat vystupuje. Všechny včetně modulu KeySchedule mohou být v hardwaru implementovány jako kombinační logika.

1.2.3.1 SubBytes

Na tento modul je možno pohlížet jako na dalších 16 nezávislých paralelních modulů, z nichž každý přijímá jeden byte, ten nahradí podle určitého pravidla a na výstupu zase jeden byte vrací. Tento paralelní modul se nazývá SBox.

SBox je jediný nelineární modul šifry AES. Podle [9] je možné tuto nelinearitu vyjádřit jako $SBox(A) + SBox(B) \neq SBox(A + B)$, kde A, B jsou dva osmibitové datové vstupy. SBox funguje jako bijektivní zobrazení, které každému z 256 možných vstupů přiřazuje přesně jeden z 256 možných výstupů a přitom žádný prvek není namapován sám na sebe. Díky tomu je možné provést při dešifrování opačnou operaci a zašifrované byty obnovit do původních hodnot. SBox je obvykle realizován jako asociativní tabulka. Vstupní hodnota určuje pozici v tabulce, na které se hledá odpovídající výstupní hodnota. Například $SBox(A9_{hex}) = D3_{hex}$, protože v řádku A a ve sloupci 9 je v tabulce hodnota $D3$.

Tabulka 1.2: Tabulka SBox

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

Tabulka 1.2, podle které funguje modul SBox, není náhodná, ale je vypočítána na základě matematických pravidel. Hodnoty v ní uvedené jsou taktéž v hexadecimální podobě. Nechť A je vstupní byte a B je výstupní byte, pak platí:

$$B = \mathfrak{T}(A^{-1}),$$

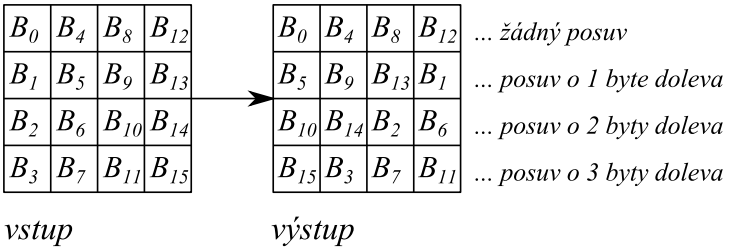
kde A^{-1} je inverze prvku A v $GF(2^8)$ – Galoisově tělese, které má 256 prvků. Pro každý vstupní prvek existuje v $GF(2^8)$ inverzní prvek s výjimkou nuly. Přestože inverzní prvek k nule neexistuje, pro SBox je inverze k nule uměle dodefinována jako nula. Druhou částí substituce je operace $\mathfrak{T}$. Ta je definována následujícím vzorcem:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \; mod\ 2.$$

V uvedeném vzorci je $B = (b_7, \ldots, b_0)$ je bitová reprezentace výstupního bytu, $B' = (b'_7, \ldots, b'_0)$ je bitová reprezentace inverze ke vstupnímu bytu, tedy $B' = A^{-1}$.

1.2.3.2 ShiftRows

Operace ShiftRows s následující operací MixColumns společně zajišťují difúzi[3] otevřeného textu. Jejich úkolem je pokud možno co nejdůkladněji zpřeházet části textu. Modul ShiftRows je jednoduchý na pochopení i implementaci. Jedná se o pouhé prohození bytů podle pravidla, které zobrazuje obrázek 1.3. Od prvního až po čtvrtý řádek matice jsou byty cyklicky posouvány vlevo po řadě o 0, 1, 2 a 3 byty. Přestože jsou výstupní byty z operace *ShiftRows* na obrázku 1.3 značeny písmeny B_n, jsou ty samé byty při vstupu do následující operace značeny písmeny C_n. Toto značení je zavedeno kvůli zřetelnému odlišení jednotlivých operací.



Obrázek 1.3: Operace ShiftRows

1.2.3.3 MixColumns

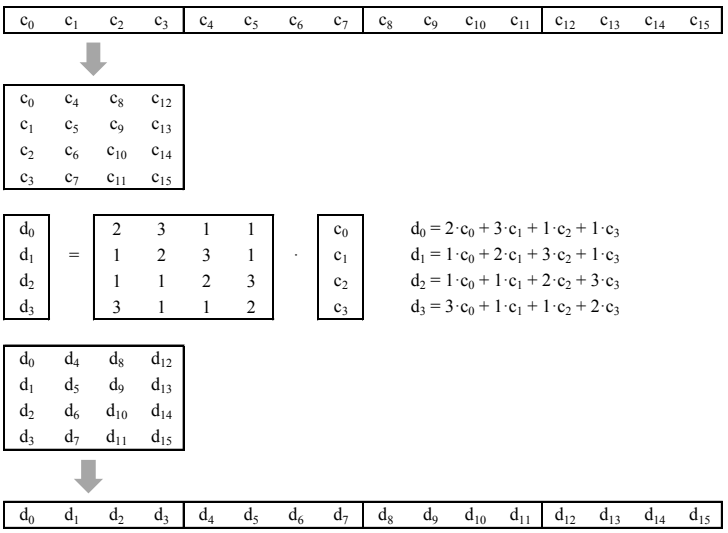
Účelem modulu MixColumns je především zajistit difúzi dat. Tento pojem vysvětluje skutečnost, že každý jeden vstupní byte ovlivní čtyři byty výstupní. Data jsou díky této operaci postupně rozptylována po celé šíři šifrového textu a již po třech rundách ovlivní jeden vstupní byte hodnotu všech 16 bytů ve výstupní matici [9]. K tomuto procesu přispívá také operace ShiftRows.

MixColumns pohlíží na matici dat po sloupcích a pracuje s 4bytovými vektory, které odpovídají sloupcům matice. Každý vektor (o délce 4 byty) je násoben tzv. násobící maticí podle následujícího vzorce:

$$\begin{pmatrix} d_k \\ d_{k+1} \\ d_{k+2} \\ d_{k+3} \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} c_k \\ c_{k+1} \\ c_{k+2} \\ c_{k+3} \end{pmatrix},$$

kde k iteruje od 0 do 3. Položky s názvy začínající na c jsou byty, které jsou v matici po proběhnutí operace ShiftRows. Byty označené písmeny $d_0 - d_{15}$

[3] Difúze je rozptyl jednotlivých částí šifrového textu po celé šíři textu.



Obrázek 1.4: Operace MixColumns

odpovídají výstupním bytům po operaci MixColumns. Například Byte d_5 ve výstupní matici se vypočítá následujícím způsobem:

$$d_5 = 01 \cdot c_4 + 02 \cdot c_5 + 03 \cdot c_6 + 01 \cdot c_7.$$

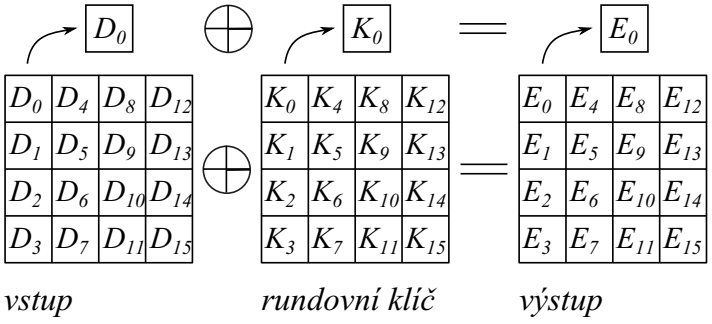
V uvedeném příkladu záměrně uvádím násobící koeficienty ve tvaru 01, 02, 03, protože jak tyto koeficienty, tak hodnoty bytů c_x a d_x pro $x \in \{0, \ldots, 15\}$ jsou v hexadecimálním tvaru a jsou zde považovány za prvky tělesa $GF(2^8)$. Například prvek 03 v násobící matici odpovídá bitovému vektoru (00000011). Veškeré matematické operace, které se zde vyskytují, se odehrávají uvnitř tělesa $GF(2^8)$. Sčítání odpovídá běžnému sčítání modulo 2. Násobení vektorem 01 nepřináší nic nového, $a \cdot 01 = a$. Při násobení bytu vektorem 02 a 03 už může dojít k zvětšení bitové délky výsledného vektoru. Pokud k němu dojde, je třeba zajistit, aby výsledek byl osmibitový. V takovém případě se provádí modulární redukce polynomem $P(x) = x^8 + x^4 + x^3 + x + 1$ [9], kterému odpovídá devítibitový vektor (100011011). Jestliže vyjde výsledek násobení delší než osmibitový, redukce se provádí následujícím způsobem:

1. Pod vektor, který je třeba zredukovat, se zapíše co nejvíce vlevo $P(x)$.
2. Mezi odpovídajícími bity se provede operace xor.
3. Pokud není výsledek osmibitový, postup se opakuje od kroku 1.

Obrázek 1.4 znázorňuje, jakým způsobem jsou zdrojová data transformována na výstupní data. Především násobení sloupce maticí je zde uvedeno srozumitelněji než v případě matematického zápisu.

1.2.3.4 AddRoundKey

Do modulu AddRoundKey vstupuje výstup z operace MixColumns a rundovní klíč. Oba vstupy jsou uspořádány do matice a mají velikost 16 B. Zde se odpovídající byty/bity (v tomto případě na tom nezáleží) sečtou modulo 2, což odpovídá bitové operaci xor. Postup je zobrazen na obrázku 1.5. O dodávku správného rundovního klíče se stará modul KeySchedule.



Obrázek 1.5: Operace AddRoundKey

1.2.3.5 KeySchedule

Anglický název tohoto modulu lze překládat jako „expanze klíče“, text se bude pro udržení kompatibility s cizojazyčnými zdroji [7], [8], [9] nebo [12] držet anglického termínu. Stručně řečeno, tento modul má za úkol ze vstupního klíče vyrobit n rundovních klíčů. n je závislé na délce vstupního klíče, závislost počtu rund a rundovních klíčů na délce vstupního klíče udává tabulka 1.3.

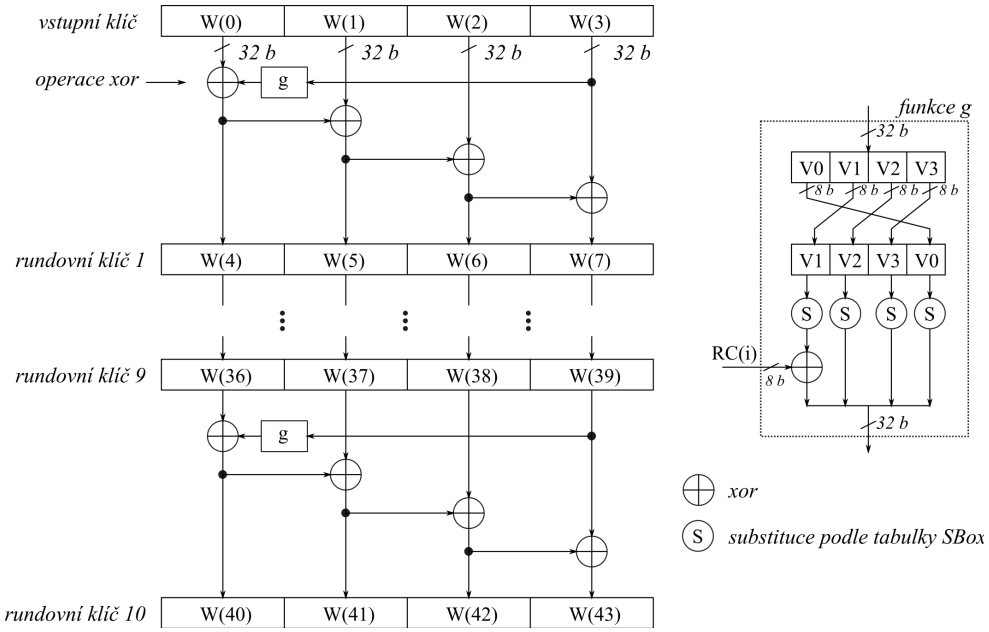
Tabulka 1.3: Závislost počtu rund a rundovních klíčů na délce klíče

Délka klíče	*Počet rund = počet rundovních klíčů*
128 b	10
192 b	12
256 b	14

V některé literatuře (například v [9]) se uvádí, že počet rundovních klíčů (v [9] je použit termín subklíč) je vždy o 1 vyšší než počet rund, což je v rozporu s udávanými hodnotami v tabulce 1.3. Odlišnost je způsobena tím, že v této sekci popisuji, jak funguje modul KeySchedule, který nějak odvozuje ze vstupního klíče rundovní klíč. Operace AddRoundKey je sice prováděna o jedna vícekrát než je počet rund, ovšem poprvé vstupuje do AddRoundKey přímo samotný klíč (nebo jeho část v případě délky klíče 24 nebo 32 bytů). Není tedy

třeba, aby byl tento „subklíč“ odvozován/generován modulem KeySchedule. Proto v tabulce 1.3 uvádím, že KeySchedule generuje stejný počet klíčů jako je počet rund.

Až dosud byl popis všech modulů uváděn univerzálně, bez ohledu na délku klíče. Realizace operací *SubBytes*, *ShiftRows*, *MixColumns* a *AddRoundKey* se s různou délkou klíče totiž neliší. To neplatí pro modul KeySchedule, který má pro každou délku vstupního klíče jinou implementaci. Vzhledem k faktu, že účelem této práce je navrhnout a zrealizovat AES s délkou klíče 16 bytů, uvádí text informace pouze o tom, jak funguje modul KeySchedule pro délku klíče 16 B. Ostatní verze si mohou zájemci dohledat v [7], [9].



Obrázek 1.6: Modul KeySchedule

Vnitřní struktura modulu KeySchedule je zobrazena na obrázku 1.6, který jsem vytvořil podle předlohy v [9]. Vstupní klíč je rozdělen na 4 stejně dlouhé bloky po 4 bytech číslované od nuly zleva. Jak je vidět na obrázku 1.6, následující rundovní klíč složený ze čtyř bloků $W(4i), W(4i+1), W(4i+2), W(4i+3)$ se z předchozího odvodí následovně:

$$W(4i) = W(4(i-1)) + g(W(4i-1))$$

pro první 4 byty zleva a

$$W(4i+j) = W(4i+j-1)) + W(4(i-1)+j)$$

pro zbývající 3krát 4 byty klíče, kde $i \in \{1, \ldots, 10\}$ a $j \in \{1, 2, 3\}$.

Z obrázku i výše uvedených vztahů vyplývá, že do procesu odvození nového klíče vstupuje i funkce *g*. Ta přijímá data o délce 4 byty a jejím výstupem je opět 4bytový vektor dat. Funkce g pracuje s daty po bytech. Nejprve provede cyklický posuv bytů vlevo, poté nahradí každý byte podle tabulky SBox (viz tabulka 1.2) a nakonec k bytu, který je nejvíce vlevo, připojí operací xor *rundovní koeficient* $RC(i)$, kde i odpovídá číslu rundy. *Rundovní koeficient* je hodnota o délce 1 B, která je prvkem $GF(2^8)$, jeho hodnota se odvodí následovně:

$$RC(i) = x^{i-1} \ mod \ 256.$$

Pro lepší porozumění přikládám tabulku 1.4, která uvádí hodnoty některých rundovních koeficientů. V případě, že by měla délka rundovního klíče vyjít delší než 8 bitů, provádí se redukce modulo polynom, jejíž postup je uveden na konci sekce 1.2.3.3.

Tabulka 1.4: Hodnoty některých rundovních koeficientů

Koeficient	*Výpočet*	*Binární hodnota*	*Hexadecimální hodnota*
$RC(1)$	x^0	0000 0001	01
$RC(2)$	x^1	0000 0010	02
		...	
$RC(10)$	x^9	0011 0110	36

1.3 Aplikace AES v FPGA

Tato kapitola uvádí několik příkladů, kde se lze setkat s implementací AES v obvodech FPGA v praxi. Algoritmus AES jako takový je dnes hojně využíván v průmyslových standardech a komerčních systémech. Vzhledem k tomu, že FPGA umožňují velmi rychlé datové přenosy (v řádech jednotek až desítek Gb/s [10]), nachází využití především v oblasti síťové komunikace. Je součástí internetového bezpečnostního standardu IPsec, SSH nebo šifrovacího standardu WiFi IEEE 802.11 [9].

Zajímavým využitím šifrování AES je zabezpečení, se kterým přišla společnost Altera, významný výrobce FPGA. Na rozdíl od „tradičního“ použití AES, kdy jsou šifrována externí data, která modulem FPGA jen protékají, v tomto případě je šifrování použito k zabezpečení samotné aplikace, která je na FPGA vytvořena. Respektive AES zde zajišťuje, aby nedošlo k neoprávněnému kopírování, reverznímu inženýrství či manipulaci s daty. Jak uvádí web Pandatron ve svém článku [13], buňky SRAM[4] uvnitř FPGA musí být

[4]SRAM (Static Random Access Memory) je typ počítačové paměti, která k uchování dat nepotřebuje jejich periodickou obnovu, po vypnutí přísunu elektrické energie je ale veškerý obsah ztracen.

naplněny konfigurací obvodu pokaždé, když dojde k jeho zapnutí. Po startu dochází k přenosu dat, která je možné zachytit a neoprávněně použít. Aby se tomu zabránilo, mají obvody od firmy Altera vyhrazen prostor přímo na svém čipu pro dešifrovací blok, který dešifruje přenášená konfigurační data zabezpečená AES šifrou. Tyto FPGA zařízení používají klíč s délkou 32 bytů. Použití šifrování není povinné a je možné šifrovací blok odpojit.

Velice slibně se vyvíjí spolupráce brněnské společnosti Invea-Tech a sdružení CESNET. Pracují na síťové FPGA kartě, pomocí níž bude možné zpracovávat provoz v síti o rychlostech 400 Gb/s. Úkolem síťové karty bude kromě filtrování provozu také zajištění bezpečnosti [14], což mohou (ale nemusí) mít na starost právě FPGA čipy uvnitř karty. Jak uvádí [15], finální produkt by měl být distribuován s vývojářským balíčkem. Zákazník dostane výkonný síťový hardware a platformu a do toho si bude moci naprogramovat vlastní aplikace. Na trh se chystá v roce 2018, obyčejný smrtelník si ji ale nekoupí. Cena by se měla pohybovat mezi 500 000 – 1 000 000 Kč a zákazníci budou poskytovatelé cloudových služeb, telekomunikační operátoři a datová centra [15].

Síťové FPGA karty ale nejsou nic nového, už v roce 2013 byla k dispozici na trhu karta PCIe-287N od společnosti Nallatech s dvěma FPGA jádry a podporou až 10 Gb ethernetu [16]. Nyní je možné si zakoupit karty až do rychlosti 100 Gb/s [17]. Z mailové konverzace s Petrem Kastovským, technickým ředitelem společnosti Netcope (výrobce síťových FPGA karet), plyne, že aktuálně prodávané FPGA karty jsou vhodné k implementaci AES. Zákazníci si je kupují často právě s cílem implementovat si vlastní šifrování (AES), díky čemuž mají jistotu, že šifrování funguje bezpečně podle jejich představ a není ohroženo potenciálními bezpečnostními mezerami ze strany dodavatele šifrování.

Další oblastí pro využití AES v FPGA je šifrování videostreamu. Zajistit bezpečnost je nutné především v podnikové sféře při videokonferencích, kde by případné narušení bezpečnosti bylo drahé. AES je v této oblasti využíván pro svoji vysokou bezpečnost [18]. Použití FPGA zaručuje rychlost a zpracování v reálném čase, které je při šifrování videostreamu nezbytné. Jedním z konkrétních příkladů je práce [18], kde tým britsko-tuniských badatelů představuje návrh systému, který šifruje v reálném čase výstup z kamery v rozlišení 720 krát 480 px při 60 fps a posílá ho na DVI[5] konektor monitoru.

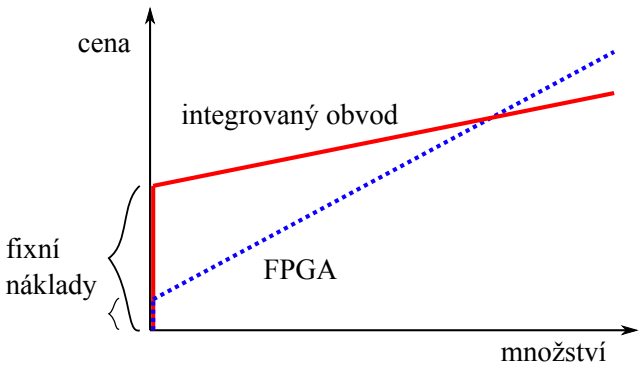
1.4 Field Programmable Gate Array – FPGA

Účelem této sekce je seznámit čtenáře ve stručnosti s programovatelnými hradlovými poli (FPGA), o nichž se v této práci často zmiňuji.

[5]DVI (Digital Visual Interface) je typ konektoru pro přenos obrazových dat mezi počítačem a obrazovkou.

Programovatelná hradlová pole jsou zařízení, která nejsou při výrobě programována pro nějaký účel. Konkrétní funkce, kterou mají vykonávat se do nich naprogramuje až po vyrobení [10]. Uplatňuje se zde chytře vymyšlená dělba zodpovědnosti. Výrobce FPGA garantuje, že je zařízení vyrobeno bez závad, a že bude spolehlivé. On je ten, kdo musí zkoumat, vyvíjet a testovat hardware. Zákazník koupí hotový výrobek, do kterého si nechá navrhnout, nebo si sám vytvoří konkrétní funkci. Jestli bude fungovat obvod, tak jak má, už je na zkušenosti a kvalitách FPGA návrháře. Zákazník i návrhář mají ale obrovskou výhodu v tom, že pokud se jejich návrh pokazí, snadno ho opraví a během několika minut ho mohou opravený nahrát do stejného hardware [19]. Později je navíc možné vhodnou úpravou software funkčnost FPGA rozšířit nebo úplně změnit.

Při vývoji zcela nového obvodu je FPGA výhodné z finančního hlediska. V případě výroby jednoúčelového integrovaného obvodu jsou počáteční fixní náklady velice vysoké (v řádu statisíců Kč) [19]. Porovnání nákladů na výrobu integrovaného obvodu a použití FPGA ilustruje obrázek 1.7. Graf je přepracován podle vzoru v [19]. Tečkovaná čára zobrazuje, že fixní náklady na FPGA jsou oproti návrhu integrovaného obvodu zanedbatelné, na druhou stranu, jeden kus FPGA je značně dražší. V principu tedy platí, že čím menší bude množství vyrobených kusů, tím více se vyplatí investovat do FPGA a obráceně.



Obrázek 1.7: Srovnání nákladů na výrobu FPGA a jednoúčelového integrovaného obvodu

FPGA má dle [19] i několik nevýhod. Z výrobní technologie vyplývá, že výsledný obvod je v porovnání s integrovaným obvodem asi 3krát pomalejší a 10krát větší. Má také větší spotřebu a větší zpoždění na propojovací síti, protože propojovací síť je mnohem větší než prosté propojení.

Kapitola 2

Návrh řešení

Zadáním práce bylo navrhnout a implementovat několik variant šifrování algoritmem AES. Všechny mnou navrhované varianty pracují s délkou klíče 128 b. Návrh jsem rozdělil na dvě fáze. Nejprve jsem navrhoval pouze základní AES algoritmus podle specifikace v kapitole 1.2.2, které jsou vhodné pro seznámení. Později jsem se zabýval návrhem pěti spolehlivostních variant.

V první fázi jsem navrhl dvě „základní“ architektury algoritmu (sekce 2.1). Pojem „základní“ zde označuje, že AES dělá přesně to, co má, ale nemá žádné další nadstavbové funkčnosti. Mezi ně patří různé spolehlivostní mechanismy, které dokážou odhalit případnou poruchu a třeba ji i opravit. Optimalizovaná varianta bývá navrhována zpravidla s ohledem na co nejkratší dobu výpočtu nebo co nejmenší obsazenou plochu na čipu. Poté, co jsem dokončil návrh základních variant, odvodil jsem z nich takzvané spolehlivostní varianty, které jsou odolné proti poruchám. O spolehlivostních verzích pojednává sekce 2.2.

Všechny mnou navržené varianty AES mají následující rozhraní, které jsem obdržel od vedoucího své práce. Rozhraní je definováno v jazyku VHDL.

```
entity aes_top is
    port
    (
        CLK           : in std_logic;
        RST           : in std_logic;
        INPUT_VALID   : in std_logic;
        IN_DATA       : in std_logic_vector(127 downto 0);

        DATA_REQUEST  : out std_logic;
        BUSY          : out std_logic;
        OUTPUT_VALID  : out std_logic;
        OUT_DATA      : out std_logic_vector(127 downto 0)
    );
end aes_top;
```

Podle zadání má nejvyšší entita *aes-top* 4 vstupní a 4 výstupní porty. Jejich popis dodal vedoucí práce a je popsán v tabulce 2.1.

Tabulka 2.1: Význam portů v entitě aes-top

Název portu	*Význam*	*Šířka*
`CLK`	hodinový signál	1 *b*
`RST`	signál reset	1 *b*
`INPUT_VALID`	signál platný po 1 hodinový takt, kdy se odstartuje výpočet, je roven `'1'`, pokud jsou `IN_DATA` platná	1 *b*
`IN_DATA`	data určená k zašifrování	128 *b*
`DATA_REQUEST`	je roven `'1'`, pokud aes-top právě nic nepočítá	1 *b*
`BUSY`	komplementární signál k `DATA_REQUEST`, je roven `'1'`, pokud je aes-top zaneprázdněn výpočtem	1 *b*
`OUTPUT_VALID`	je roven `'1'` pouze po dobu jednoho hodinového taktu, pokud jsou výstupní data `OUT_DATA` platná	1 *b*
`OUT_DATA`	zašifrovaná data na výstupu	128 *b*

Šifrovací klíč dodává samostatná komponenta v rámci AES návrhu. Práce má experimentální charakter a s klíčem se tedy pracuje uvnitř jiné komponenty v mojí práci označené jako `KEYGEN`.

Standardním postupem při návrhu hardwaru je dekompozice celku na dvě menší části – datovou cestu a řadič. Datová část je dále rozdělena na soustavu malých komponent, kde každá plní jednoduchou funkci. Příkladem komponenty může být třeba operace *SubBytes* nebo *ShiftRows*. Tyto komponenty jsou následně spojeny do celku, který je ovládán řídícími signály. Ty generuje řadič, realizovaný jako konečný automat.

2.1 Varianty základní verze AES

Vnitřní struktura základního AES algoritmu může být navržena mnoha způsoby s ohledem na měřenou veličinu, u které chceme dosáhnout nejlepších hodnot. Na základě dodaného rozhraní jsem nejprve uvažoval nad návrhem celkem čtyř následujících řešení.

AES01

Verze, která zpracovává jeden blok dat (16 B) za 10 hodinových taktů. Každá runda je hotová přesně za jeden takt. Datová cesta je pro každou rundu společná, což ušetří prostor na čipu.

AES02
Celý AES je vytvořen jako jeden kombinační obvod. Šifrovaný výstup je k dispozici okamžitě, navíc není třeba řadič. Návrh je ale poměrně nepřehledný. Každá z 10 rund musí mít všechny moduly(SubBytes, ShiftRows, MixColumns, AddRoundKey) vlastní. Některý hardware se tedy desetkrát téměř identicky opakuje, návrh je proto velice neekonomický z hlediska spotřebované plochy na čipu.

AES03
Jde o AES01 časově rozbitý až na úroveň operací. Po každé ze zmíněných operací přijde hodinový takt. Šifrování jednoho bloku dat tedy trvá 40 taktů, což je sice hodně, na druhou stranu kritická cesta bude oproti ostatním verzím velice krátká, proto může taková verze mít nejvyšší frekvenci hodin. Při vhodném návrhu řadiče je možné některé části hardwaru sdílet a ušetřit tak místo.

AES04
Implementace s využitím pipeline[6]. Některé části šifry se zduplikují a díky tomu dochází v jeden čas k šifrování více bloků naráz. Šifrování je tedy tolikrát rychlejší, kolikrát jsou zkopírovány všechny operace. Toto řešení ale stejně jako u AES02 vede k velké spotřebě plochy. Oproti ostatním verzím má také tu nevýhodu, že při resetu musí dojít k vymazání částečně zpracovaných dat z pipeline. K tomu jsou zapotřebí obvody navíc.

Mojí snahou bylo navrhnout architekturu, která minimalizuje čas potřebný k šifrování a také architekturu, která optimalizuje využitou plochu na čipu. Je výhodné, když je vybraná verze snadno upravitelná na verzi odolnou proti poruchám. Z výše uvedených možností jsem tedy přistoupil k bližšímu průzkumu verze AES01 a AES02.

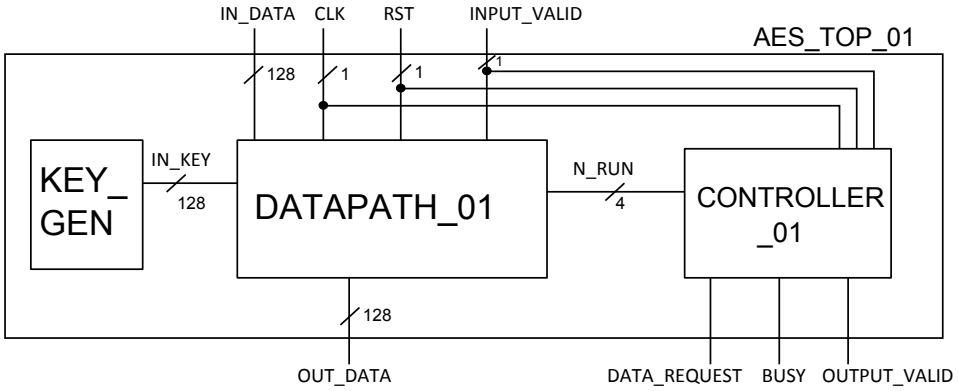
2.1.1 Sekvenční AES na 10 taktů – AES01

Následující seznam uvádí základní charakteristiky návrhu.

- Jeden blok dat je zašifrován za dobu deseti hodinových taktů.
- Za jeden takt je provedena jedna runda.
- Kombinační část datové cesty zpracovává jednu rundu, číslo právě prováděné rundy dodává signálem řadič.

[6]Pipeline nebo také zřetězení vystihuje stav, kdy je více za sebou následujících operací rozděleno na menší suboperace, které jsou zpracovávány v samostatných jednotkách. Při uvolnění 1. jednotky může jednotka ihned provádět ekvivalentní suboperaci z další operace, díky čemuž je zpracování výrazně urychleno.

Tuto verzi jsem navrhl se snahou o co nejjednodušší návrh, který půjde dobře implementovat a bude srozumitelný. Nejvyšší entita se jmenuje `AES_TOP_01` a ukrývá v sobě 3 moduly. Řadič (`CONTROLLER_01`), datovou cestu (`DATAPATH_01`) a `KEYGEN` ukazuje obrázek 2.1. Řadič je s datovou cestou propojen jediným 4bitovým řídícím signálem `N_RUN`, který posílá datové cestě informaci o tom, v jaké rundě se šifrování právě nachází. Hodnota signálu `N_RUN` může být 0, pokud výpočet neprobíhá nebo 1 – 10, což odpovídá aktuální rundě.

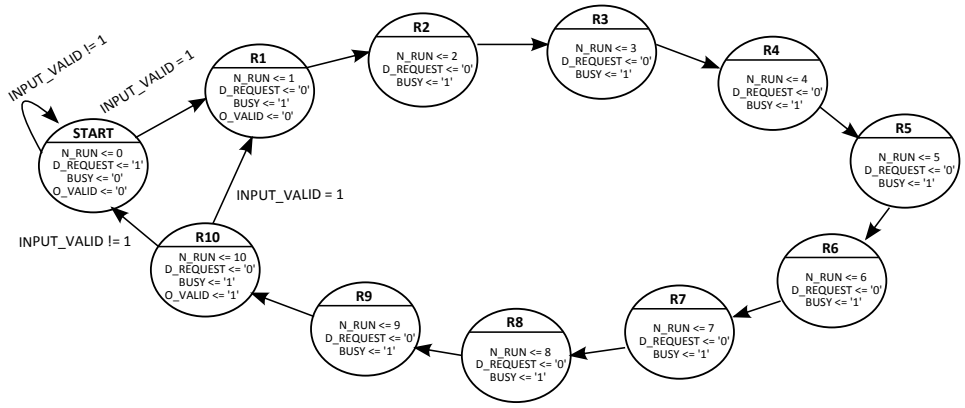


Obrázek 2.1: Nejvyšší entita verze AES01 – AES_TOP_01

Datová cesta se skládá z celkem šesti výpočetních modulů (dvakrát *AddRoundKey*, *SubBytes*, *ShiftRows*, *MixColumns* a *MakeNextKey*), které vykonávají operace samotného šifrování. Modul *MakeNextKey* má na starosti odvozování rundovních klíčů. Dále obsahuje datová cesta tři řídící moduly (*Switch1*, *Switch2*, *Switch3*), které na základě čísla rundy řídí výpočet a nakonec čtyři 16bytové registry – dva na data a dva na klíč. Kompletní schéma datové cesty je v příloze C.1.

Řadič je poměrně jednoduchý. Je navržen jako konečný automat typu *Moore*[7]. Má 11 stavů – `START` a 10 stavů pro každou rundu (`R1` – `R10`), vidět je na obrázku 2.2. Úkolem řadiče je správně nastavovat výstupní signály `DATA_REQUEST`, `BUSY`, `OUTPUT_VALID` (uvedeny v tabulce 2.1) a především `N_RUN`, který dává informace datové cestě o tom, v jaké rundě se výpočet právě nachází. Přechod ze stavu `START` se provede pouze tehdy, je-li signál `INPUT_VALID` roven jedné, což značí platná data na vstupu. Přechody ze stavů `R1` – `R10` probíhají pouze na náběžnou hranu hodin. V obrázku 2.2 není kvůli přehlednosti zaznačen signál `RESET`. Pokud je roven `'1'`, pak se z každého stavu přechází při nejbližší náběžné hraně hodin do stavu `START`.

[7]Konečný automat typu Moore je varianta sekvenčního obvodu, u kterého je výstup závislý pouze na stavu, ve kterém se automat právě nachází.



Obrázek 2.2: Řadič verze AES01 – CONTROLLER_01

Moduly *AddRoundKey*, *SubBytes*, *ShiftRows*, *MixColumns* a *MakeNextKey* jsou navrženy podle sekce 1.2.3, implementačními detaily se budu zabývat v kapitole Realizace (3).

2.1.2 AES jako kombinační logika – AES02

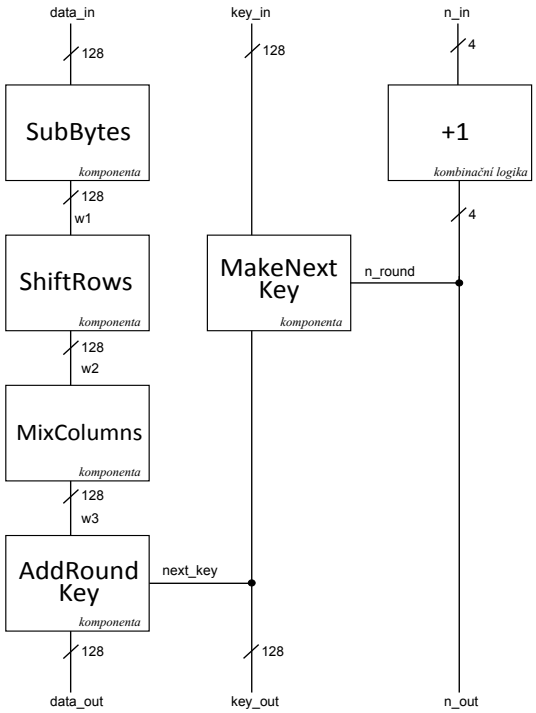
Pro připomenutí opět uvádím základní charakteristiky návrhu v přehledném seznamu.

- Celý AES je kombinační logika.
- Jeden blok dat je zašifrován téměř okamžitě.
- Ve srovnání s AES01 je zabraná plocha čipu přibližně desetinásobná.

Stěžejní myšlenkou tohoto návrhu je maximalizovat rychlost šifrování s cílem zajistit co největší propustnost dat. Proto jsem se rozhodl úplně vypustit sekvenční část. AES02 tedy neobsahuje žádné registry, řadič ani jiné sekvenční obvody. Úlohu řadiče zčásti nahrazuje jinak řešená datová cesta a zčásti nejvyšší entita *AES02_TOP*.

Jedinou možností jak docílit řešení, které nebude závislé na hodinách, je poskytnout každé operaci vlastní modul. Při šifrování jednoho bloku dat se operace *SubBytes*, *ShiftRows* a *MakeNextKey* vykonávají dohromady celkem 30krát, operace *AddRoundKey* se vykonává 11krát a operace *MixColumns* 9krát. Celkem proto obsahuje verze AES02 50 modulů zřetězených za sebe ve správném pořadí. Předpokládám, že kvůli tomu bude obsazovat přibližně 10krát více plochy čipu než verze AES01.

Aby byl návrh přehledný, vytvořil jsem modul `1ROUND`, který provádí jednu rundu šifry (ne poslední). Jak je vidět na obrázku 2.3, provedou se zde čtyři základní operace šifrování a odvodí se rundovní klíč. Protože poslední runda neobsahuje operaci *MixColumns* a jejím výstupem jsou už jen data, vytvořil



Obrázek 2.3: Modul ONE_ROUND verze AES02

jsem pro ni speciální modul s názvem `LAST_ROUND`. Schéma tohoto modulu, datové cesty i top entity je v příloze C.2.

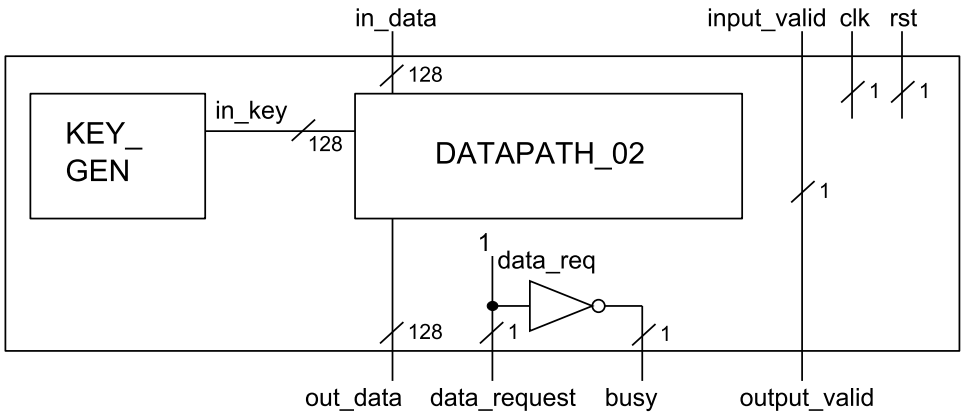
Modul *MakeNextKey* ve verzi AES01 potřebuje pro svoji funkci znát číslo rundy, ve které se nachází. Vzhledem k tomu, že AES02 neobsahuje řadič, který by číslo rundy produkoval, vyřešil jsem absenci řídícího signálu následovně. Datová cesta obsahuje signál, který má konstantně hodnotu 0 a je přiveden na port `n_in` do prvního modulu `1ROUND`. Ten inkrementuje hodnotu o jedna, předá ji modulu *MakeNextKey* a zároveň ji pošle na svůj výstup `n_out`. Postup zachycuje obrázek 2.3. Tímto způsobem si moduly `1ROUND` mezi sebou sami předávají aktuální číslo rundy, tedy každý ví, jaký rundovní klíč má generovat.

Nejvyšší entita `AES_TOP_02` zčásti nahrazuje řadič, když sama určuje výstupní 1bitové signály. Jejich zapojení je zřejmé z obrázku 2.4.

2.2 Spolehlivostní varianty AES

V práci se zabývám nejen návrhem základních variant AES, ale implementuji i šifry s užitím redundance s cílem zabezpečit šifru proti poruchám. V následujících odstavcích krátce pohovořím o návrhu *fault tolerant systémů*[8].

[8]Fault Tolerant systém (FT) je systém navrhovaný s cílem minimalizovat riziko selhání systému v důsledku poruch vzniklých nepříznivým vlivem prostředí či stárnutím.



Obrázek 2.4: Nejvyšší entita verze AES02 – AES_TOP_02

Fault tolerant systémy se používají například v leteckém průmyslu, železniční dopravě, automobilovém průmyslu a dalších. V těchto odvětvích selhání systémů v důsledku poruchy není žádoucí. Při návrhu FT systémů se člověk nevyhne několika základním pojmům. Zde uvádím jejich vysvětlení podle [20] a [21]:

porucha: fyzický defekt

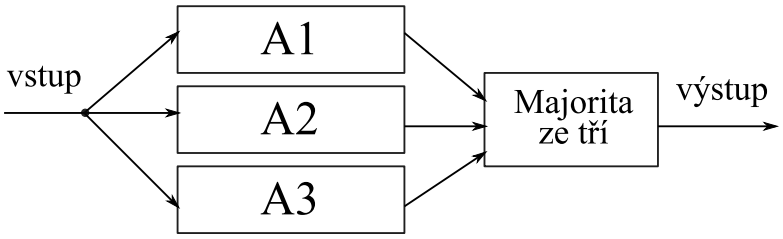
chyba: projev poruchy v systému (například špatná data na signálu)

redundance: duplikování, opakování

Základním principem při návrhu FT systému je použití redundance při návrhu blokového modelu. Systém nechráněný proti poruchám se skládá ze série bloků. Aby systém pracoval bezporuchově, musí fungovat bezporuchově všechny bloky. Jestliže je použita redundance, jsou některé bloky duplikovány a zapojeny paralelně. Pak stačí, aby fungoval alespoň jeden z redundantních bloků a celý systém bude stále fungovat. Takový návrh se v [20] označuje jako paralelní model. Jedním z možných a podle [21] i nejpoužívanějších řešení při modelování redundance je TMR[9]. Princip TMR ilustruje obrázek 2.5. Blok je zkopírován a zapojen paralelně třikrát. Výstupy z bloků kontroluje majorita, která na svůj výstup pošle data, která se shodují alespoň na dvou vstupech. Řešení tedy dokáže eliminovat poruchu jednoho modulu.

Podle toho, jaká veličina je duplikována, rozlišuje se redundance prostorová, časová a informační. Je-li návrh zabezpečen prostorovou redundancí, pak je duplikován blok. Výpočet v duplikovaných blocích probíhá současně. Pokud je použita redundance časová, není duplikován blok, ale výpočet, který proběhne ve stejném hardwaru několikrát po sobě. V případě informační redundance jsou do výpočtu přidávány dodatečné informace jako například paritní

[9]TMR (Triple Major Redundancy) je blokový model pro návrh FT systémů.



Obrázek 2.5: Princip fungování TMR

bity, které jsou závislé na správnosti dat. Jsou-li data během výpočtu vlivem poruchy poškozena, poškodí se i dodatečné informace. To poznají speciální obvody, které na chybu upozorní.

Nyní se obloukem vracím k šifře AES. Předchozí řádky platí obecně pro návrh jakéhokoliv hardware, následující už jsou zaměřeny na návrh šifry AES.

Všechny uvedené spolehlivostní varianty mohou být podle [22] implementovány na různých úrovních. Vysvětlení podává tabulka 2.2. Údaje uvedené v ní čerpají z [22].

Na základě dohody s vedoucím práce jsem navrhl varianty uvedené v tabulce 2.3. Jako zdroj pro všechny spolehlivostní varianty posloužil návrh AES01, který jsem podle požadavků modifikoval.

Tabulka 2.2: Úrovně implementace spolehlivostních variant při použití různých typů redundancí

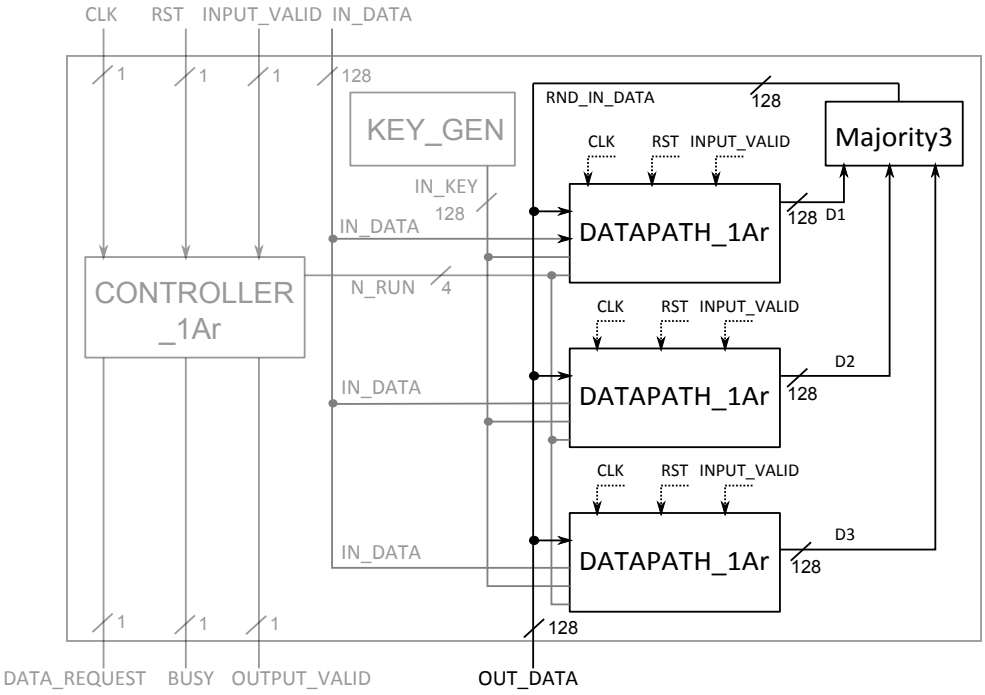
	Prostorová redundance	*Časová redundance*	*Informační redundance*
operace	Každá operace (např. *SubBytes*) je duplikována. Duplikovaný výstup je vyhodnocen a případná chyba signalizována. Toto řešení je ze všech ve sloupci nejpracnější, zabírá nejvíce prostoru, ale má nejrychlejší detekci poruchy.	Provádění libovolné operace (nebo všech) je opakováno. Po každém opakování se výstup uloží do paměti nebo registru. Po posledním opakování se uložené hodnoty porovnají. Tato verze nejrychleji detekuje poruchu.	Před vstupem do operace mohou být ke vstupním datům přidány kontrolní bity. Po výstupu z operace zjistí kontrolní modul jejich správnost a signalizuje případnou chybu.
runda	Duplikována je celá runda. Výstupy jsou vyhodnoceny a případné chyby signalizovány. Je kompromisem mezi redundancí na úrovni operace a algoritmu.	Provádění celé rundy je opakováno. Po každém opakování se uloží hodnoty a ty se nakonec porovnají. Oproti ostatním časovým redundancím má tato verze nejnižší nárůst obsazeného prostoru na čipu.	Nepoužívá se.
algoritmus	Duplikován je celý šifrovací algoritmus. Toto řešení je nejjednodušší na realizaci a obsahuje ze všech prostorových redundancí nejméně vyhodnocovacích obvodů. Avšak detekce poruchy je nejpomalejší.	Po skončení šifrování se celé šifrování provede znovu, atd. Porovnáním výsledků ze všech šifrování je odhalena případná chyba, avšak s největším zpožděním. Výkon takového šifrování je nejhorší, na druhou stranu realizace je velmi snadná.	Nepoužívá se.

Tabulka 2.3: Navržené spolehlivostní varianty algoritmu AES

Úroveň	*Prostorová redundance*	*Časová redundance*	*Informační redundance*
operace			AES_1p
runda	AES_1Ar	AES_1Tr	
algoritmus	AES_1Aa	AES_1Ta	

2.2.1 AES_1Ar

`AES_1Ar` je spolehlivostní varianta s prostorovou redundancí na úrovni rundy. Znamená to, že datová cesta je v šifře třikrát. Při šifrování pracují nezávisle ve stejný čas všechny datové cesty, jejichž výstupy jsou zpracovány v modulu `Majority3`. Ten porovná u vstupů všechny jejich i-té bity a na výstup s indexem i pošle hodnotu, která na vstupech převažuje.



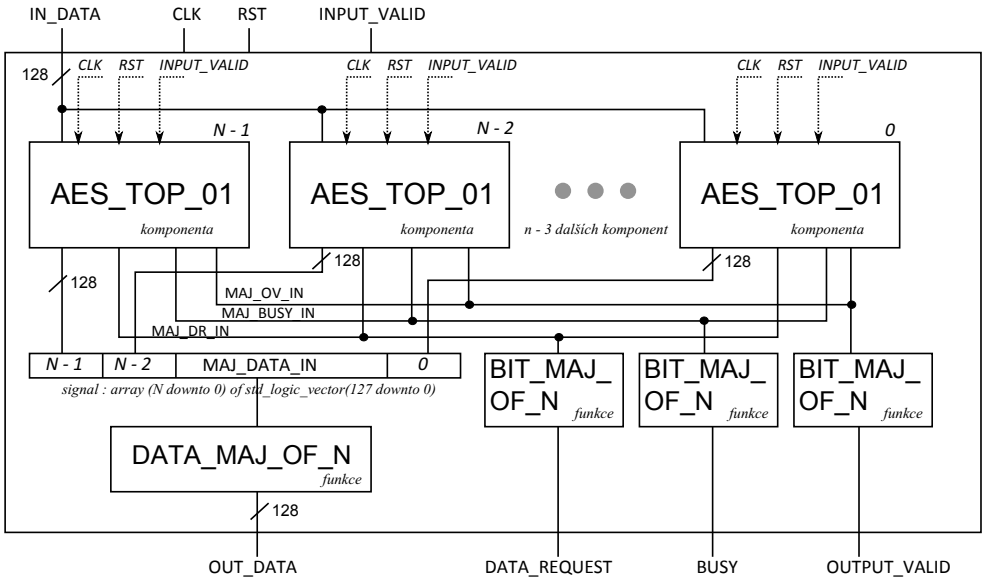
Obrázek 2.6: AES_TOP_1Ar – změny oproti AES01

Oproti základní verzi `AES01` má datová cesta jediný rozdíl. Data opouštějící každou rundu se nevrací do registru jako vstupní pro další rundu, ale jsou výstupním portem `R_OUT_DATA`. Data, která vstupují do další rundy, přichází do registru `REG_DATA2` z nového vstupního portu `R_IN_DATA`. Ve vyšším modulu

musí rundovní data nejprve projít majoritou, než se dostanou do datové cesty pro šifrování v další rundě. Řadič je zde s verzí `AES01` shodný. Největší změny jsou viditelné v top modulu. Datová cesta je třikrát. Rundovní data určuje modul `Majority3` a posílá je zpět do všech datových cest, jak je vidět na obrázku 2.6. Kompletní dokumentace je na straně 70.

2.2.2 AES_1Aa

`AES_1Aa` je spolehlivostní varianta s prostorovou redundancí na úrovni algoritmu. Celá hotová šifra `AES01` se $n$krát zkopíruje. Šifrování probíhá paralelně ve všech modulech a trvá tedy stejný čas, jako kdyby se šifrovalo základní verzí. Výhodou této varianty je, že má parametrizovatelný počet modulů `AES_TOP_01`. Změnou jediné konstanty v kódu je možné určit počet opakování. Top modul obsahuje n bloků `AES_TOP_01`, blok `DATA_MAJ_OF_N` a 3 bloky `BIT_MAJ_OF_N`. Vše zachycuje obrázek 2.7.



Obrázek 2.7: Modul AES_TOP_1Aa verze AES_1Aa

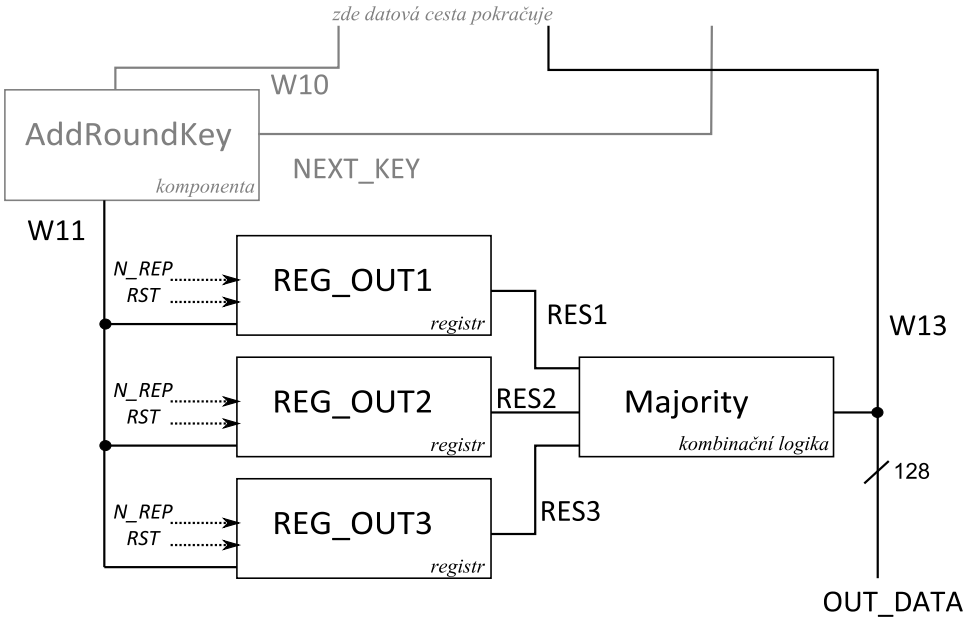
Modul `DATA_MAJ_OF_N` přijímá pole délky n, jehož každý prvek je 16bytový vektor. Výstupem je jeden 16bytový vektor, jehož každý bit je vypočítán jako majorita[10] z odpovídajících bitů v prvcích pole `MAJ_DATA_IN`. Vstupem modulu `BIT_MAJ_OF_N` je pole délky n, jehož prvky jsou jednobitové. Výstup o šířce jeden bit je vypočten jako majorita ze všech prvků vstupního pole.

[10]Majorita je funkce, která z libovolného lichého počtu jednobitových vstupů vytvoří jeden jednobitový výstup, kde bude ta hodnota, která na vstupních vodičích převažuje.

2.2.3 AES_1Tr

`AES_1Tr` je spolehlivostní varianta s časovou redundancí na úrovni rundy. Na rozdíl od verzí `AES_1Ar` a `AES_1Aa` zde spočívá redundance v opakování postupu šifrování prostřednictvím stejného hardware. Každá runda se opakuje třikrát, výsledky z jednotlivých opakování se uloží a poté porovnají. K tomu je třeba provést několik drobných změn v datové cestě a značně předělat řadič. Šifrování zákonitě probíhá trojnásobně delší dobu než prostřednictvím `AES01`.

Velice důležitým se stal nový signál `N_REP`, který počítá opakování. Vzhledem k tomu, že `AES_1Tr` opakuje každou rundu 3krát, signál `N_REP` je 2bitový a může nabývat hodnot od 0 do 2 (číslo 2 odpovídá třetímu opakování). Ovlivňuje správné ukládání rundovního klíče do registru, protože komponenta *MakeNextKey* je nastavena tak, aby stále odvozovala rundovní klíč. Ukládání rundovního klíče se tedy provádí jen jednou za 3 takty. Na konec rundy jsem do datové cesty přidal 3 registry (16bytové), které ukládají hodnotu dat po každém opakování. Například registr `REG_OUT1` ukládá hodnotu po prvním průchodu dat rundou. Analogicky pracují zbylé registry. Kdy si má který registr uložit data, závisí na signálu `N_REP`. Výstupy z registrů zpracovává majorita. Rozšíření datové cesty je na obrázku 2.8.



Obrázek 2.8: Výřez datové cesty AES_1Tr - registry a majorita

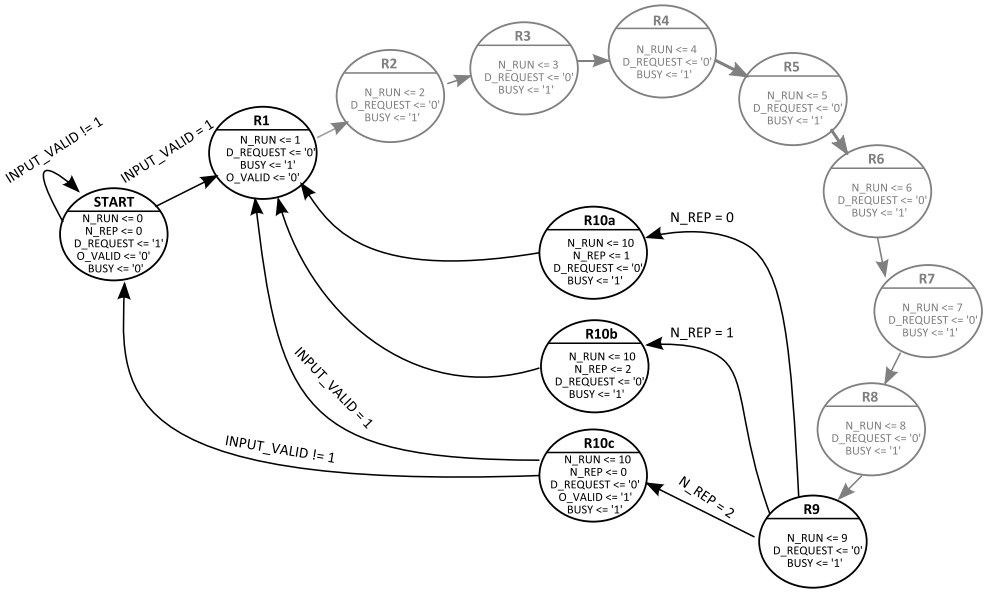
V případě časové redundance má majorita jednu zásadní nevýhodu. Existuje určitá pravděpodobnost, že i přes ochranu nebude případná chyba detekována. Pokud se například porouchá modul `X`, který 3krát provádí šifrování,

vypočte se šifrování 3krát s tou samou chybou. Ta není díky svému opakování pomocí majoritní funkce detekovatelná.

Řadič je navržen jako konečný automat s 11 stavy (stejně jako u `AES01`). Mezi stavy se přechází, pokud je `N_REP` = 2. Součástí je čítač `COUNTER_MOD3`, který za předpokladu, že probíhá šifrování, periodicky generuje hodnoty signálu `N_REP` v rozsahu 0 – 2. Všechna schémata jsou v dokumentaci od strany 73.

2.2.4 AES_1Ta

`AES_1Ta` je spolehlivostní varianta s časovou redundancí na úrovni algoritmu. Tedy po prvním šifrování dat (10 rund) se ta samá vstupní data ještě dvakrát znovu zašifrují. Šifrování proběhne celkem 3krát za sebou, zabere tedy stejnou dobu jako `AES_1Tr`.



Obrázek 2.9: Řadič verze AES_1Ta – nejdůležitější stavy

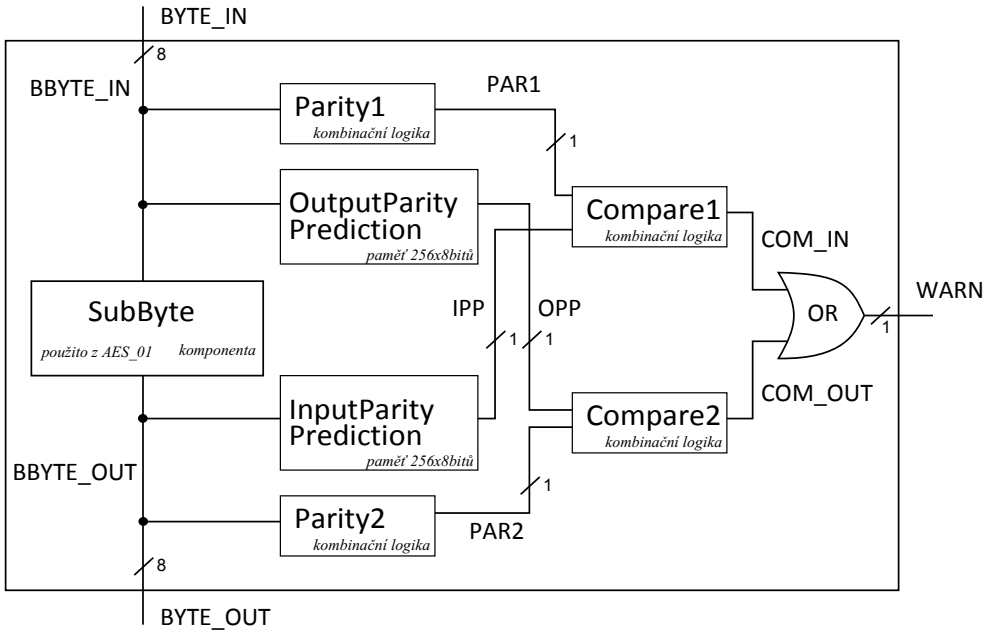
Datová cesta obsahuje navíc 3 registry pro ukládání zašifrovaných dat. Do každého registru se uloží hodnoty po jednom ze tří opakování. Aby se ukládala správná data, řídí jejich ukládání do registrů signály `N_RUN` (číslo rundy) a `N_REP` (číslo opakování). Nakonec jsou data extrahována pomocí majority ze tří.

Konečný automat, který implementuje řadič, má poslední stav rozdělen na 3 stavy – `R10a`, `R10b`, `R10c`. Do nich se řízení dostává podle aktuální hodnoty signálu `N_REP`. Je-li mezi stavy šipka bez popisu, znamená to, že přechod je jen na náběžnou hranu hodin. Všechny informace o řadiči udává obrázek 2.9. Přehled všech schémat je v dokumentaci na straně 76.

2.2.5 AES_1p

`AES_1p` je spolehlivostní varianta s informační redundancí na úrovni operace. Od ostatních spolehlivostních variant se značně odlišuje ve dvou ohledech. Zaprvé se jedná o informační redundanci, takže do návrhu přidávám bity navíc, které v mém případě generuje sudá parita[11] (odtud písmeno „p“ v názvu varianty). Druhou odlišností je redundance na úrovni jediné operace – *SubBytes*. Při návrhu této varianty jsem se inspiroval v [23]. Aniž bych to dále explicitně uváděl, čerpá mnoho údajů v této sekci právě z [23].

Na rozdíl od ostatních nezachovává modul *SubBytes* paritu, což vyplývá z principu jeho fungování uvedeného v sekci 1.2.3.1. Proto je nutné do něj přidat dodatečné obvody, které umí predikovat na základě vstupních dat paritu na výstupu a obráceně. I po přidání parity zůstane modul *SubBytes* kombinačním obvodem. Hlavní myšlenkou je, že není třeba upravovat jádro *SubByte*, které řeší výměnu jednoho bytu za jiný byte. Ze vstupního bytu spočítám paritu a porovnám ji s paritou, kterou předpovídá blok `InputParityPrediction` na základě výstupu. Navíc spočítám paritu z výstupu a srovnám ji s výsledkem z bloku `OutputParityPrediction`. Při každé náhradě bytu provádím dvojí kontrolu parity.



Obrázek 2.10: AES_1p – dvojí předpověď parity

[11] Parita je funkce, která sečte všechny jedničky na vstupu a přidá jeden bit s hodnotou 1 nebo 0 tak, aby součet byl vždy sudý/lichý.

Význam bloků z obrázku 2.10 vysvětluje následující seznam:

- `Parity1/2` vytváří jednobitovou paritu z 8bitového vstupu.
- `Compare1/2` porovná 2 jednobitové vodiče a pokud se shodují, na výstup pošle `'0'`.
- Jestliže alespoň jeden srovnávací obvod vysílá `'1'`, došlo k chybě a výstup `WARN` je také `'1'`.

Jak fungují bloky `InputParityP.` a `OutputParityPrediction` názorně vysvětluje obrázek 2.11. Nejprve se vytvoří tabulka se čtyřmi sloupci. V 1. sloupci je vstup, ve druhém sloupci je parita vypočtená ze vstupu. Do třetího sloupce se dosadí byty, které vrátí *SBox* na základě vstupu z 1. sloupce a konečně do 4. sloupce se vypočítá parita podle třetího sloupce. Predikce vstupní i výstupní parity se určuje podle tabulek, které vzniknou správným rozdělením výše popsané tabulky.

vstup	*parita*	*výstup z SBox*	*parita z výstupu*
0000 0000	0	0110 0011	0
0000 0001	1	0111 1100	1
0000 0010	1	0111 0111	0
0000 0011	0	0111 1011	0
...	...	...	...

OutputParityPrediction	
0000 0000	0
0000 0001	1
0000 0010	0
0000 0011	0
...	...

InputParityPrediction	
0110 0011	0
0111 1100	1
0111 0111	1
0111 1011	0
...	...

Obrázek 2.11: AES_1p – vypočítané paritní tabulky

Kapitola 3

Realizace

Zde se čtenář seznámí s implementačními detaily práce a s také s naměřenými výsledky pro vytvořené varianty šifry AES. Kapitola obsahuje konkrétní ukázky, jak jsem programoval některé funkčnosti šifry, nebo které problémy jsem při práci řešil. To vše by nešlo bez softwarové výbavy, pomocí které jsem šifru nejprve naprogramoval, následně otestoval a nakonec změřil její vlastnosti. Hned na začátku je proto stručný popis použitého softwaru.

3.1 Použitý software

K realizaci této bakalářské práce jsem použil software Quartus II [24], ModelSim SE [25], Rijndael Animation [12] a Rijndael Inspector [26]. K vytvoření textové části bakalářské práce ve formátu `PDF` jsem použil sázecí systém LaTeX [27]. Téměř veškerý kód, který jsem pro bakalářskou práci vytvořil, jsem psal v textovém editoru Sublime Text [28]. Vektorové obrázky schémat a automatů jsem tvořil v programu Inkscape [29]. Použití těchto tří programů není úzce spjato s tématem mojí práce, proto se o nich dále již nezmiňuji.

Rijndael Animation je webová prezentace, která velmi názorně ukazuje, jak algoritmus Rijndael pracuje. Pro moji práci měla stěžejní význam. V počátečních fázích mi pomohla pochopit princip šifrování a vřele ji doporučuji ke shlédnutí každému, kdo by chtěl porozumět šifře AES [12]. Quartus II slouží k vytváření programů pro FPGA. Ve své práci jsem použil Quartus k provedení syntézy (více o syntéze v sekci 3.4). Posledním a velice důležitým programem, který mi při práci pomohl, je ModelSim. Na základě předloženého testovací souboru (testbenche) provádí simulace číslicových obvodů popsaných v jazyku VHDL. Při psaní této práce jsem ModelSim použil především k testování a často jsem s jeho pomocí prováděl kontrolu syntaxe zdrojových kódů. Díky bakalářské práci jsem se s programem naučil pracovat na základní úrovni.

3.2 Realizace základních modulů

V této kapitole uvádím, jak jsem naprogramoval některé základní operace šifry AES. Zatímco u modulů *AddRoundKey* a *ShiftRows* není díky jejich jednoduchosti co vysvětlovat, u některých jiných je možné zvolit vícero přístupů, z nichž některé ukážu. Podle zadání jsem pro realizaci použil jazyk pro popis hardwaru VHDL.

Předpokládám znalosti programování v jazyku VHDL, které nemohou být pro svoji obsáhlost součástí této práce. Čtenářům, kteří nemusí rozumět implementačním detailům, nebo nemají alespoň minimální znalosti programování, doporučuji tuto kapitolu přeskočit.

3.2.1 Realizace operace SubBytes

Při velkém zjednodušení lze tvrdit, že operace SubBytes je náhrada jednoho bytů podle nějakého pravidla za jiný byte. V tom případě jsou nasnadě 2 přístupy. Buď budu při každém vstupu uplatňovat pravidlo a počítat výstup nebo si vypočítám všechny výstupy předem a uložím je do asociativní tabulky. Druhý přístup je rychlejší, první zas spotřebuje méně paměti. Já jsem se rozhodl pro implementaci pomocí tabulky. Podle mého názoru je tento přístup přehlednější, tudíž je i menší pravděpodobnost výskytu chyby. Modul *SubBytes* reprezentuje entita `SUB_BYTES`. Uvnitř architektury `SUB_BYTES_BODY` implementující entitu jsem deklaroval[12] pole dvěstěpadesátišesti 1bytových vektorů:

```
type TAB256 is array (0 to 255) of std_logic_vector(7 downto 0);
constant SBOX_TABLE : TAB256 := (
    x"63", x"7C", x"77", ...
);
```

Přijde-li na vstup byte, pak považuji jeho hodnotu za adresu do paměti `SBOX_TABLE` a výstupem je hodnota uložená v paměti na příslušné adrese. Protože paměť je organizována jako lineární jednorozměrné pole o velikosti 256 prvků, mohu převádět hodnoty 1bytových vstupů na adresy tak, že horní čtyři bity násobím šestnácti a dolní 4 bity přičtu:

```
DATA_OUT_GEN: for i in 0 to 15 generate
    DATA_OUT((i*8)+7 downto (i*8)) <= SBOX_TABLE(16 *
              conv_integer((DATA_IN((i*8)+7 downto (i*8)+4)))
              + conv_integer((DATA_IN((i*8)+3 downto (i*8)))) );
end generate DATA_OUT_GEN;
```

[12]Deklarace je pojem z oblasti programovacích jazyků. Signál/pole/proměnná je deklarována tam, kde se v kódu poprvé objeví. Obvykle je to na začátku programu.

Protože vstupní data do modulu *SubBytes* mají šířku 16 bytů, je třeba vybrat 16krát jeden byte, nahradit ho a byty ve správném pořadí uložit do výstupního signálu `DATA_OUT`. K tomu jsem použil příkaz *for – generate*, který provádí paralelní zapojení několika stejných komponent. Uvnitř cyklu vybírám z tabulky správné hodnoty a ukládám je do výstupního signálu. Funkce *conv_integer* slouží ke konverzi typu *std_logic_vector* na *integer*, protože pole se adresuje datovým typem *integer*.

3.2.2 Realizace operace MixColumns

Modul *MixColumns* v mojí implementaci reprezentuje entita `MIX_COLUMNS`. Její architektura obsahuje 3 procesy, které dělí 16bytový vektor na čtyři 4bytové, násobí každý vektor matic a nakonec zase spojí vynásobené vektory do jednoho výstupního. Za pozornost stojí násobení maticí, které jsem implementoval pomocí funkce `MULT_BY_MATRIX()`. Jejím jediným parametrem je 4 B dlouhý vektor, který představuje jeden sloupec dat.

Funkce provádí sčítání a násobení podle předpisu uvedeného v sekci 1.2.3.3, přičemž samotné násobení jednoho bytu je prováděno pomocí dalších funkcí. Vzhledem k tomu, že operace násobení je v hardwaru náročná na čas (na rozdíl od sčítání se musí provádět sekvenčně) a je i poměrně komplikovaná na implementaci, provedl jsem realizaci jinak. Po prozkoumání násobící matice jsem zjistil, že obsahuje pouze koeficienty `01`, `02` a `03`. Pokud je byte násoben číslem 1, výsledek je shodný se vstupem, k tomu není třeba funkce. Vytvořil jsem tedy pouze 2 funkce. `MULT_2()` zadaný parametr vynásobí dvěma a `MULT_3()` provádí násobení zadaného parametru třemi. Funkce pro násobení dvěma provede bitový posun o 1 pozici vlevo a pokud vyjde výsledek devítibitový, provede redukci modulo polynom $P(x) = x^8 + x^4 + x^3 + x + 1$. Zde je ukázka části funkce `MULT_2()` zapsaná ve VHDL:

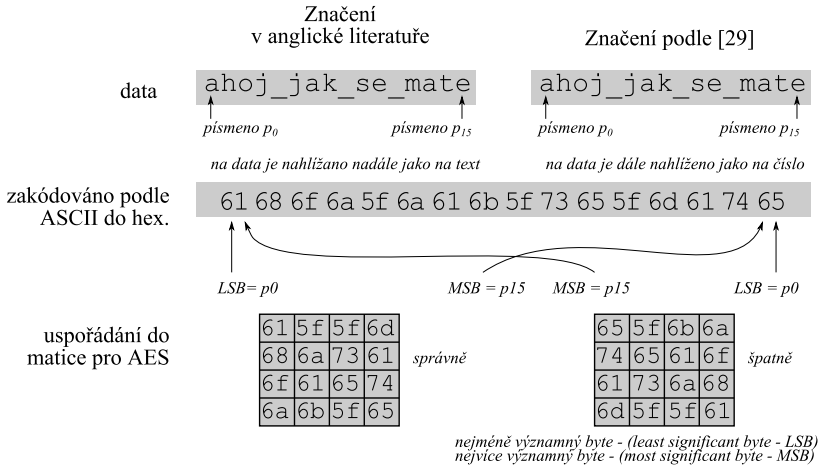
```
A(8 downto 1) := DATA; -- logicky posun vstupnich dat o~1 <--
A(0) := '0'; -- dopln zprava nulou
if A(8) = '1' then -- pokud je vysledek devitibitovy
  POLYNOM := "100011011";
  A~:= A~xor POLYNOM; -- zkrat vysledek na 8 b delenim P(x)
end if ;
RESULT := A(7 downto 0);
return RESULT;
```

Funkce pro násobení třemi je výše uvedené velice podobná. Také provede bitový posun o jedna vlevo a navíc k výsledku přičte původní hodnotu. Matematicky provádí přesně toto: $3 * a = 2 * a + a$. Je-li výsledek devítibitový, provede se opět redukce modulo $P(x)$.

Násobení v modulu *MixColumns* je možné podle [9] provádět také s pomocí dvou asociativních tabulek. V první budou předpočítány všechny možné

násobky dvou a v druhé všechny možné násobky tří. Při šifrování si algoritmus místo násobení jen vyzvedne z tabulky výsledek. To je obdobný přístup, jako jsem použil při implementaci modulu *SubBytes*. Nejenže je řešení pomocí tabulky výrazně náročnější na paměť, má i další nevýhodu. Při implementaci bych musel nějakým způsobem hodnoty násobků do tabulek vypočítat. To by zabralo mnohem více času než mnou použitý přístup. Tabulka pro *SubBytes* je snadno k nalezení na internetu, její obsah jsem počítat nemusel.

Implementace modulu *MixColumns* byla v mém případě nejsložitější ze všech operací AES. Dlouhou dobu jsem nemohl přijít na zdroj chyby, kterou jsem v kódu měl. Šlo o nesprávné zapojení správně vypočítaných bytů a prohození výsledných vektorů. Problém byl způsoben pro mě nelogickým značením vektorů ve zdrojové literatuře [7] a [9], odkud jsem čerpal. Značení je v rozporu s přednáškou [30], podle které jsem byl zvyklý značit delší vektory bytů. Totiž, že nejméně významný bit (dále jen LSB) datového vektoru bitů je nejvíce napravo a má označení X_0, zatímco nejvíce významný bit (dále jen MSB) je vlevo. V literatuře je nejnižší značený byte označen s indexem 0, avšak pozičně je umístěn nejvíce doleva, což je matoucí. Odtud pramenily moje chyby. V některých případech (například právě v operaci *MixColumns*) je třeba data chápat jako číslo, protože vstupují například do operace násobení a tedy na tom, který byte je LSB a který MSB, záleží.

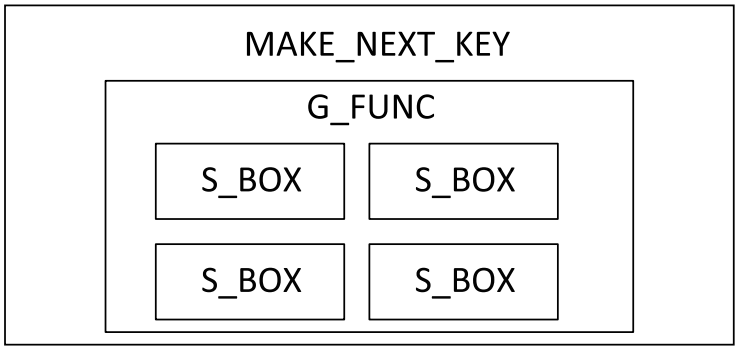


Obrázek 3.1: Porovnání značení bytů a jejich uspořádání do matice

Avšak později se ukázalo, že výše popsaná domněnka je mylná. Správný výklad je, že poziční uspořádání v literatuře (zejména v [9]) je dobře, ale číslování je otočeno, tedy nejvíce významný byte vlevo má hodnotu X_0, nejméně významný vpravo pak X_{15} (nejnižší řád). Chyba ve značení a správném uspořádání bytů do matice měla vliv také na moduly *ShiftRows* a *KeySchedule*. Přístup, který je používán v anglické literatuře, srovnává s přístupem, který je mi bližší, obrázek 3.1.

3.2.3 Realizace modulu KeySchedule

KeySchedule jsem implementoval jako šest komponent. Nejvyšší z nich je `MAKE_NEXT_KEY`, která na základě aktuálního klíče a čísla rundy odvodí klíč následující. Uvnitř je `G_FUNC`, jejíž princip vysvětluje obrázek 1.6. Součástí entity `G_FUNC` jsou 4 komponenty `S_BOX`. Zapouzdření komponent je vidět na obrázku 3.2.



Obrázek 3.2: Komponenty modulu KeySchedule

Uvnitř komponent je provedení poměrně jednoduché. U všech je jejich fungování popsáno jedním procesem, kde jsem si vystačil s operací *xor* a přiřazovacím operátorem `<=`. Komponenta `SUB_BYTE` je stejně jako modul *SubBytes* realizována pomocí asociativní tabulky.

3.3 Realizace variant šifry AES

Standardním postupem při návrhu rozsáhlejšího hardwaru je dekompozice celku na 2 menší části – datovou cestu a řadič. Datová cesta je dále rozdělena na soustavu malých komponent, kde každá plní jednoduchou funkci. Příkladem komponenty může být třeba operace *SubBytes* Tyto komponenty jsou následně spojeny do celku ovládaného řídícími signály, které generuje řadič. Jak datová cesta, tak řadič jsou vždy uloženy v samostatném souboru. Nejvyšším modulem je pro každou variantu AES takzvaná *top entita*, v níž jsou propojeny datová cesta a řadič do finálního produktu. Při pojmenovávání jsem se držel pravidla, že název datové cesty začíná klíčovým slovem `datapath`, název řadiče začíná slovem `controller` a *top entita* je pojmenována `aes_top`. Název každé entity je zakončen příponou označující variantu AES.

Datové cesty mají v mých návrzích na starost správné propojení komponent do jednoho celku. Obvykle neprovádí s daty žádné operace (xor, and, ...), k tomu slouží komponenty na nižší úrovni. Pokud varianta AES obsahuje řadič, postupoval jsem při jeho realizaci bez ohledu na variantu shodně. Řadič navržený jako konečný automat typu Moore jsem ve VHDL realizoval

vždy jako jednu entitu, jejíž architektura se skládá ze tří procesů. Jeden proces řeší kombinační logiku přechodů, druhý nastavuje výstupy a poslední implementuje registr s resetem, který si pamatuje stav, ve kterém se automat právě nachází.

Jak už bylo řečeno, všechny spolehlivostní varianty vycházejí z varianty `AES01` a tam, kde to je možné, mají shodný kód. Pro snadnější orientaci jsem vytvořil tabulku vzájemné shody jednotlivých implementací. Tabulka 3.1 uvádí na levé straně přehled všech entit a na pravé straně je uvedeno, ve kterých variantách šifry se daná entita vyskytuje. Jedna entita je vždy uložena v jednom souboru se shodným názvem.

Není cílem zde detailně rozepsat, jak jsem každou verzi vytvořil, to by pak jen tato sekce měla desítky stran. Základní povědomí o variantách AES poskytuje čtenáři kapitola Návrh (2). Tato sekce bude spíše extraktem těch nejzajímavějších použitých technik při programování jednotlivých verzí.

3.3.1 AES_1Aa – nastavitelná redundance

Na pokyn vedoucího práce jsem sestrojil jednu ze spolehlivostních variant tak, aby míra redundance byla nastavitelná. Nejvhodnější verzí pro takovou realizaci je AES s prostorovou redundancí na úrovni algoritmu. Výsledkem je AES, kde mohu změnou jediné konstanty v kódu určit, kolikrát bude celý AES na ploše FPGA zkopírován.

Top entita `AES_TOP_1Aa` (na obrázku 2.7 na straně 26) může obsahovat libovolný počet entit `AES_TOP_01`. Tento počet budu dále označovat konstantou n. Aby bylo možné generovat libovolný počet entit, musí být komponenta `AES_TOP_01` uvedena v deklarační části architektury. Dále je třeba mít uvnitř entity `AES_TOP_1Aa` deklarovanou konstantu n: `generic (N: integer:=2);`. Vygenerování n entit se v jazyku VHDL zapíše pomocí příkazu *for – generate*:

```
GENERATE_AES_top_modules: for i in 0 to N generate
    AES_TOP_X : AES_TOP_01
        port map (
            -- napojeni vstupnich signalu
            CLK           => CLK,
            RST           => RST,
            INPUT_VALID   => INPUT_VALID,
            IN_DATA       => IN_DATA,
            -- a tak dale ...
        );
end generate GENERATE_AES_top_modules;
```

Napojení vstupních signálů (hodiny, reset, ...) je prosté, protože signál je možné rozvětvit jednoduše. Horší je práce s výstupními signály. Každá z n redundantních entit produkuje 4 výstupy – `OUT_DATA` o šířce 16 bytů

Tabulka 3.1: Tabulka vzájemné shody jednotlivých implementací

	entita			*varianta AES...*						
id	*jméno*	*úroveň*[1]	*logika*[2]	*vnořené entity*[3]	*01*	*02*	*_1Ta*	*_1Tr*	*_1Aa*	*_1Ar*	*_1p*
1	add_round_key	3	K	–	ano	ano	ano	ano	ano	ano	ano
2	aes_top	8	S	4, 6	*_01	*_02[4]	*_1Ta	*_1Tr	*_01	*_1Ar	*_01
3	aes_top_1Aa	9	S	aes_top_01	ne	ne	ne	ne	ano	ne	ne
4	controller	7	S	5	*_01	ne	*_1Ta	*_1Tr	*_01	*_1Ar	*_01
5	counter_mod3	3	S	–	ne	ne	ne	*_1Tr	ne	ne	ne
6	datapath	7	S	1, 9, 10, 13, 15	*_01	*_02[4]	*_1Ta	*_1Tr	*_01	*_1Ar	*_1p
7	g_func	4	K	14	ano	ano	ano	ano	ano	ano	ano
8	last_round	6	K	1, 9, 13, 15	ne	ano	ne	ne	ne	ne	ne
9	make_next_key	5	K	7	ano	ano	ano	ano	ano	ano	ano
10	mix_columns	3	K	–	ano	ano	ano	ano	ano	ano	ano
11	one_round	6	K	1, 9, 10, 13, 15	ne	ano	ne	ne	ne	ne	ne
12	sbox_with_parity	2	K	14	ne	ne	ne	ne	ne	ne	ano
13	shift_rows	3	K	–	ano	ano	ano	ano	ano	ano	ano
14	sub_byte	1	K	–	ano	ano	ano	ano	ano	ano	ano
15	sub_bytes	3	K	–	ano	ano	ano	ano	ano	ano	*_with_parity

Poznámky

ne – varianta neobsahuje tuto entitu

**popis* – varianta obsahuje tuto entitu, ale entita se jmenuje *starejmeno+popis* a má jinou implementaci

ano – varianta obsahuje přesně tuto entitu se stejnou implementací

(1) úroveň entity – čím nižší číslo, tím konkrétnější, čím vyšší číslo, tím více může obsahovat v sobě zanořených komponent

(2) K – kombinační logika, S – sekvenční logika

(3) id komponent, které jsou obsaženy uvnitř dané entity

(4) zde se jedná o kombinační obvod

a tři jednobitové signály `OUTPUT_VALID`, `DATA_REQUEST` a `BUSY`. Abych mohl zpracovat čtyři n-tice výstupních signálů z redundantních entit, vytvořil jsem si čtyři pole proměnlivé délky n:

```
-- datovy typ pole pro DATA_OUT
type ARRAY_128 is array (N downto 0)
  of std_logic_vector(127 downto 0);
-- datovy typ pole pro BUSY, OUTPUT_VALID, DATA_REQUEST
type ARRAY_1 is array (N downto 0) of std_logic;

-- vytvoreni signalu promenlive delky
-- pole pro vsechny DATA_OUT vystupy z~vice AESu
signal MAJ_DATA_IN : ARRAY_128;
```

Při napojování redundantní top entity s pořadovým číslem i napojím její výstup `OUT_DATA` jako i-tou položku pole `MAJ_DATA_IN(i)`. To se provede uvnitř příkazu *for – generate*. Zpracování polí a jejich následnou redukci na velikost jejich jediného prvku provádí dvě funkce. Obě dávají na výstup majoritu. Liší se jen v tom, že jedna funkce (`DATA_MAJ_OF_N`)přijímá pole 16bytových vektorů o délce n, zatímco druhá (`BIT_MAJ_OF_N`)přijímá pole jednobitových prvků taktéž o délce n. Funkce `DATA_MAJ_OF_N` je realizována pomocí dvou vnořených cyklů. Ve vnějším cyklu program iteruje pomocí proměnné i přes všechny bity 128bitového vektoru. Následně vstoupí do vnitřního cyklu, kde iteruje pomocí proměnné j přes všechny prvky pole. Ve vnitřním cyklu provede součet hodnot na i-tých pozicích přes všechny prvky pole. Je-li součet větší než $n/2$, pak umístí na pozici i výstupního vektoru hodnotu `'1'`, jinak `'0'`. Funkce `BIT_MAJ_OF_N` je realizována podle stejného principu.

3.3.2 AES_1Tr – odlišnosti v řadiči

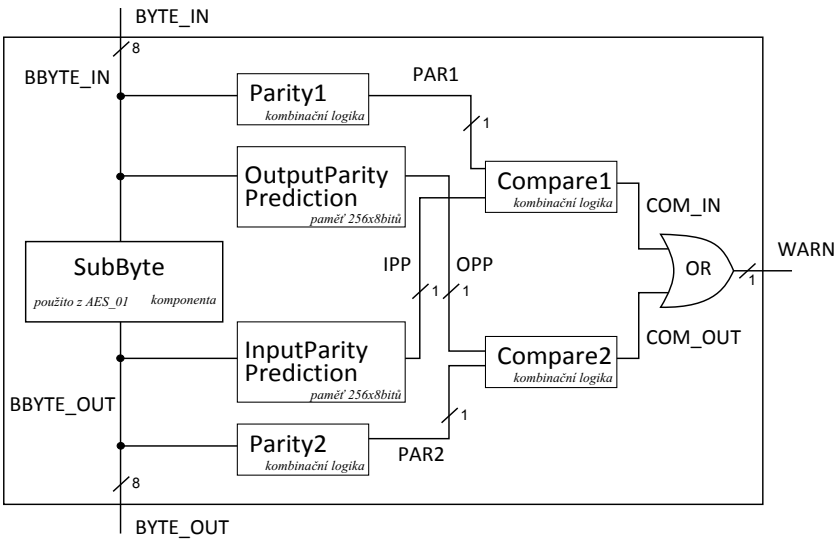
Ve srovnání s ostatními řadiči, se ten u verze `AES_1Tr` (dokumentace na straně 73) v několika aspektech liší. Varianta `AES_1Tr` implementuje časovou redundanci na úrovni rundy, tedy každá runda proběhne třikrát a až poté postoupí šifrování do rundy následující. Počet stavů automatu se oproti `AES_01` nezměnil, bylo však třeba nějak zabudovat do automatu časovou redundanci. K tomu jsem použil čítač `COUNTER_MOD3_1Tr`, který čítá stále dokola od nuly do dvou a svoji hodnotu zvyšuje na každou náběžnou hranu hodin. Čítač jsem implementoval jako samostatnou entitu a do řadiče jsem ho vsadil jako komponentu. Čítač má mimo jiné vstupní port `N_RUN` (číslo rundy), protože čítá jen tehdy, není-li číslo rundy rovno nule. Výstupem čítače je dvoubitový výstup `N_REP` (číslo opakování).

Přechody ze stavů `R1` – `R10` probíhají pouze tehdy, pokud je číslo opakování rovno dvěma. Díky tomu zůstává řadič v každém stavu 3 hodinové takty a datová cesta tedy třikrát za sebou zůstává ve stejné rundě.

Také určení, kdy je výstup platný, nyní probíhá v samostatném procesu uvnitř řadiče. `OUTPUT_VALID` se nastaví na `'1'` pouze tehdy, pokud je číslo rundy rovno deseti a číslo opakování rovno dvěma. První podmínka (`N_RUN`) sama o sobě nestačí, protože pak by byl signalizován platný výstup ještě před tím, než se dopočítají redundantní výsledky v 10. rundě. Pokud by nastala porucha, jejich dopočítání by pro určení správně zašifrovaných dat bylo nezbytné.

3.3.3 Realizace verze AES_1p

Varianta `AES_1p` je zabezpečena paritou na úrovni operace *SubBytes*. Operace *SubBytes* je v ostatních verzích AES implementována jako jediná entita, která neobsahuje další komponenty. Vzhledem ke složitosti realizace s paritou jsem se rozhodl sestavit modul `SUB_BYTES_WITH_PARITY` ze tří vnořených komponent. Ten obsahuje pouze správné propojení. Pomocí příkazu *for – generate* je 16krát napojena komponenta `SBOX_WITH_PARITY`. Jejím vstupem je 1 byte vstupních, 1 byte výstupních dat a signál `WARN` o šířce 1 b, který signalizuje případnou poruchu. Komponenta vykonává přesně to, co je zobrazeno na obrázku 3.3.



Obrázek 3.3: Entita SBOX_WITH_PARITY – verze AES_1p

Realizaci komponenty `SBOX_WITH_PARITY` jsem provedl dvěma způsoby. Každému způsobu odpovídá jedna architektura, entita tedy obsahuje dvě architektury. Při napojování do nadřazené entity se výběr požadované architektury provede zápisem jména architektury do kulatých závorek za název entity:

```
SBOX_WITH_PARITY_X :
  entity work.SBOX_WITH_PARITY(SBOX_WITH_PARITY_1)
  port map ( ... );
```

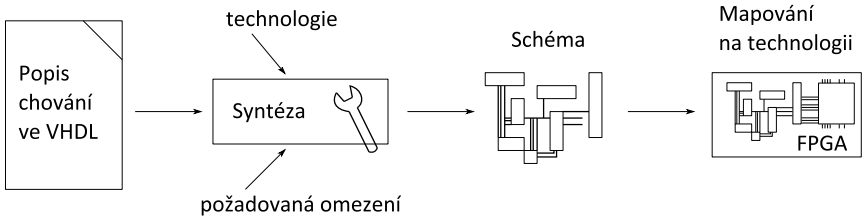
První architektura implementuje operace `Parity1`, `Parity2`, `Compare1` a `Compare2` jako procesy a vodiče mezi nimi jsou signály. Druhá architektura oproti první realizuje všechny výše uvedené operace jako přiřazení v rámci jediného obsáhlého procesu a vodiče mezi nimi jsou považovány za proměnné. V obou architekturách jsou shodně predikce parity realizovány s pomocí asociativních tabulek, které jsou umístěny v deklarační části architektur. Pro jednoduchost navíc stačí, když jsou tabulky realizovány jako jednorozměrné pole. Následující úsek kódu ukazuje, jak se deklaruje datový typ `TABLE_256x1` a jak se deklaruje pole tohoto typu.

```
type TABLE_256x1 is array (0 to 255) of std_logic;
-- tabulka pro "Output parity prediction"
constant OUTPUT_PARITY_PREDICTION : TABLE_256x1 := (
  '0', ...
);
```

Pokud by mě například zajímalo, jaká bude hodnota parity pro vstup 53_{hex}, pak vypočítám $5 * 16 + 3$ a podívám se na hodnotu uloženou v poli pod indexem 83. Tabulky s predikcemi parit bylo třeba naplnit hodnotami, které je sice snadné spočítat, ale abych si usnadnil práci a ušetřil čas, vytvořil jsem si na generování tabulek program v jazyku C++. Na začátku programu „parita.cpp“ je deklarováno pole `sBoxOut`, které obsahuje vypočítané výstupy z modulu *SBox* pro všechny možné vstupy. Jeho hodnoty jsou dobře známé, já jsem je zkopíroval z [9]. Poté jsem vytvořil pole o velikosti 256 prvků, do kterého jsem uložil paritní hodnoty podle hodnot v `sBoxOut`. Hodnoty z pole jsem vytiskl ve vhodném formátu a měl jsem hotovou tabulku pro předpověď parity výstupu. Obdobně jsem vytvořil i tabulku pro předpověď parity vstupu. Nejprve jsem vypočítal paritu z hodnot 1 – 255 a výslednou tabulku jsem pak seřadil podle klíčů v poli `sBoxOut`, aby fungovala jako asociativní. Program „parita.cpp“ byl zkompilován na operačním systému Windows 8.1, je spustitelný z příkazového řádku, kam také tiskne tabulky s paritou. Tabulky lze snadno dostat do textového souboru pomocí operátoru přesměrování „»“ v příkazovém řádku.

3.4 Srovnání implementací

Po tom, co jsem naprogramoval všechny varianty algoritmu AES, zasadil jsem každou z variant do připraveného projektu a spustil jsem syntézu. Syntéza je transformace abstraktního popisu hardwaru na reálné prvky, které existují na cílové platformě. Roli syntézy v procesu návrhu hardware znázorňuje obrázek 3.4. Po provedení syntézy je možné napojit porty top entity na porty konkrétního FPGA. K tomu jsem použil program Quartus II [24], který provedl také rozmístění a propojení entit v přípravku. Quartus po syntéze provádí časovou analýzu, která uvádí délku kritické cesty a nejvyšší možnou frekvenci hodin.



Obrázek 3.4: Role syntézy v procesu návrhu hardware

K syntéze jsem použil FPGA Cyclone III: EPC25F256C8, která je dostatečně velká, aby pojala i plošně náročnou variantu AES02. Tabulka 3.2 obsahuje důležité výsledky syntézy. Zajímal mě počet logických elementů[13], které daná varianta na desce obsadila (sloupec 1), počet obsazených registrů (sloupec 2) a maximální dosažitelná frekvence hodin (sloupec 3). Projekt, jehož součástí jsou i top entity AES, obsahuje kromě realizace AES i další komponenty, které sice nejsou tak velké a složité jako AES, ale jsou dostatečné na to, aby při měření způsobily drobné zkreslení. Tabulka 3.2 například uvádí, že varianta `AES_1Aa` obsadila 16 311 LE, zatímco `AES_01` 6 014 LE. Podle kapitoly 2.2.2 mají být ve variantě `AES_1Aa` třikrát vedle sebe stejné obvody, které jsou ve verzi `AES_01`, což také platí. Samotný `AES_01` obsadil 4 977 LE a samotný `AES_1Aa` zaujal 15 102 LE, zbývajících přibližně tisíc logických elementů je u obou variant obsazeno obvody, které neprovádí šifrování.

Tabulka 3.2: Výsledky syntézy všech variant AES

varianta	*celkem LE*$^{(1)}$	*registry*$^{(2)}$	*frekvence*$^{(3)}$
AES_01	6 041	1 208	60,29
AES_1p	6 046	1 208	64,73
AES_1Ta	6 516	1 594	61,40
AES_1Tr	6 612	1 594	63,32
AES_1Ar	16 236	2 106	56,19
AES_1Aa	16 311	2 128	58,56
AES_02	35 897	747	14,60

Poznámky:
(1) počet obsazených logických elementů (LE)
(2) počet obsazených registrů
(3) maximální dosažitelná frekvence hodin v MHz

[13]Logický element (LE) je nejmenší logická jednotka, pomocí které je realizována uvnitř FPGA libovolná čtyřvstupová logická funkce nebo registr [31].

Tabulka 3.3 srovnává všechny varianty na základě počtu obsazených logických elementů uvnitř FPGA a na základě maximální hodinové frekvence. Pátý sloupec udává, kolik ns je třeba k zašifrování jednoho bloku dat (16 B). Vzhledem k tomu, že `AES_01` je referenční varianta, ze které jsou všechny ostatní odvozené, potvrdilo se, že je prostorově nejúspornější. Sloupce 2 a 6 udávají procentuální rozdíl od referenční varianty v počtu obsazených LE a v čase potřebném pro zašifrování jednoho bloku. Ve čtvrtém sloupci je počet hodinových taktů potřebných k zašifrování jednoho bloku.

Za překvapivé považuji výsledky verze `AES_02`. Ukázalo se, že je o 59 % rychlejší oproti referenční verzi a přitom jen asi 6krát prostorově objemnější. Vzhledem k tomu, že obsahuje téměř 10krát více kombinační logiky, je úspora značná. Způsobena je optimalizacemi, které provádí syntézní nástroj. Zajímavý je také výsledek `AES_1p` – při téměř stejném počtu logických elementů je varianta s paritou o 7 % rychlejší než `AES_01`. Mezi všemi spolehlivostními variantami je jednoznačně nejlepší. U bezpečných verzí s prostorovou redundancí je daní za odolnost proti poruchám vysoká spotřeba prostoru – tedy logických elementů. Ta je v případě `AES_1Aa` až o 170 % vyšší než má `AES_01`. Bezpečné varianty s časovou redundancí jsou obě téměř 3krát pomalejší, prostoru zabírají téměř stejně. Lépe vychází z porovnání verze `AES_1Tr`, která potřebuje k zašifrování bloku dat 474 ns, zatímco verze `AES_1Ta` celých 489 ns.

Tabulka 3.3: Srovnání variant AES z hlediska plochy a času

varianta	$LE^{(1)}$	*změna* $LE^{(2)}$	$f^{(3)}$	$TkZ^{(4)}$	$ČZ^{(5)}$	$+/-$ $ČZ^{(6)}$
AES_01	6 041	100 %	60,29	10	166	100 %
AES_1p	6 046	0 %	64,73	10	154	- 7 %
AES_1Ta	6 516	+ 8 %	61,40	30	489	+ 195 %
AES_1Tr	6 612	+ 9 %	63,32	30	474	+ 186 %
AES_1Ar	16 236	+ 169 %	56,19	10	178	+ 7 %
AES_1Aa	16 311	+ 170 %	58,56	10	171	+ 3 %
AES_02	35 897	+ 494 %	14,60	1	68	- 59 %

Poznámky:
(1) počet obsazených logických elementů (LE)
(2) procentuální rozdíl v počtu LE oproti AES_01
(3) maximální dosažitelná frekvence hodin v MHz
(4) počet hodinových taktů potřebných k zašifrování 1 bloku
(5) čas potřebný k zašifrování 1 bloku v ns
(6) procentuální rozdíl v času k zašifrování oproti AES_01

Výsledky srovnání potvrdily, že nejrychlejší nezabezpečená verze je `AES_02`. Při frekvenci 14,6 MHz je možné šifrovat data rychlostí 1,87 Gb/s. Nejrychlejší zabezpečenou verzí AES je `AES_1p`, která šifruje rychlostí 828 Mb/s.

3.4.1 Jak získat správná data

O způsobu, kterým budu provádět měření jednotlivých implementací, jsem měl jasno od doby, kdy jsem práci začal psát. Později se ale ukázalo, že plány může zhatit jedna nezaškrtnutá položka v nastavení. Na poslední chvíli přišel přestup ke konkurenci, což se ve finále ukázalo jako správný tah.

Původně jsem chtěl k syntéze použít software ISE Design Suite [32], jehož ovládání mi bylo dobře známé z předchozího studia. Vytvořil jsem tedy pro AES projekt, vložil do něj zdrojové soubory a nastavil jsem „od boku“ typ FPGA, do kterého se bude návrh generovat. Šlo o první pokus a musel jsem nějak začít, proto jsem typ FPGA nastavoval od oka. Syntéza verze `AES01` proběhla během 5 minut (to je dobrý čas) a poskytla ve výpisu mnoho údajů k analýze. Mimo jiné i pro mě stěžejní data o zabrané ploše zařízení a o kritické cestě. ISE sice nedělí čip na logické elementy (LE), ale pro jednoduchost tyto části budu označovat stejně. První varianta algoritmu AES tedy obsadila na vybraném FPGA 193 % logických elementů. Bez ohledu na to, že jde o nesmysl, pokračoval jsem v měření. Pro mě nebyla podstatná absolutní čísla, ale vzájemné porovnání variant. Do stejného procesu jsem pustil variantu `AES_1Aa`. Vzhledem k tomu, že jde o redundantní realizaci, kde je celý algoritmus 3krát zduplikován, očekával jsem přibližně 3krát vyšší obsazení FPGA. Jenže obsazení logických elementů se opět pohybovalo mezi 190 – 200 %, což bylo zcela jistě špatně.

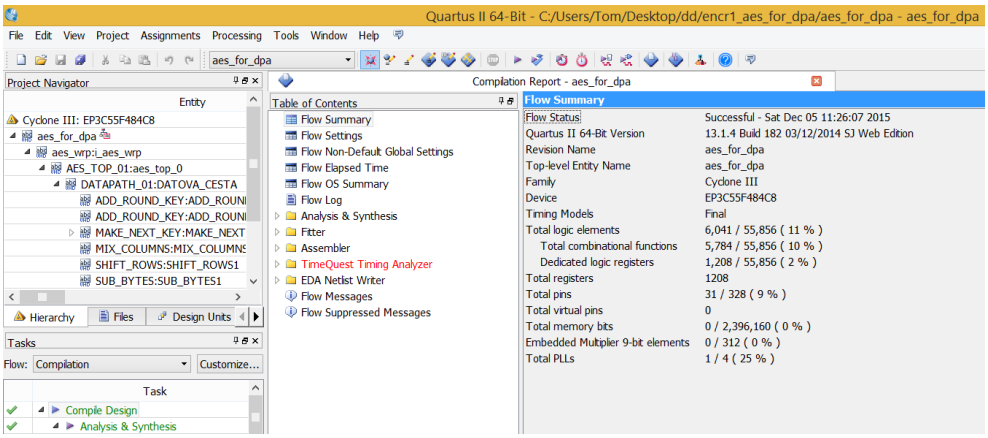
Na vině byla optimalizace redundantní logiky, kterou syntézní nástroj v ISE implicitně provádí. Vypnutí optimalizace mělo být údajně někde v nastavení syntézy, jenže se mi ho nepodařilo najít. Ani nikdo jiný, koho jsem se na problém ptal, neznal řešení. Proto jsem dostal od vedoucího práce tip na jiný syntézní nástroj – Quartus [24]. Verze pro studijní účely je po krátké registraci k dispozici zdarma. Instalace proběhla během 20 minut, což je v porovnání s ISE dobrá hodnota. Po prvním spuštění jsem byl příjemně překvapen. Grafické rozhraní mi přišlo přehledné a i po čase, když jsem se naučil základní operace, musím potvrdit, že se mi s Quartusem pracovalo o něco příjemněji než s ISE. K měření jsem dostal od vedoucího ukázkový projekt i s instrukcemi, jak mám postupovat při dalších variantách AES. Projekt byl nakonfigurován pro FPGA *Cyclone III: EP3C25F256C8*. Samozřejmě také Quartus při syntéze optimalizuje, a tak bylo nutné provést v nastavení několik úprav. Nastavení syntézy, která se liší od implicitních, uvádí tabulka 3.4.

Pro méně prostorově náročné varianty AES včetně `AES_1Ar` a `AES_1Aa` proběhla syntéza úspěšně. Když jsem ale spustil syntézu pro variantu `AES02` (AES jako kombinační logika), výsledkem byla po 72 minutách chyba. Quartus zjistil, že se tolik logiky do tohoto FPGA zkrátka nevejde. Naštěstí jsem byl

Tabulka 3.4: Nastavení syntézy v software Quartus

nastavení	*použitá hodnota*
Remove duplicate registers	off
Remove redundant logic	off

z neúplné zprávy schopen vyčíst odhad počtu LE. Poté jsem vybral větší zařízení (*Cyclone III: EP3C55F484C8*) a spustil jsem syntézu znovu. Druhý pokus se po 92 minutách vydařil. Vzhledem ke změně zařízení bylo třeba všechny ostatní již odměřené verze znovu nechat projít syntézou s novým nastavením. Na obrázku 3.5 je vidět, jak vypadá zpráva o syntéze v software Quartus, odkud jsem čerpal potřebná data.



Obrázek 3.5: Ukázka výsledků syntézy v nástroji Quartus

Ovšem nebyla to jediná překážka v měření. Uživatelům, kteří si nepřiplatí za prémiovou verzi (tedy i mně), provádí syntézu projektu maximálně jeden procesor. To je velmi málo. Jeden průběh trval průměrně 3 – 5 minut. V extrémním případě u varianty `AES02` trval 92 minut. Syntézu bylo třeba spustit pro 7 verzí, navíc než jsem vyladil nastavení do finálního stavu, provedl jsem nejméně 5 – 10 dalších syntéz. Abych urychlil práci, vytvořil jsem si několik kopií projektu, spustil jsem Quartus ve 4 instancích a v každém z nich jsem zároveň prováděl syntézu jiné varianty AES. I s tímto zlepšovákem trvalo měření odhadem více než 5 hodin.

Také jsem nevěděl, kde ve výsledcích syntézy najít čas pro kritickou cestu nebo alespoň maximální možnou hodinovou frekvenci. Quartus provádí komplexní časovou analýzu, jejíž výstupem je více než 20 podrobných tabulek, se kterými jsem se doposud nesetkal. Po delším pátrání jsem nakonec objevil tabulku, která obsahovala položku „maximální dosažitelná frekvence“. Popis k tabulce chyběl, nicméně hodnota v ní uvedená přibližně odpovídala té, kterou ještě při jednom z prvních měření ukázal ISE. Proto jsem se s ní spokojil.

Kapitola 4

Testování

Cílem této kapitoly je ukázat, jakým způsobem jsem implementované varianty testoval a jak testy dopadly. Text seznámí čtenáře s obecným postupem při simulaci hardwaru i s konkrétními výsledky testů, které jsem provedl. Testování jsem uskutečnil formou simulace v ModelSim SE pomocí testovacích souborů v jazyku VHDL. Pro testování jsem používal dvě testovací sady – vlastní a dodanou od vedoucího práce.

V první úrovni jsem testoval operace uvnitř rundy – *AddRoundKey*, *MixColumns*, *ShiftRows* a *SubBytes* pomocí dodaných testů. Tyto ověřené moduly jsem poté použil při tvorbě téměř všech variant šifry AES. O simulaci základních modulů pojednává sekce 4.2. Ve druhé úrovni jsem testoval větší moduly složené z více komponent a celé top entity variant AES pomocí vlastních testů. Podrobnosti k těmto testům se nachází v sekci 4.3.

4.1 Obecný postup při simulaci

Ještě před tím, než přejdu ke konkrétním testům a jejich výsledkům, stručně vysvětlím, jakým způsobem probíhá testování hardwarových komponent. Než popis hardwarových modulů (v mém případě v jazyku VHDL) projde syntézou, provádí se simulace, která testuje, jestli modul provádí to, co má. K tomu slouží simulační nástroje. Já jsem prováděl simulaci v programu ModelSim SE 10.0c [25]. ModelSim provádí simulace číslicových obvodů popsaných v jazyku VHDL. Na základě předloženého testovacího souboru (testbenche) otestuje aplikaci proti zadaným (referenčním) hodnotám v zadaném simulačním čase. Cílem této kapitoly je proto vysvětlení postupů při simulacích v programu ModelSim.

Simulaci jsem prováděl pomocí skriptů[14] s příponou „.do“. Výstupem simulace je grafický průběh signálů a textový záznam v konzoli o jejím průběhu.

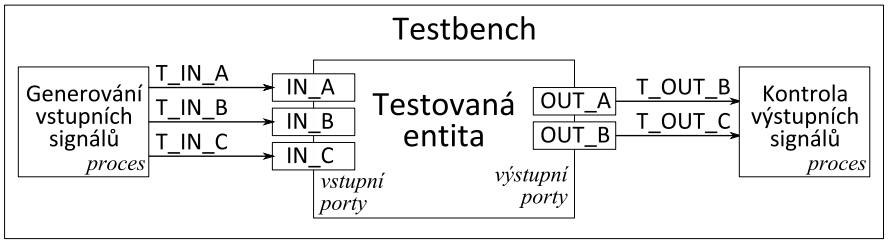
[14]Skript je sada příkazů uložená v souboru. Po spuštění skriptu vykoná program všechny příkazy v něm uvedené.

K testu hardwarového modulu `X` je třeba mít:

- Zdrojový kód modulu `X` v jazyku VHDL.
- Testovací soubor, který definuje průběh testu, vyvolává podněty a kontroluje reakce testovaného objektu na ně. Nazývá se *testbench* a je psán stejně jako zdrojové kódy k implementaci v jazyku VHDL [33].
- Skript, který zajistí spuštění všech náležitostí nezbytných k provedení testu.

4.1.1 Jak funguje testbench

Testbench je soubor s testovací entitou. Stejně jako standardní entity je popsán v jazyku VHDL. Na rozdíl od běžné entity nemá žádné vstupní ani výstupní porty, ale má jen vnitřní signály, které se napojí na vstupní a výstupní porty testovaného modulu. Napojení testbenche na entitu, která je testována, ukazuje obrázek 4.1. Testbench generuje na vstupy testované entity hodnoty a kontroluje jeho výstupní porty, které porovnává vůči referenčním hodnotám.



Obrázek 4.1: Testbench – princip fungování

Kontrolu správných hodnot provádí příkaz `assert`. Jeho parametry jsou:

- *porovnání*, které se dá vyhodnotit jako `TRUE` nebo `FALSE`,
- *textový výstup* v případě, že porovnání je vyhodnoceno jako `FALSE` a
- *závažnost* chyby v případě, že porovnání je vyhodnoceno jako `FALSE`.

Závažnost určuje, co se stane se simulací v případě, že je detekována chyba. Existují volby od nejjemnější (výpis informace do konzole) až po nejtvrdší, kdy je přerušena simulace. Následující úsek kódu předvádí kontrolu pomocí příkazu `assert`. Příkaz kontroluje, jestli signál `DATA` má hodnotu 01_{hex}, pokud by tuto hodnotu neměl, vypíše se hlášení a přeruší se simulace.

```
assert DATA = x"01"
  report "Na signalu DATA je chyba"
  severity FAILURE; -- prerusi simulaci
```

4.1.2 Prováděcí skript

Prováděcí skript je sada příkazů uložená v souboru s příponou „.do“. Slouží k tomu, abych nemusel při každém spuštění programu ModelSim znovu nastavovat prostředí a opakovaně provádět rutinní činnost. Po spuštění skriptu vykoná ModelSim všechny příkazy v něm uvedené.

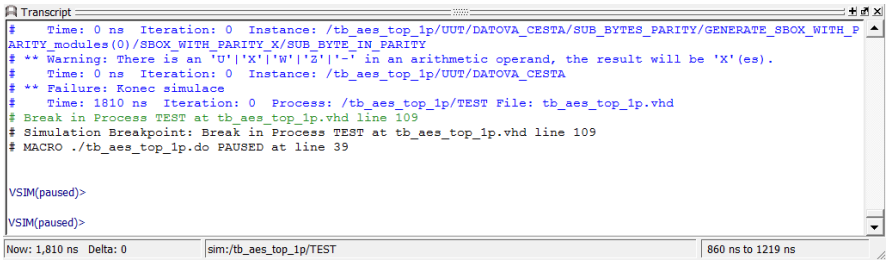
Nejčastěji jsem od prováděcího skriptu vyžadoval, aby prováděl překlad zdrojových souborů, spustil simulaci a vykreslil průběhy signálů v požadované podobě. Akce, které spouští prováděcí skript a k nim odpovídající příkazy, popisuje tabulka 4.1. Konkrétní příklad v tabulce překládá testbench `aes_top_1Ta.vhd`, na jehož základě provádí ModelSim simulaci. Vykreslovat se bude signál `T_CLK`, jeho hodnota bude interpretována v desítkové soustavě. Signál `T_CLK` je deklarován v entitě `tb_aes_top_1Ta`.

Tabulka 4.1: ModelSim – akce spouštěné prováděcím skriptem

Akce	*Příkaz*
překlad zdrojových kódů	`vcom -93 ../aes_top_1Ta.vhd`
spuštění simulace	`vsim tb_aes_top_1Ta`
vykreslení průběhu signálu	`add wave -label T_CLK -radix decimal sim:/tb_aes_top_1Ta/T_CLK`

4.1.3 Postup při simulaci v programu ModelSim

Než uvedu postup, seznámím čtenáře nejprve s prostředím v programu ModelSim. To se skládá z mnoha oken, které je možné zobrazit/skrýt podle vůle uživatele. Nejdůležitějšími pro mojí práci jsou okna *Transcript* (obrázek 4.2) a *Wave* (obrázek 4.3). Okno *Transcript* slouží jako příkazový řádek. Zde mohu zadávat příkazy, odtud spouštím prováděcí skript a sem se vypisují případné hlášení a varování [25]. Do okna *Wave* se při simulaci vykreslují průběhy signálů. Uživatel, který testuje, má díky tomu vizuální přehled o tom, jak testovaná entita funguje.



Obrázek 4.2: Okno Transcript v programu ModelSim SE

Užitečnou funkci skrývá také tlačítko *Compile* na nástrojové liště, pomocí kterého je možné provádět kompilaci jednotlivých VHDL zdrojových kódů [34]. Tato funkce mi byla užitečná především ve fázi vývoje, kdy jsem kontroloval, zda nemám v kódu překlepy.

Jak jsem postupoval při testování každého modulu, popisuje krok po kroku následující seznam.

1. Podle modulu, který měl být testován, jsem vytvořil vhodný testbench.

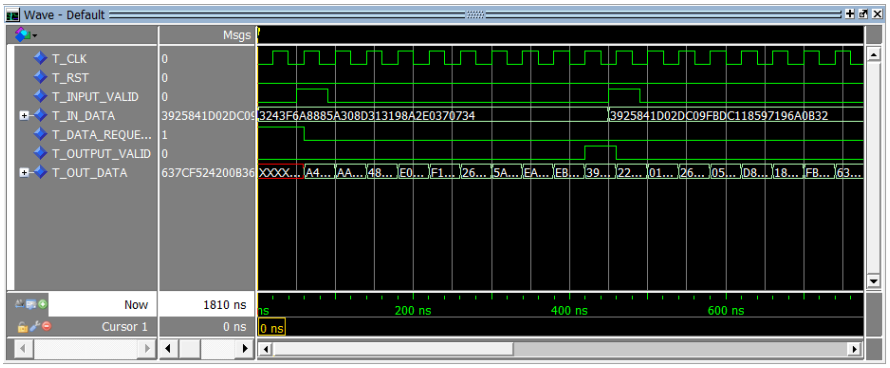
2. Vytvořil jsem skript, který provádí především kompilaci modulu a jeho testbenche, spouští simulaci a vykresluje průběhy signálů.

3. Spustil jsem program ModelSim a přesunul jsem se do okna Transcript, které je na obrázku 4.2. V konzoli jsem se pomocí příkazů přesunul do adresáře se skriptem `test.do` a příkazem `do test.do` jsem spustil simulaci.

4. Během několika sekund se provedla simulace, na obrazovku se do okna Wave (obrázek 4.3) vykreslily průběhy signálů a do okna konzole se vypsalo hlášení.

5. Podle hlášení jsem případně nastavoval kurzor na správnou hodnotu simulačního času a pozoroval jsem, zda-li se výstup mění podle vstupu tak, jak má.

6. V případě, že jsem byl se simulací spokojen, ukončil jsem ji. V opačném případě, jsem se vrátil buď do fáze implementace nebo jsem vhodně upravil testbench. Občas se stalo, že odhalená chyba nebyla ve skutečnosti chyba, ale chybně napsaný testbench.



Obrázek 4.3: Okno Wave v programu ModelSim SE

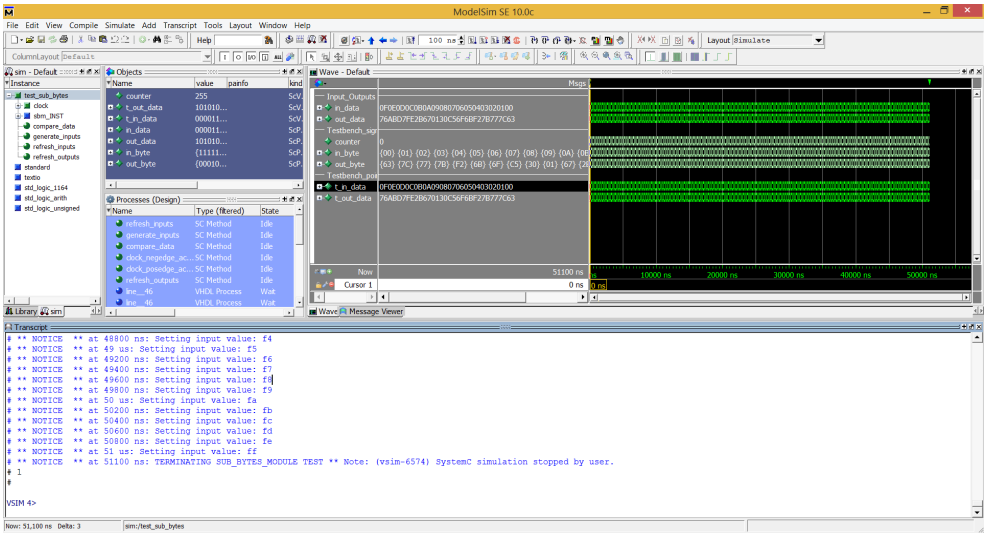
4.2 Testování základních modulů

Moduly *SubBytes*, *ShiftRows*, *MixColumns* a *AddRoundKey* jsem testoval pomocí testů dodaných vedoucím práce. Obdržel jsem sadu testbenchů v jazyku SystemC i s prováděcími skripty. Tyto testbenche generují všechny možné kombinace vstupů a testují tedy výše zmíněné moduly ve všech situacích, které mohou nastat. Výstupy jsou počítány uvnitř testbenchů vlastním algoritmem.

S jazykem SystemC jsem se do doby psaní práce nesetkal a testbenche v něm napsané se značně vymykaly představě, kterou jsem o nich do té doby měl. Zpočátku jsem se v něm vůbec neorientoval, avšak po instruktáži od vedoucího práce jsem byl schopen alespoň porozumět kódu. Také mi pomohlo, že syntaxe vychází z jazyka C, který ovládám. To pro účely mojí práce stačilo, dále jsem se jazyku SystemC nevěnoval.

4.2.1 Výsledky testů

V době odevzdání práce prokázaly testy bezchybnou funkčnost všech základních modulů. Na obrázku 4.4 přikládám výsledek simulace, která kontroluje modul *SubBytes*. V okně Wave jsou průběhy úmyslně zmenšeny tak, aby se do okna vešel celý simulační čas. Je tedy vidět, že v celém průběhu simulace nebyly indikovány žádné chyby.



Obrázek 4.4: Výsledek simulace modulu SubBytes

Do doby, než testy ukázaly správnost, jsem se potýkal s mnoha problémy. Následující seznam uvádí, kde se vyskytovaly největší chyby a jak jsem je řešil.

- V modulu *ShiftRows* bylo nesprávné (opačné) zapojení s ohledem na LSB a MSB[15]. Podrobněji je problém rozveden v textu na konci sekce 3.2.2. Simulace ukazovala, že není správně žádný výstup, po opravě prošel modul simulací úspěšně.
- U modulu *MixColumns* ukazovala simulace, že jsou všechny výstupy špatně. Příčinou bylo stejně jako u *ShiftRows* nesprávné uspořádání bytů ve vektorech, které vstupují do násobení maticí. Více je o problému uvedeno na konci sekce 3.2.2.

4.3 Testování vyšších modulů

Protože způsob, jakým budu testovat, záležel na mně, vybral jsem si pro testování vyšších modulů jazyk VHDL. S jazykem VHDL mám zkušenosti, již jsem pomocí něho testoval několik projektů, volba byla proto jasná.

Struktura testbenchů je ve srovnání s testy nižších modulů odlišná. Testbenche nejsou tak důkladné, obvykle generují pouze 2 – 3 sady vstupních hodnot, pro které kontrolují výstupy. Tento postup si mohu dovolit, protože vyšší celky jsou složené z modulů, které jsem dříve otestoval a o kterých jsem si jist, že fungují dobře. Vyšší celky jako datová cesta už jsou jen propojením celků nižších. Pokud tedy test ukáže u 2 – 3 vstupů správné výstupy, nemůže se jednat o náhodu a správná funkčnost modulu je tím doložena.

Obrázek 4.5: Ukázka počítačového programu Rijndael Inspector

K získání referenčních hodnot jsem použil program Rijndael Inspector [26]. Ten slouží k zašifrování nebo dešifrování 16bytových dat. Zároveň ukazuje mezivýsledky šifrování/dešifrování nejen po každé rundě, ale dokonce po každé

[15]LSB – nejméně významný byte, MSB – nejvíce významný byte

operaci, a to v přehledné maticové formě. Umožňuje šifrovat libovolná data, která uživatel zadá, ale má v sobě uložené i dva bloky testovacích dat, jež jsou dostupná jedním kliknutím. Právě tato testovací data obsahují moje testbenche, jejichž referenční hodnoty jsou získány programem Rijndael Inspector. Na obrázku 4.5 je k vidění prostředí programu Rijndael Inspector.

4.3.1 Testování variant AES na 10 cyklů

Pro všechny varianty AES, které splňují podmínku, že šifrují jeden blok dat po dobu 10 cyklů, jsem použil stejný test top entity. Nejvyšší entity variant `AES01`, `AES_1p`, `AES_1Aa` a `AES_1Ar` jsem testoval testbenchem `TB_AES_TOP_X`, kde `X` je identifikátor varianty, který se shoduje s příponou v názvu varianty. Následující odstavce detailně popisují, co a jak testbench testuje.

Do testbenche je jako komponenta zapojen modul `AES_TOP_X`, kde Signály, které nastavuji a jejichž hodnoty testuji, jsou napojeny na piny modulu `AES_TOP_X` a jmenují se téměř stejně. Jediným odlišením je, že všechny začínají předponou „T_“. V architektuře jsou 2 procesy. Jeden generuje periodický hodinový signál, ve druhém procesu provádím nastavení vstupních signálů a testuji výstupy. Celkem obsahuje druhý proces 3 testy.

Na počátku simulačního času nastavuji signály následovně.

- `T_INPUT_VALID <= '0';`
- `T_RST <= '0';`
- `T_IN_DATA <= x"3243f6a8885a308d313198a2e0370734";`

Vstupní data odpovídají testovacímu vektoru v aplikaci Rijnadael Inspector. Po nastavení signálů nastavím na jeden hodinový takt signál indikující, že jsou data platná. Pak čekám po dobu 10 hodinových cyklů a kontroluji, že výstupy jsou následující.

- `T_OUT_DATA = x"3925841d02dc09fbdc118597196a0b32"`
- `T_DATA_REQUEST = '0'`
- `T_BUSY = '1'`
- `T_OUTPUT_VALID = '1'`

Poté opakuji stejný průběh s jinými vstupními daty. Třetí test kontroluje správnou reakci na reset a naběhnutí výpočtu po resetu. Na začátku nastavuji hodnoty jako při předchozích testech, avšak po 5 hodinových taktech, kdy je šifrování přibližně v polovině, nastavuji signál `T_RST` na `'1'`. Ihned poté nastavuji:

- `T_INPUT_VALID <= '1';`
- `T_IN_DATA     <= x"3925841d02dc09fbdc118597196a0b32";`

a po 10 hodinových cyklech testuji výstupy:

- `T_OUT_DATA     = x"7dfdff39cc79c14315baf5ef727cc0cf"`
- `T_DATA_REQUEST = '0'`
- `T_BUSY         = '1'`
- `T_OUTPUT_VALID = '1'.`

Po proběhnutí posledního třetího testu přeruším provádění simulace příkazem `assert FALSE severity FAILURE;`. Po přerušení simulace ModelSim zobrazí průběhy signálů a varování. Pokud bych simulaci nepřerušil, proces s testem by běžel periodicky a nikdy by neskončil.

Simulace potvrdila, že na všech výstupních signálech uvedených výše jsou požadované hodnoty a tedy že všechny top entity fungují správně.

V testu modulu `AES_1p` simulace ukázala, že v modulu `SBOX_WITH_PARITY` se na signálu `WARN` vyskytují zákmity. Signál `WARN` informuje, pokud dojde k případné poruše. Zákmity jsou způsobeny zpožděním v kombinační logice, která je zachycena na obrázku 2.10 a jejich výskyt neindikuje chybu v návrhu. Přesto jsem se je pokusil odstranit. Entitě `SBOX_WITH_PARITY` jsem vytvořil druhou architekturu, ve které jsem veškerou kombinační logiku, která vyhodnocuje porovnání parit, implementoval v jednom procesu, přičemž místo signálů jsem použil proměnné. Přesto byly zákmity na signálu `WARN` při simulaci stále patrné. Nicméně pro účely této práce zůstává tento signál dále nenapojen (nevyužit) a není nikde čten, proto jsem tento pseudoproblém dále neřešil.

4.3.2 Testování varianty AES02

Datová cesta verze `AES_02` se skládá z modulů `ONE_ROUND` a `LAST_ROUND`. Pro každý z nich jsem vytvořil testbench. Testbench `TB_ONE_ROUND` obsahuje uvnitř architektury jediný testovací proces, v němž jsou dva testy. První z nich nastavuje vstupy následovně:

- `T_DATA_IN <= x"193de3bea0f4e22b9ac68d2ae9f84808";`
- `T_KEY_IN  <= x"2b7e151628aed2a6abf7158809cf4f3c";`
- `T_N_IN    <= x"0";`,

a očekává výstupy

- `T_DATA_OUT = x"a49c7ff2689f352b6b5bea43026a5049"`
- `T_KEY_OUT  = x"a0fafe1788542cb123a339392a6c7605"`
- `T_N_OUT    = x"1".`

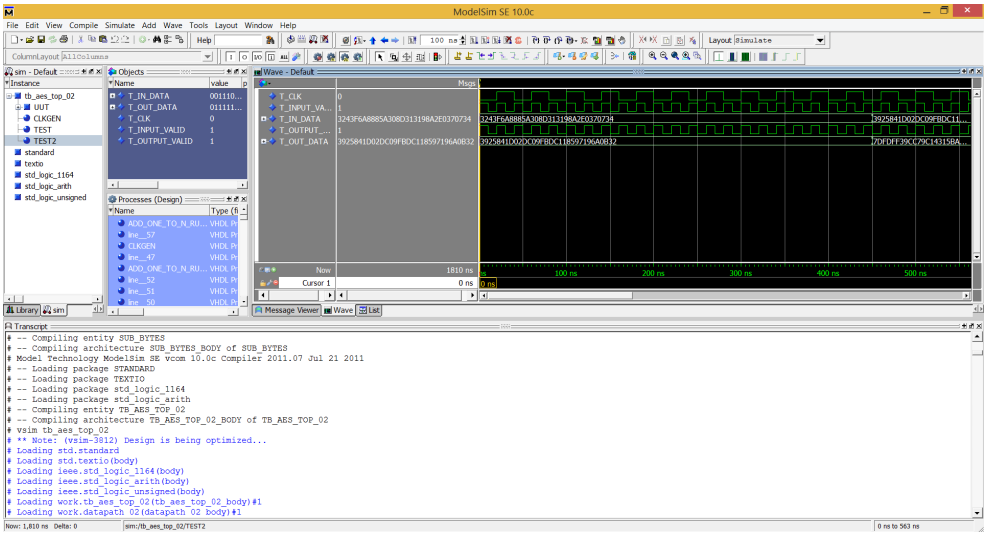
Druhý test přivádí na vstup rundy klíč a data, která vystupují z 1. testu. Na výstupu očekává následující hodnoty.

- `T_DATA_OUT = x"aa8f5f0361dde3ef82d24ad26832469a"`
- `T_KEY_OUT  = x"f2c295f27a96b9435935807a7359f67f"`
- `T_N_OUT    = x"2"`.

Testbench `TB_LAST_ROUND` obsahuje uvnitř architektury rovněž jediný testovací proces. V něm jsou dva nezávislé testy. První test nastavuje 16bytový klíč, data a číslo rundy. Protože jde o poslední rundu, vstupující číslo rundy musí být 9. Na výstupu jsou očekávána zašifrovaná data (po 10. rundě). Druhý test provádí to samé, jen s jinými daty.

Nejvyšší entitu `AES_TOP_02` kontroluje testbench, který je velice podobný testům nejvyšších entit u dalších verzí AES. Architektura testbenche obsahuje tři procesy. Kromě procesu generujícího hodiny se zde nachází proces, který nastavuje pouze datový vstup a kontroluje datový výstup. Hodnoty přivedené na datový vstup jsou opět totožné s těmi, které jsou v testech top entit ostatních variant AES. Poslední proces generuje periodicky signál `INPUT_VALID` a kontroluje související signál `OUTPUT_VALID`, který musí být vždy shodný. Výsledek testu entity `AES_TOP_02` je na obrázku 4.6. V okně Wave je dobře vidět, že vstupní data jsou zašifrována ihned.

Všechny testy dopadly dobře a ověřily, že jak komponenty `ONE_ROUND` a `LAST_ROUND`, tak top entita `AES_TOP_02` fungují podle požadavků.



Obrázek 4.6: Výsledek testu top entity varianty AES02

4.3.3 Testování variant AES na 30 cyklů

Testy pro top entity variant `AES_1Tr` a `AES_1Ta` se od testů pro verze na 10 cyklů téměř neliší. Zásadním rozdílem je, že v testech variant s časovou redundancí se na výsledky čeká 3krát déle – tedy 30 taktů. Tím ale veškeré odlišnosti končí a zbytek testbenche je stejný. Simulace potvrdila, že testované varianty fungují správně podle specifikace.

Architektura testbenche obsahuje dva procesy – jeden generuje průběh hodinového signálu, druhý provádí testy. Uvnitř 2. procesu se provádí čtyři testy. První dva testy kontrolují správné zašifrování dat. Tabulka 4.2 udává očekávané výstupy na zadané vstupy. Třetí test simuluje sepnutí resetu a čtvrtý kontroluje správné naběhnutí po resetu.

Tabulka 4.2: Testovací hodnoty v testech AES_1Tr a AES_1Ta

generované vstupy	*očekávané výstupy po 30 taktech*

`T_IN_DATA`
`  <= x"3243f6a8885a308d`
`  313198a2e0370734";`
`T_INPUT_VALID <= '1';`
`T_RST  <= '0';`
	`T_OUT_DATA`
	`  = x"3925841d02dc09fb`
	`  dc118597196a0b32"`
	`T_DATA_REQUEST = '0'`
	`T_BUSY  = '1'`
	`T_OUTPUT_VALID = '1'`

`T_IN_DATA`
`  <= x"3925841d02dc09fb`
`  dc118597196a0b32";`
`T_INPUT_VALID <= '1';`
`T_RST  <= '0';`
	`T_OUT_DATA`
	`  = x"7dfdff39cc79c143`
	`  15baf5ef727cc0cf"`
	`T_DATA_REQUEST = '0'`
	`T_BUSY  = '1'`
	`T_OUTPUT_VALID = '1'`

4.3.4 Shrnutí testů

Všechny mnou vytvořené testy i testy dodané vedoucím práce dopadly úspěšně. Ať již na úrovni jednotlivých modulů, tak i na vyšších úrovních (runda, celý algoritmus). Nejdůkladněji byly testovány nejnižší moduly z důvodu celkové složitosti AES algoritmu. Vyšší moduly nebyly testovány tak detailně, důraz byl kladen především na správné zapojení komponent. Během testování byly zjištěny nesrovnalosti v návrhu (viz MixColumns), které byly následně odstraněny.

Závěr

V práci jsem se zabýval návrhem a realizací algoritmu AES pro FPGA. Cílem práce bylo seznámit se s fungováním algoritmu a se stávajícími řešeními implementace AES pro FPGA. Na základě získaných poznatků jsem měl vytvořit návrh a realizovat několik variant AES algoritmu v jazyku VHDL. Úkolem bylo vytvořené varianty porovnat z hlediska obsazené plochy na čipu a času potřebného k šifrování. Výsledkem práce je sada algoritmů určená k měření spotřeby energie za účelem výzkumu v oblasti tzv. útoků postranními kanály.

Po důkladném seznámení s algoritmem AES jsem na základě zjištěných poznatků vytvořil prvotní návrh, který posloužil jako výchozí bod pro realizaci spolehlivostních variant. Odolnost návrhu jsem zajistil použitím redundance na různých úrovních. Celkem jsem navrhl a vytvořil 5 spolehlivostních návrhů, z nichž dva využívají principu prostorové redundance, u dvou je využito principů časové redundance a u jednoho redundance informační.

Variantu s prostorovou redundancí na úrovni algoritmu jsem realizoval tak, že je možné nastavit libovolné množství šifrovacích jednotek, které provádějí paralelně to samé šifrování. Změnou jediné konstanty lze řídit počet opakování, který je implicitně nastaven na hodnotu 3. Tato hodnota byla použita i při syntéze, ze které následně vzešlo srovnání všech variant.

Spolehlivostní varianty jsou vytvořeny podle specifikace, aby je bylo možné zasadit do projektu připraveného k měření spotřeby. Všechny realizace jsou syntetizovatelné pro dodané FPGA Cyclone III: EPC25F256C8 v software Quartus II a všechny byly verifikovány pomocí nástroje ModelSim SE s použitím zčásti vlastní a zčásti dodané sady testů. Simulace v ModelSim prokázala, že jak nižší, tak vyšší moduly jednotlivých realizací fungují správně.

Nad rámec zadání se mi podařilo vytvořit variantu AES02, která vůbec neobsahuje sekvenční část. Je tvořena jen kombinační logikou a výsledky syntézy ukázaly, že šifrování je oproti referenční variantě o 59 % rychlejší. Takového zlepšení jsem ovšem dosáhl za cenu téměř šestinásobné spotřeby prostoru.

Porovnání vytvořených variant jsem provedl na základě výsledků syntézy v nástroji Quartus. Ukázalo se, že dodaný typ FPGA není dostatečně velký,

aby pojal variantu AES02 s extrémní spotřebou plochy. Proto jsem se rozhodl provést syntézu všech variant ještě jednou, abych mohl vytvořit objektivní srovnání všech variant včetně AES02. K tomu jsem vybral FPGA Cyclone III: EP3C55F484C8, které obsahuje pro moje účely dostatek paměti i logických elementů. Po syntéze každé varianty jsem ve výstupní zprávě sledoval počet obsazených logických elementů, registrů a maximální možnou frekvenci hodin, které je možné dosáhnout. Vytvořil jsem srovnání, kde jsem jako referenční variantu použil první implementovanou – AES01, jejíž výsledky představovaly 100 %.

Potvrdilo se, že spolehlivé varianty, které šifrují třikrát za sebou, jsou skutečně téměř třikrát pomalejší. Na druhou stranu varianty, kde je výpočetní část třikrát replikována, nezaujaly třikrát, ale pouze 2,7ktrát více místa. Jev je způsoben prostorovými optimalizacemi, které provádí syntézní nástroj. Překvapivým vítězem testu se stala spolehlivostní varianta, která využívá k detekci chyb paritu. Nejenže obsadila téměř stejný prostor jako referenční verze, navíc se ukázalo, že je o 7 % rychlejší.

Přestože výsledky práce ukazují, která z variant je v kterém ohledu nejlepší, parametry implementací je jistě možné ještě vylepšit. Z důvodu velkého množství variant nedošlo k optimalizacím, díky kterým by mohlo být šifrování efektivnější. To je směr, kde by se v budoucnu dalo na práci navázat. Vhodné by bylo zaměřit se na jednu až dvě z představených verzí algoritmu a ty optimalizovat za účelem dosažení co nejnižšího času potřebného k šifrování nebo co nejnižšího obsazeného prostoru na FPGA.

Sada algoritmů vytvořená v rámci práce je nyní připravena k měření spotřeby na vývojové desce Evariste II. Jednotlivé varianty AES je možné zasadit do připraveného rozhraní v rámci projektu, jehož cílem je výzkum útoků na různé varianty šifry pomocí tzv. postranních kanálů. Projekt je vhodný jako téma mojí budoucí diplomové práce nebo je k dispozici pro kohokoliv, kdo by měl zájem o výzkum v oblasti útoků na šifru AES.

Literatura

[1] Singh, S.: *Kniha kódů a Šifer: Tajná komunikace od starého Egypta po kvantovou kryptografii*. Praha: Dokořán, Argo, 2009, ISBN 978-80-7363-268-7.

[2] Bitto, O.: Historie kryptologie. [online], Naposledy navštíveno 27. 10. 2015. Dostupné z: `http://www.fi.muni.cz/usr/jkucera/pv109/2003/xbitto.htm`

[3] Lórencz, R.: Rozdělení šifer, proudové šifry, RC4, A5/1 [přednáška]. Technická zpráva, Fakulta informačních technologií, ČVUT, Praha, [2014-03-04].

[4] Güneysu, T.; Kasper, T.; Novotný, M.; aj.: Cryptoanalysis with COPA-COBANA. *IEEE Transactions on Computer*, ročník 57, 2008.

[5] National Institute of Standards and Technology. [online], [cit. 2015-04-18]. Dostupné z: `http://www.nist.gov/`

[6] Přibyl, T.: Historie hackingu: Stručná historie lámání šifer. [online], září 2011, [cit. 2015-10-19]. Dostupné z: `http://computerworld.cz/securityworld/historie-hackingu-strucna-historie-lamani-sifer-48338`

[7] Daemen, J.; Rijmen, V.: *The Design of Rijndael*. listopad 2001, [online], [cit. 2015-04-18]. Dostupné z: `http://jda.noekeon.org/JDA_VRI_Rijndael_2002.pdf`

[8] Nechvatal, J.; Barker, E.; Bassham, L.; aj.: Report on the development of the Advanced Encryption Standard (AES). Technická zpráva, DTIC Document, říjen 2000. Dostupné z: `http://csrc.nist.gov/archive/aes/round2/r2report.pdf`

[9] Paar, C.; Pelzl, J.: *Understanding cryptography.* Heidelberg: Springer, 2010, ISBN 978-3-642-04100-6.

[10] Smékal, D.: Implementation of AES algorithm on FPGA. [online], květen 2015, [cit. 2015-10-27]. Dostupné z: https://dspace.vutbr.cz/bitstream/handle/11012/42973/eeict2015-193-smekal.pdf?sequence=1&isAllowed=y

[11] Lórencz, R.: Blokové šifry, DES, AES [přednáška]. Technická zpráva, Fakulta informačních technologií, ČVUT, Praha, [2014-03-11].

[12] Zabala, E.: Rijndael cipher: Animation using 128 bit size for each the data block and the key. [online animace], [cit. 2015-10-25]. Dostupné z: http://www.formaestudio.com/rijndaelinspector/archivos/Rijndael_Animation_v4_eng.swf

[13] Zabezpečení konfigurace obvodů FPGA. [online], červen 2013, [cit. 2015-10-31]. Dostupné z: http://pandatron.cz/?2744&zabezpeceni_konfigurace_obvodu_fpga

[14] Invea-Tech spouští společně s CESNET projekt na ochranu 400G sítí. [online], duben 2015, [cit. 2015-11-01]. Dostupné z: https://www.cesnet.cz/sdruzeni/zpravy/tiskove-zpravy/invea-tech-spousti-spolecne-s-cesnet-projekt-na-ochranu-400g-siti/

[15] Sedlák, J.: Česi vyvíjejí superrychlou síťovou kartu budoucnosti. [online], červen 2015, [cit. 2015-10-31]. Dostupné z: http://e-svet.e15.cz/it-byznys/cesi-vyvijeji-superrychlou-sitovou-kartu-budoucnosti-1195505

[16] Síťová karta s FPGA Xilinx pro 1 a 10GbE. [online], červen 2013, [cit. 2015-10-31]. Dostupné z: http://pandatron.cz/?2939&sitova_karta_s_fpga_xilinx_pro_1_a_10gbe

[17] FPGA karty. [online], [cit. 2015-11-01]. Dostupné z: https://www.invea.com/cs/produkty-sluzby/fpga-karty

[18] Kotel, S.; Zeghid, M.; Baganne, A.; aj.: FPGA-Based Real-Time Implementation of AES Algorithm for Video Encryption. Technická zpráva, 2014. Dostupné z: http://www.wseas.us/e-library/conferences/2014/Istanbul/TELEDU/TELEDU-03.pdf

[19] Bečvář; Daněk; Schmidt; aj.: Technologie návrhu a realizace číslicových obvodů [přednáška]. Technická zpráva, Fakulta informačních technologií, ČVUT, Praha, [2013-11-19].

[20] Kubátová, H.: Spolehlivost, odolnost proti poruchám [přednáška]. Technická zpráva, Fakulta informačních technologií, ČVUT, Praha, [2015-02-23].

[21] Hlavička, J.; Racek, S.; Golan, P.; aj.: *Číslicové systémy odolné proti poruchám*. Praha: Vydavatelství ČVUT, 1992, ISBN 80-01-00852-5.

[22] Karri, R.; Wu, K.; Mishra, P.; aj.: Concurrent error detection schemes for fault-based side-channel cryptanalysis of symmetric block ciphers. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, ročník 21, 2002: s. 1509–1517.

[23] Natale, G. D.; Flottes, M.-L.; Rouzeyre, B.: A Novel Parity Bit Scheme for SBox in AES Circuits. In *DDECS'07: Design and Diagnostics of Electronic Circuits and Systems*, IEEE, duben 2007, ISBN 1-4244-1161-0, s. 267–271.

[24] Altera Corporation: *Quartus II 64-Bit 13.1.4*. [software]. [přístup 24. listopadu 2015]. Dostupné z: `http://dl.altera.com/13.1/`

[25] Mentor Graphics Corporation: *ModelSim SE 10.0c*. [software]. [přístup 13. listopadu 2015]. Dostupné z: `https://wl.altera.com/download/software/modelsim-starter/10.0`

[26] Zabala, E.: *Rijndael Inspector 1.1*. [software]. [přístup 10. října 2015]. Dostupné z: `http://www.formaestudio.com/rijndaelinspector/`

[27] TeX Users Group: *LaTeX*. [software]. [přístup 15. listopadu 2015]. Dostupné z: `http://www.latex-project.org/ftp.html`

[28] Sublime HQ Pty Ltd: *Sublime Text 2.0*. [software]. [přístup 10. prosince 2015]. Dostupné z: `http://www.sublimetext.com/2`

[29] Free Software Foundation, Inc.: *Inkscape 0.48*. [software]. [přístup 13. září 2015]. Dostupné z: `https://inkscape.org/en/download/`

[30] Bečvář; Daněk; Schmidt; aj.: VHDL II – Automaty, typy, generický popis [přednáška]. Technická zpráva, Fakulta informačních technologií, ČVUT, Praha, [2013-10-15].

[31] Altera Corporation: *Cyclone III Device Handbook*. prosinec 2011. Dostupné z: `https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/cyc3/cyc3_ciii51002.pdf`

[32] Xilinx, Inc.: *ISE Design Suite 14.1*. [software]. [přístup 11. listopadu 2015]. Dostupné z: `http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools/v14_1.html`

[33] Bečvář; Daněk; Schmidt; aj.: Přehled sekvenčních a paralelních příkazů, VHDL pro verifikační prostředí [přednáška]. Technická zpráva, Fakulta informačních technologií, ČVUT, Praha, [2013-10-29].

[34] Pravda, M.: *Úvod do simulačního nástroje ModelSim*. Praha: Fakulta elektrotechnická ČVUT, 2011. Dostupné z: `http://data.cedupoint.cz/oppa_e-learning/2_KME/041.pdf`

Příloha **A**

Seznam pojmů a zkratek

AES Advanced Encryption standard
šifrovací algoritmus

ASIC Application-Specific Integrated Circuits
integrovaný obvod vyráběný pro určitou aplikaci

difúze
rozptyl jednotlivých částí šifrového textu po celé šíři textu

DVI Digital Visual Interface
konektor pro přenos multimediálního obsahu mezi počítačem a obrazovkou
umožňuje přenos obrazových dat

FPGA Field Programmable Gate Array
programovatelné hradlové pole

FT systém Fault Tolerant systém
systém odolný proti poruchám (fyzickým defektům v hardware)

majorita
funkce, která z libovolného lichého počtu jednobitových vstupů vytvoří jeden jednobitový výstup

Mealy označení pro typ konečného automatu
konečný automat je typu Mealy, pokud je výstup závislý na stavu, ve kterém se automat nachází, a na vstupních hodnotách

Moore označení pro typ konečného automatu
konečný automat je typu Moore, pokud je výstup závislý pouze na stavu, ve kterém se automat nachází

NIST National Institute of Standards and Technology
americký Národní úřad pro standardizaci a technologie

otevřený text
text určený k zašifrování
obvykle převeden do číselné reprezentace s použitím kódování

parita
parita je funkce, která sečte všechny jedničky na vstupu a přidá jeden bit s hodnotou 1 nebo 0 tak, aby součet byl vždy sudý/lichý

pipeline zřetězení
pojem vystihující stav, kdy je více za sebou následujících operací rozděleno na menší suboperace, které jsou zpracovávány v samostatných jednotkách
při uvolnění 1. jednotky může jednotka ihned provádět ekvivalentní suboperaci z další operace, díky tomu je zpracování téměř n-násobně zrychleno, kde n odpovídá počtu jednotek

skript
sada příkazů uložená v souboru
po spuštění skriptu vykoná program všechny příkazy v něm uvedené

SRAM Static Random Access Memory
typ počítačové paměti, která k uchování dat nepotřebuje jejich periodickou obnovu, což je hlavní rozdíl oproti DRAM
obě zmíněné paměti po vypnutí přísunu elektrické energie přijdou o veškerý obsah

šifrový text
zašifrovaný text, který není možné bez dešifrování přečíst

TMR Triple Major Redundancy
blokový model pro návrh FT systémů
klíčový blok se ztrojí a použitým výstupem bude majorita ze tří výstupů každého bloku

top modul
při návrhu hardware se celý návrh rozdělí na menší bloky (moduly), z nichž top modul je ten, který je nejvíce navrchu (obsahuje v sobě všechny bloky návrhu)

XOR
binární operace s bity
$a \oplus b = 1$, pokud je právě jeden z bitů roven 1 a druhý 0, jinak je výstupem 0

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `doc` ..... veškerá dokumentace v podobě návrhových schémat
- `src` ..... zdrojové kódy implementace
  - `variants` ..... zdrojové kódy všech variant AES
    - `AES_1Aa`
    - `AES_1Ar`
    - `AES_1p`
    - `AES_1Ta`
    - `AES_1Tr`
    - `AES01`
    - `AES02`
    - `Přehled_implementovaných_variant.txt` ..... přehled a charakteristiky variant AES
  - `encr1_aes_for_dpa.zip` ... archiv s projektem pro software Quartus
- `text` ..... elektronická verze bakalářské práce
  - `src` ..... zdrojová forma práce ve formátu LaTeX
  - `Implementace_AES_pro_FPGA_Tomáš_Zimmerhakl.pdf` .. text práce ve formátu PDF
  - `Zadání_bakalářské_práce_Tomáš_Zimmerhakl.pdf` .. zadání práce ve formátu PDF

Příloha C

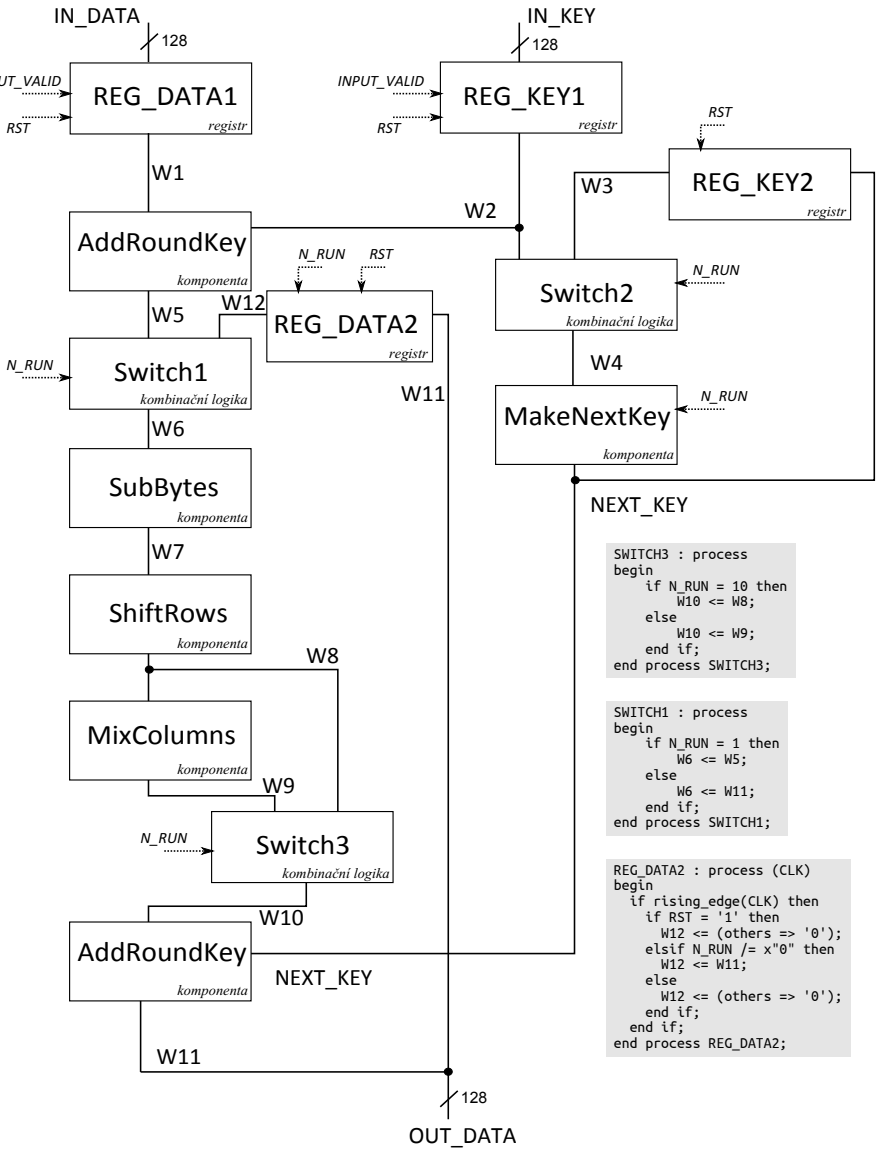
Dokumentace

Tato příloha obsahuje kompletní dokumentaci všech navržených a implementovaných variant. Pro každou variantu je uvedena minimálně datová cesta, řadič a vnitřní zapojení nejvyšší entity. Následující tabulka zobrazuje přehled variant a odkazy na stránky s dokumentací a stránky s popisem a charakteristikou.

Tabulka C.1: Přehled dokumentace k implementovaným variantám AES

Verze AES	*Strana s dokumentací*	*Strana s popisem*
`AES01`	66	19
`AES02`	68	21
`AES_1Ar`	70	25
`AES_1Aa`	72	26
`AES_1Tr`	73	27
`AES_1Ta`	76	28
`AES_1p`	78	29

C.1 AES01



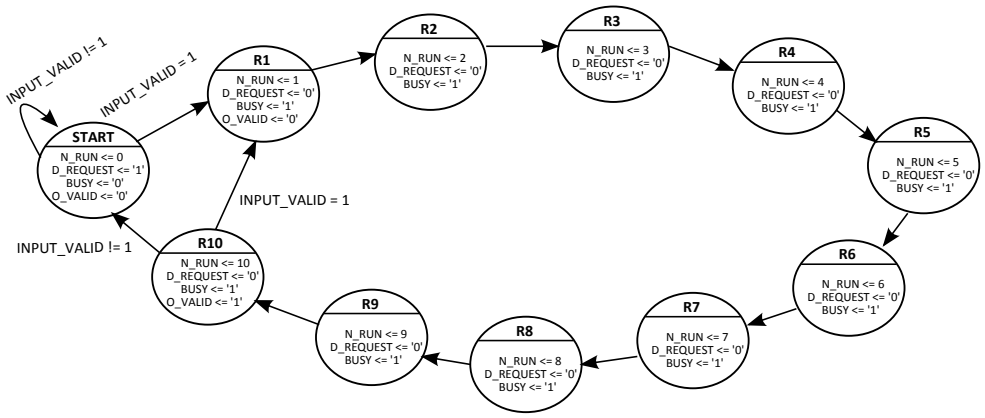

```
SWITCH3 : process
begin
    if N_RUN = 10 then
        W10 <= W8;
    else
        W10 <= W9;
    end if;
end process SWITCH3;
```

```
SWITCH1 : process
begin
    if N_RUN = 1 then
        W6 <= W5;
    else
        W6 <= W11;
    end if;
end process SWITCH1;
```

```
REG_DATA2 : process (CLK)
begin
  if rising_edge(CLK) then
    if RST = '1' then
      W12 <= (others => '0');
    elsif N_RUN /= x"0" then
      W12 <= W11;
    else
      W12 <= (others => '0');
    end if;
  end if;
end process REG_DATA2;
```

- Všechny dráty, kde není uvedeno jinak, mají šířku 128b
- kurzívou a tečkovanými čárami jsou označeny řídící signály:
  INPUT_VALID ..... platný vstup (1 b)
  RST ..... reset (1 b)
  N_RUN ..... číslo rundy 0-10_{dec} (4 b)
- pokud N_RUN == 0, pak výpočet neprobíhá
- signál CLK není uveden, vede ke všem registrům

Obrázek C.1: Datová cesta AES01 – DATAPATH_01

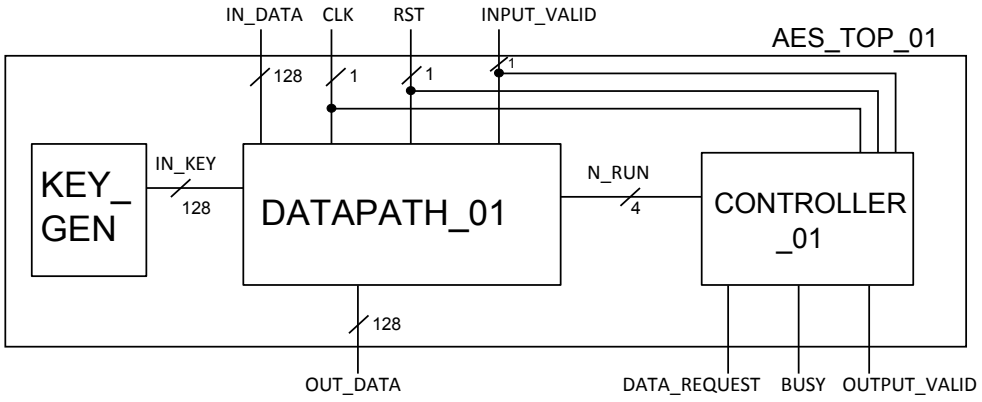


- konečný automat typu Moore
- šipka bez popisu znamená, že přechod je jen na náběžnou hranu hodin
- ze všech stavů se po příchodu signálu RST přechází do stavu Start
- pokud je N_RUN == 0, pak výpočet neprobíhá

Přednastavení hodnot ve stavu START
N_RUN <= 0 (číslo rundy)
D_REQUEST <= '1'
BUSY <= '0'
O_VALID <= '0'

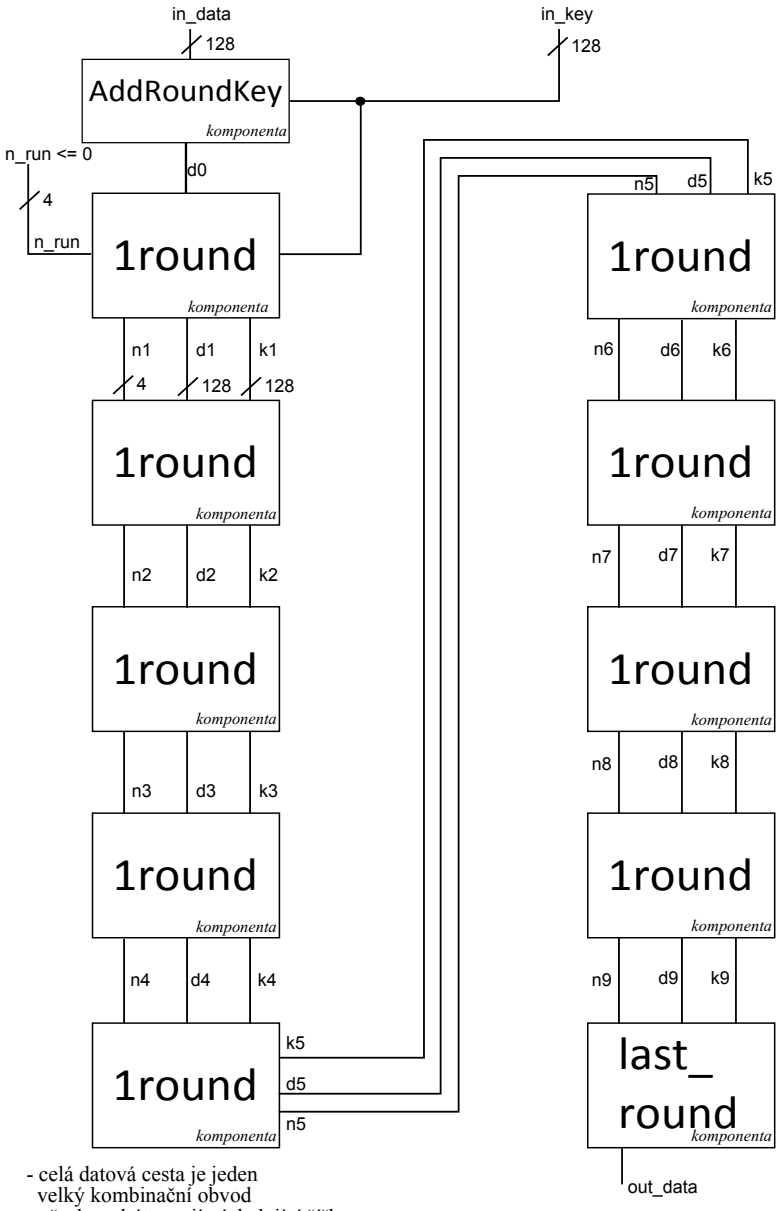
```
entity CONTROLLER_01 is
    port ( CLK          : in std_logic;
           RST          : in std_logic;
           INPUT_VALID  : in std_logic;

           N_RUN        : out std_logic_vector (3 downto 0);
           DATA_REQUEST : out std_logic;
           BUSY         : out std_logic;
           OUTPUT_VALID : out std_logic
    );
end entity CONTROLLER_01;
```

Obrázek C.2: Řadič verze AES01 – CONTROLLER_01

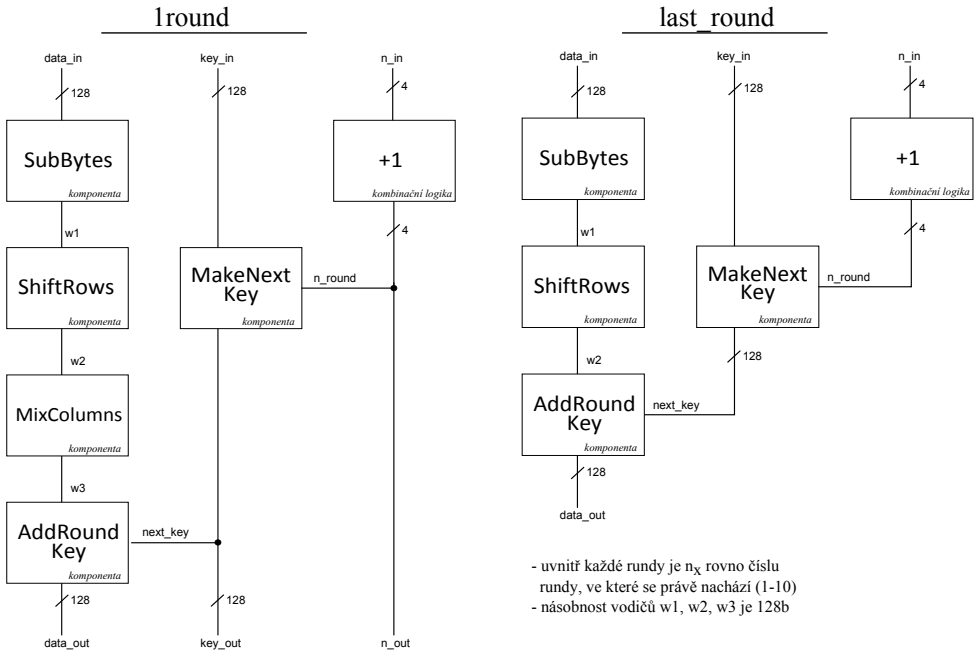


Obrázek C.3: Nejvyšší entita verze AES01 – AES_TOP_01

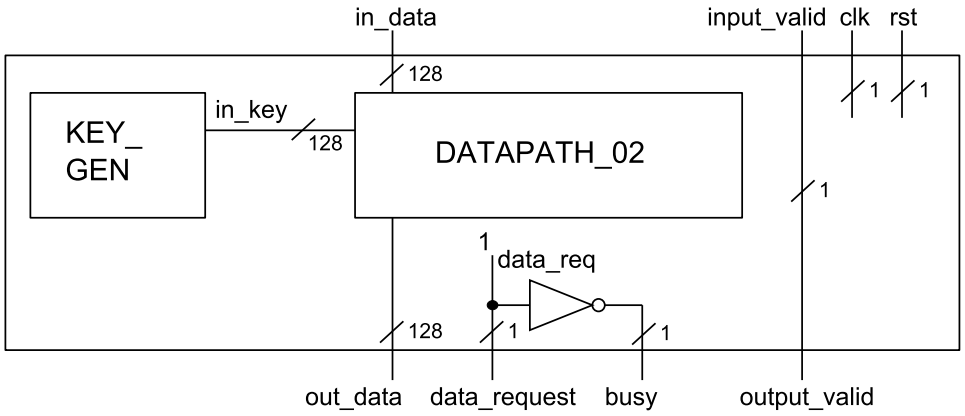
C.2 AES02



Obrázek C.4: Datová cesta AES02 – DATAPATH_02

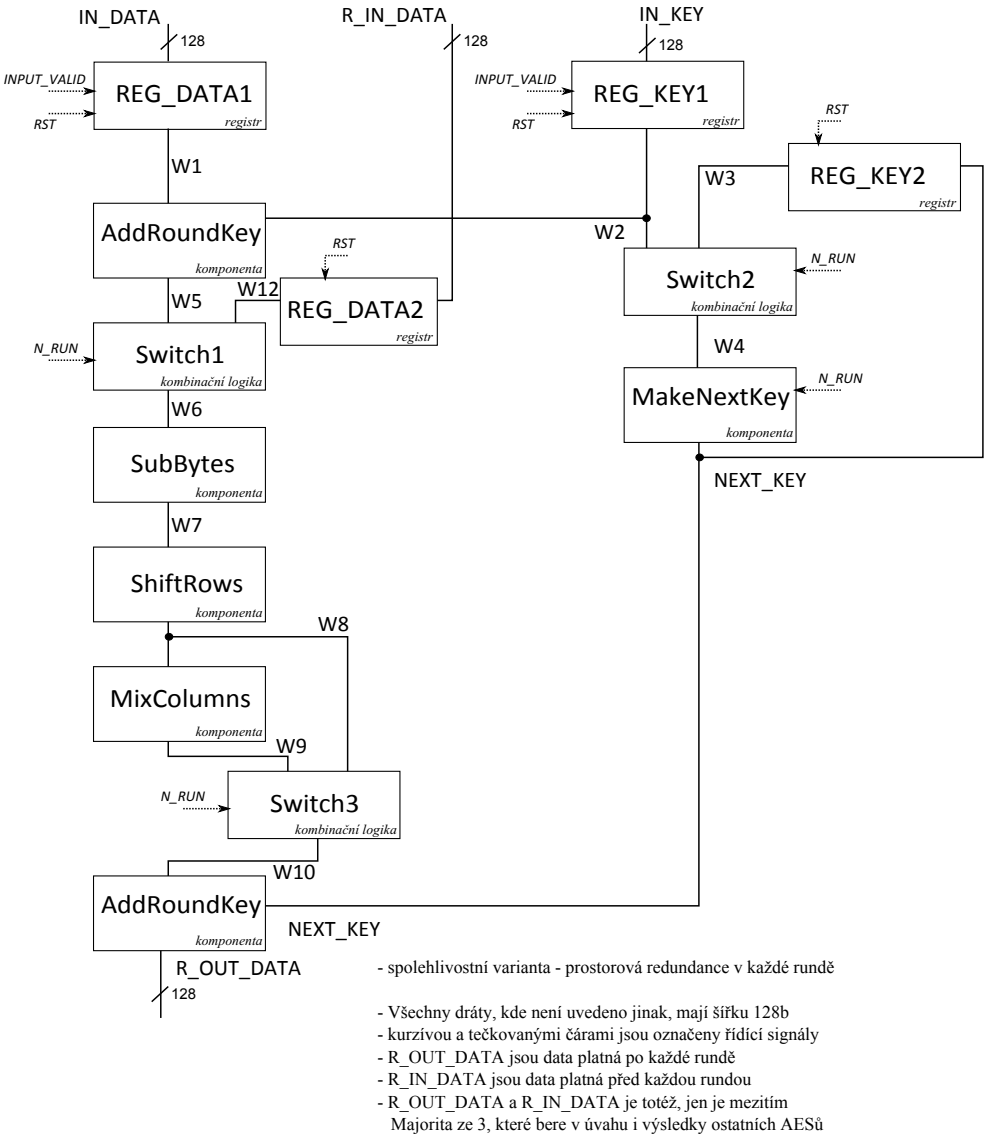


Obrázek C.5: Moduly pro rundy verze AES02 – 1ROUND a LAST_ROUND

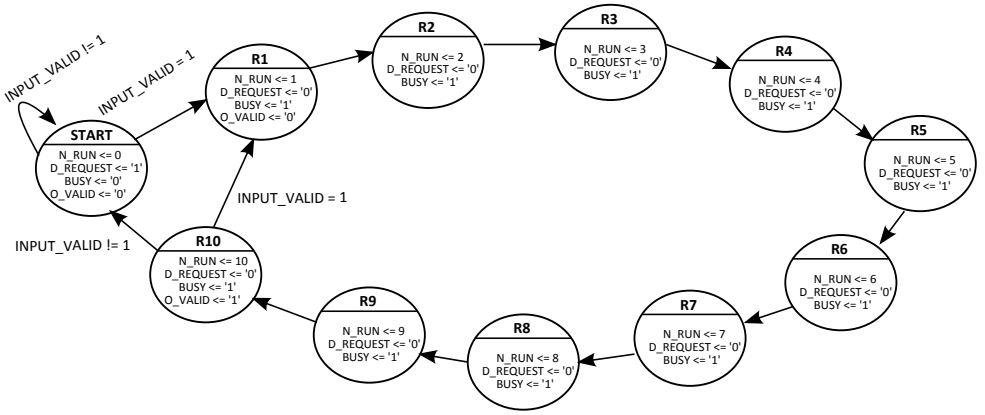


Obrázek C.6: Nejvyšší entita verze AES02 – AES_TOP_02

C.3 AES_1Ar



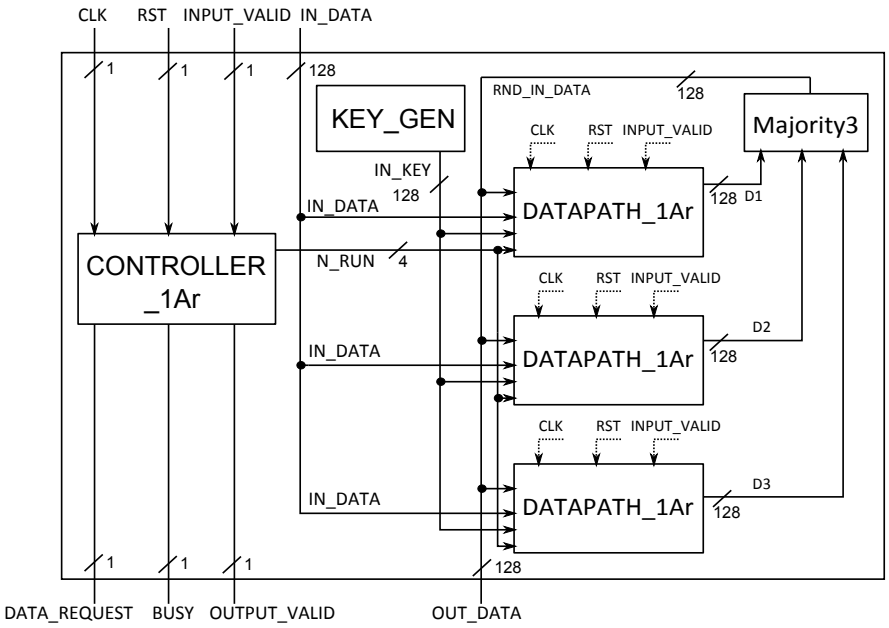
Obrázek C.7: Datová cesta AES_1Ar – DATAPATH_02



- řadič je naprosto identický jako pro variantu AES01
- konečný automat typu Moore
- šipka bez popisu znamená, že přechod je jen na náběžnou hranu hodin
- ze všech stavů se po příchodu signálu RST přechází do stavu Start
- pokud je N_RUN == 0, pak výpočet neprobíhá

```
entity AES_1Ar_CONTROLLER is
    port ( CLK          : in std_logic;
           RST          : in std_logic;
           INPUT_VALID  : in std_logic;

           N_RUN        : out std_logic_vector (3 downto 0);
           DATA_REQUEST : out std_logic;
           BUSY         : out std_logic;
           OUTPUT_VALID : out std_logic
    );
end entity AES_1Ar_CONTROLLER;
```

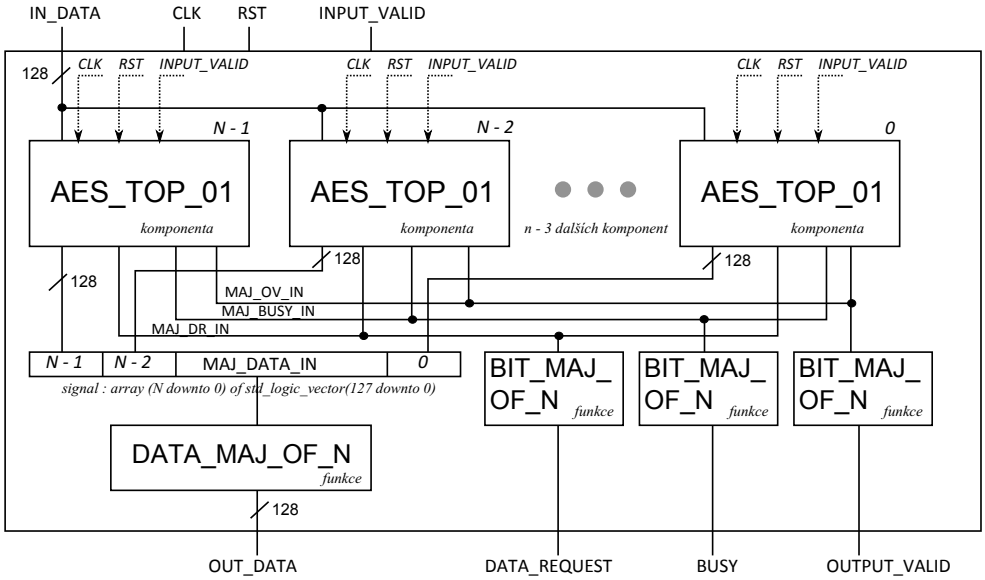
Obrázek C.8: Řadič verze AES_1Ar – CONTROLLER_1Ar



- Majority3 je kombinační funkce, která vybírá majoritu ze 3 výstupů

Obrázek C.9: Nejvyšší entita verze AES_1Ar – AES_TOP_1Ar

C.4 AES_1Aa



- Všechny dráty, kde není uvedeno jinak, mají šířku 128b
- kurzívou a tečkovanými čárami jsou označeny řídící signály
- funkce DATA_MAJ_OF_N přijímá pole N 128bitových vektorů a vrací jeden 128bitový vektor, jehož každý bit je vypočítán jako majorita

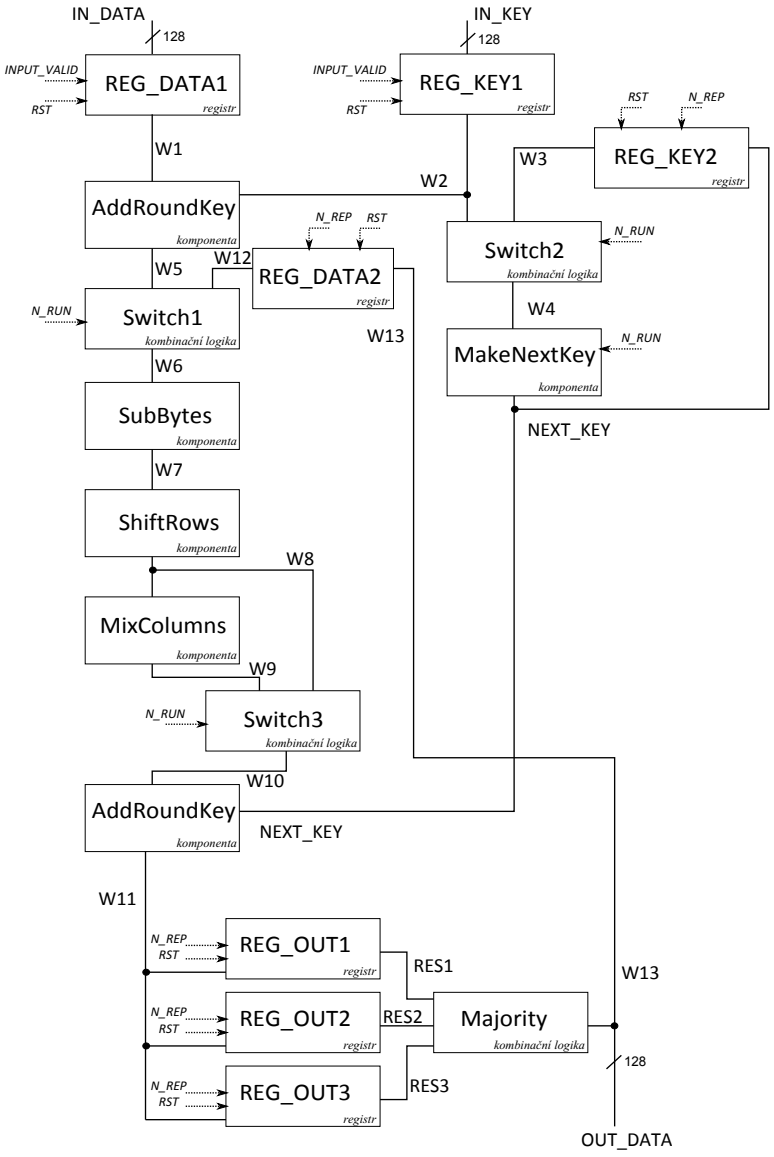
DATA_MAJ_OF_N
Přijímá pole délky N, jehož prvky jsou 128bitové
vektory std-logic MAJ_DATA_IN(n-1) - MAJ_DATA_IN(0).
Výstupem je MAJ_DATA_OUT, což je std_logic_vector (127 downto 0).

Výstup MAJ_DATA_OUT se určí:
MAJ_DATA_OUT(i) = Sum(MAJ_DATA_IN(k)(i), pro k od 0 do n-1) > N/2 ? 1 : 0;
Jinými slovy: každý i-tý bit výstupního vektoru se určí jako 1, pokud většina i-tých bitů každého prvku pole MAJ_DATA_IN je 1, jinak bude 0.

BIT_MAJ_OF_N
Přijímá pole délky N, jehož prvky jsou 1bitové. Na základě toho,
jestli vektor obsahuje více '1' či '0', bude jednobitový výstup '1' nebo '0'.

Obrázek C.10: Nejvyšší entita verze AES_1Aa – AES_TOP_1Aa

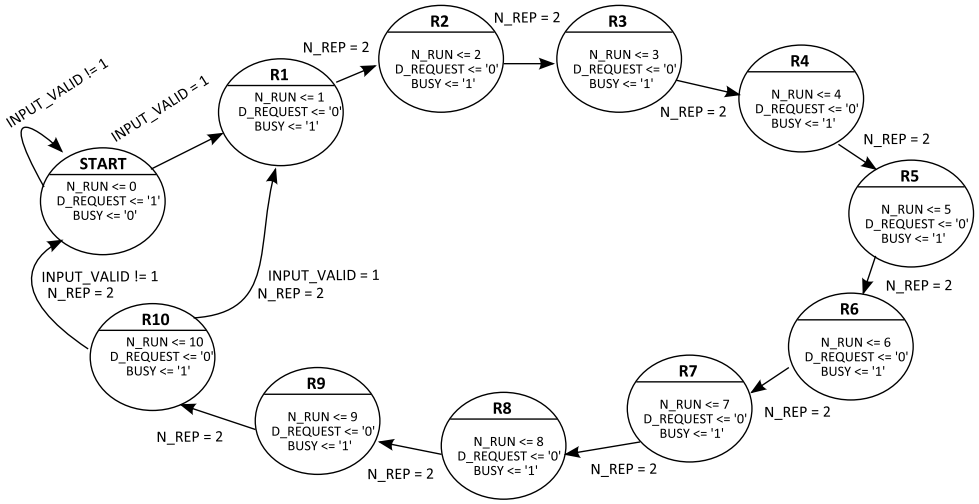
C.5 AES_1Tr



- spolehlivostní varianta - časová redundance po každé rundě

- Všechny dráty, kde není uvedeno jinak, mají šířku 128b
- kurzívou a tečkovanými čárami jsou označeny řídící signály
- REG_DATA2 si pamatuje data, která vstupují do každé rundy, tedy zapisuje jen, když je N_REP == 2
- REG_KEY2 si pamatuje run. klíč, který vstupuje do každé rundy, jen tehdy, pokud je N_REP == 2, tedy při každém posledním opakování si zapamatuje hodnotu pro další rundu

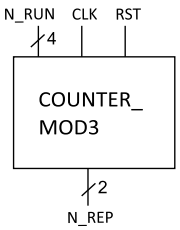
Obrázek C.11: Datová cesta AES_1Tr – DATAPATH_1Tr



- šipka bez popisu znamená, že přechod je jen na náběžnou hranu hodin
- ze všech stavů se po příchodu signálu RST přechází do stavu Start
- pokud je N_RUN == 0, pak výpočet neprobíhá
- signál OUTPUT_VALID nastavuje samostatný proces uvnitř řadiče, OUTPUT_VALID = '1', jedině pokud je N_RUN = 10 a N_REP = 2

```
entity AES_1Tr_CONTROLLER is
    port ( CLK          : in std_logic;
           RST          : in std_logic;
           INPUT_VALID  : in std_logic;

           N_RUN        : out std_logic_vector (3 downto 0);
           N_REP        : out std_logic_vector (1 downto 0);
           DATA_REQUEST : out std_logic;
           BUSY         : out std_logic;
           OUTPUT_VALID : out std_logic
    );
end entity AES_1Tr_CONTROLLER;
```

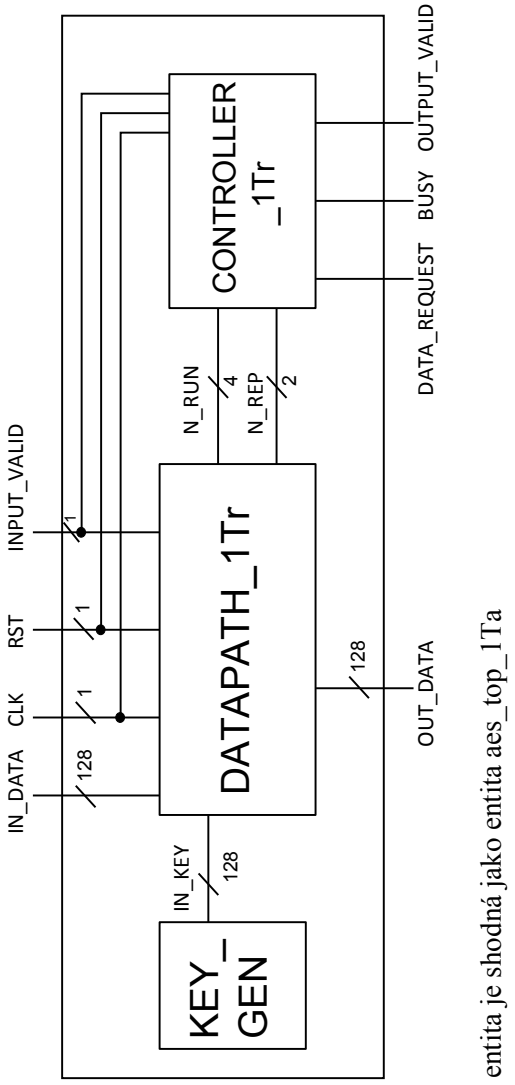
Součástí řadiče je i
čítač modulo 3
counter_mod3_1Tr:



čítač modulo 3 s resetem
inkrementuje se na každou náběžnou hranu hodin,
pokud není N_RUN == 0

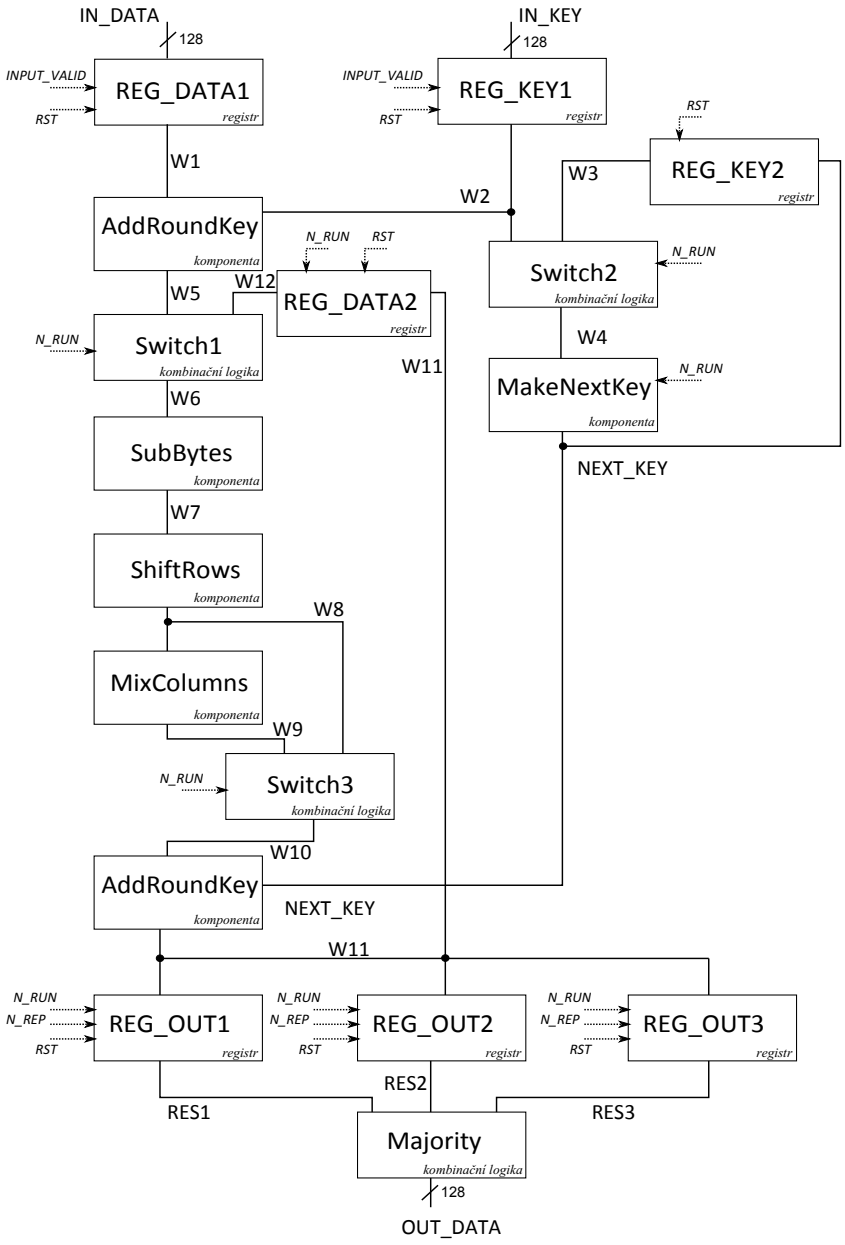
```
entity COUNTER_MOD3_1Tr is
    port ( CLK          : in std_logic;
           RST          : in std_logic;
           N_RUN        : in std_logic_vector (3 downto 0);

           N_REP        : out std_logic_vector (1 downto 0);
    );
end entity COUNTER_MOD3_1Tr;
```

Obrázek C.12: Řadič verze AES_1Tr – CONTROLLER_1Tr



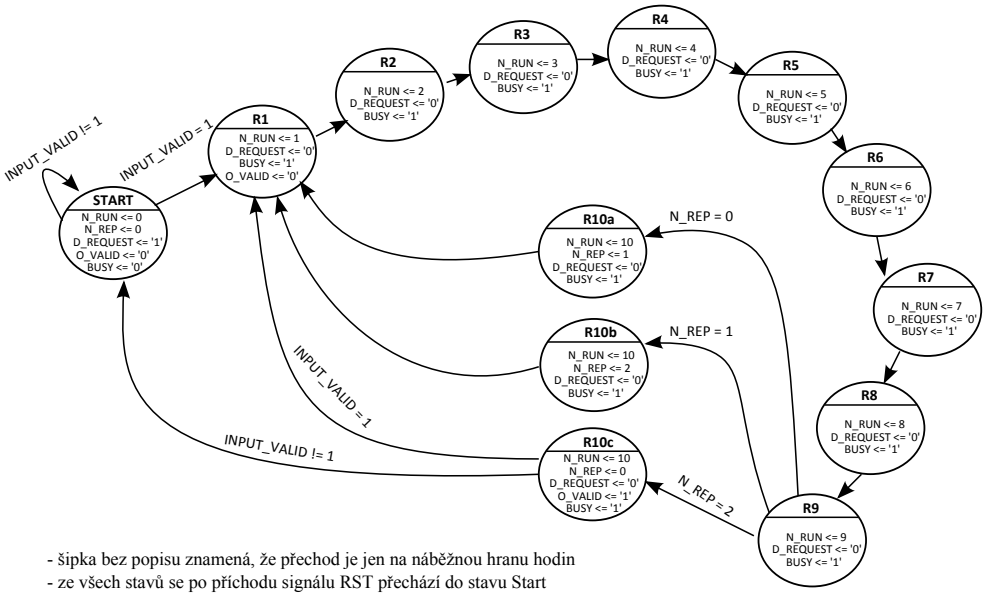
Obrázek C.13: Nejvyšší entita verze AES_1Tr – AES_TOP_1Tr

C.6 AES_1Ta



- Všechny dráty, kde není uvedeno jinak, mají šířku 128b
- kurzívou a tečkovanými čárami jsou označeny řídící signály

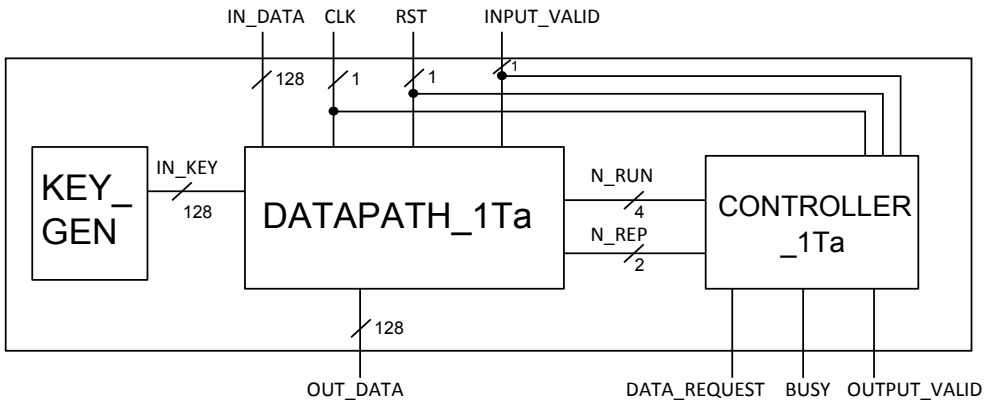
Obrázek C.14: Datová cesta AES_1Ta – DATAPATH_1Ta



- šipka bez popisu znamená, že přechod je jen na náběžnou hranu hodin
- ze všech stavů se po příchodu signálu RST přechází do stavu Start
- pokud je N_RUN == 0, pak výpočet neprobíhá
- N_RUN ... číslo rundy; N_REP ... číslo opakování

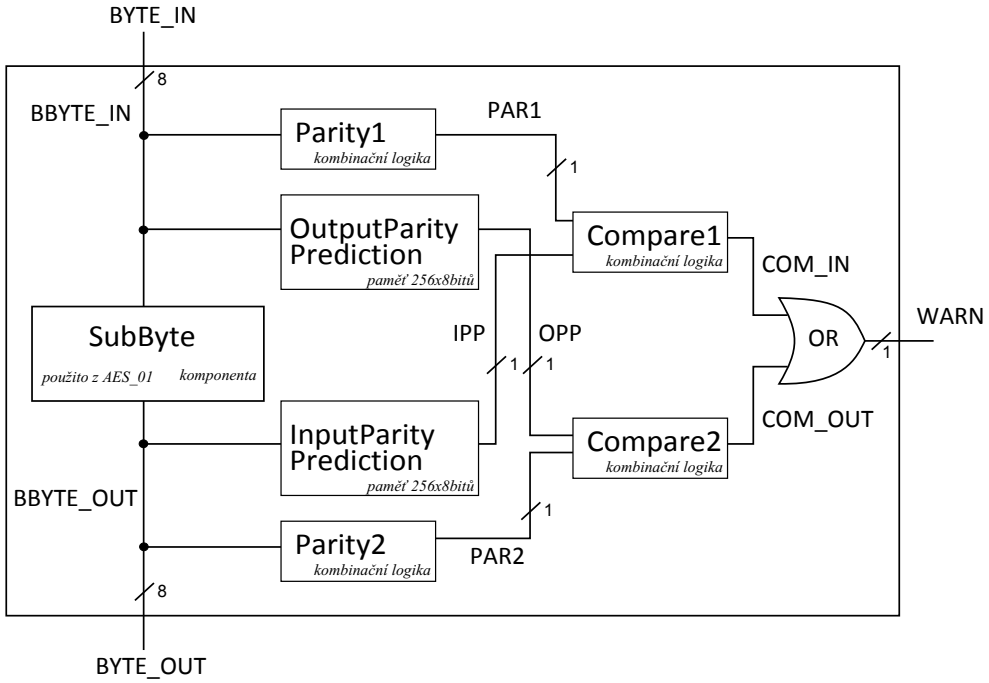
```
entity AES_1Ta_CONTROLLER is
    port ( CLK          : in std_logic;
           RST          : in std_logic;
           INPUT_VALID  : in std_logic;

           N_RUN        : out std_logic_vector (3 downto 0);
           N_REP        : inout std_logic_vector (1 downto 0);
           DATA_REQUEST : out std_logic;
           BUSY         : out std_logic;
           OUTPUT_VALID : out std_logic
    );
end entity AES_1Ta_CONTROLLER;
```

Obrázek C.15: Řadič verze AES_1Ta – CONTROLLER_1Ta



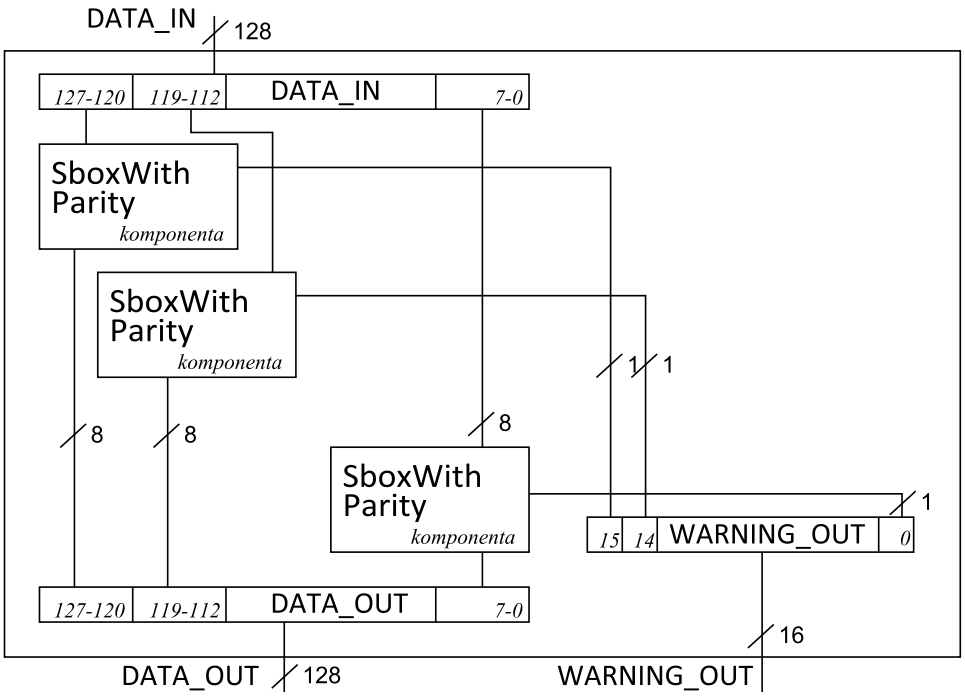
Obrázek C.16: Nejvyšší entita verze AES_1Ta – AES_TOP_1Ta

C.7 AES_1p



- entita SBOX_WITH_PARITY je kombinační obvod
- Parity1 a Parity2 - stejná implementace, z 8bitového vstupu
  vytvoří 1bitový výstup, tak že provede xor všech bitů ze vstupu
- Compare1 a Compare2 - stejná implementace, srovná 2 bity na vstupu,
  pokud se shodují, pošle na výstup '0', jinak '1'
- Warning - pokud alespoň jeden Compare hlásí '1', došlo někde chybě a
  WARNING <= '1', jinak '0'
- InputParityPrediction - asociativní paměť 256 x 8bitů
  podle BBYTE_OUT určí paritu, která odpovídá bytu BBYTE_IN
- OutputParityPrediction - asociativní paměť 256 x 8bitů
  podle BBYTE_IN určí paritu, která odpovídá bytu BBYTE_OUT

Obrázek C.17: AES_1p, SBox s predikcí parity – SBOX_WITH_PARITY



Obrázek C.18: AES_1p, Sub_Bytes s paritou – SUB_BYTES_WITH_PARITY