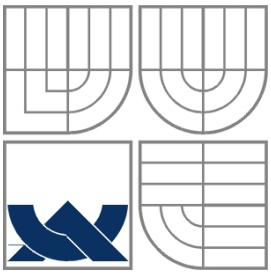
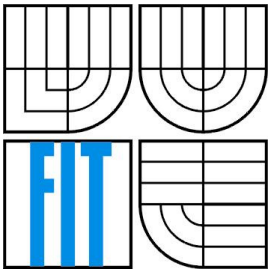


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

IMPLEMENTACE KRYPTOGRAFICKÝCH ALGORITMŮ V FPGA

IMPLEMENTATION OF CRYPTOGRAPHIC ALGORITHMS IN FPGA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — Tomáš Foltýn
AUTHOR

VEDOUCÍ PRÁCE — Doc. Dr. Ing. Petr Hanáček
SUPERVISOR

BRNO 2016

Abstrakt

Tato práce se zabývá návrhem a implementací šifrovacího algoritmu AES v programovatelném hradlovém poli (FPGA). Návrh jednotky se zaměřuje na kompaktní design a výsledná datová propustnost je spíše druhotná. Implementovaná jednotka je schopná šifrování i dešifrování dat s použitím uživatelem zvoleného klíče. Funkčnost implementace byla ověřena v simulačním programu ModelSim a experimentálně na vývojovém kitu FITkit osazeném FPGA z rodiny Spartan 3.

Abstract

This thesis describes design and implementation of the AES cryptographic algorithm in FPGA. Design of this unit aims at compact size in exchange for lower throughput. Implemented unit is able of both ciphering and deciphering with user selected key. Resulting design was tested in ModelSim program and on FITkit development board with Spartan 3 family FPGA.

Klíčová slova

Kryptografie, FPGA, AES, FITkit, VHDL

Keywords

Cryptography, FPGA, AES, FITkit, VHDL

Citace

Foltýn Tomáš: Implementace kryptografických algoritmů v FPGA, bakalářská práce, Brno, FIT VUT v Brně, 2016

Implementace kryptografických algoritmů v FPGA

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Doc. Dr. Ing. Petra Hanáčka.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Tomáš Foltýn
16. května 2016

Poděkování

Rád bych poděkoval Doc. Dr. Ing. Petru Hanáčkovi za odborné vedení při psaní této práce a svému zaměstnavateli a projevenou vstřícnost při dokončování této práce.

Obsah

Kapitola 1

Úvod

V několika posledních desetiletích jsme svědky velkého a překotného vývoje v téměř všech oblastech lidské činnosti. Tento rychlý vývoj je mnohdy dopředu hnán schopnostmi výpočetní techniky, která sama prochází rychlým vývojem. Rostoucí výkon, zmenšující se rozměry a větší dostupnost počítačů v uplynulých dekádách způsobily, že počítače dnes téměř nahradily jiné druhy zpracování a uchování informací. To do odvětví výpočetní techniky přineslo nové výzvy v oblasti zabezpečování informací. A nejedná se jen o zabezpečení dat v místě, kde jsou uložena, ale také, a to je možná ještě důležitější, o zabezpečení velkého množství informací, které dnes přes počítače všeho druhu proudí. To klade vysoké nároky na výkon šifrovacích systémů. Nejmodernější výkonné procesory již obsahují speciální obvody a instrukce urychlující kryptografické operace, přesto existuje mnoho oblastí, kde je jejich nasazení nedostačující, nevhodné, nebo dokonce nemožné. Z těchto důvodů se v takových situacích používají obvody specializované na konkrétní typ úlohy, v tomto případě na šifrování. Jeden ze způsobů, jak tyto obvody realizovat, je použití rekonfigurovatelných obvodů FPGA.

Cílem této bakalářské práce je realizace šifrovacího algoritmu v takovémto FPGA. Jako algoritmus k realizaci byl vybrán AES, který se v současnosti těší velké oblibě a je považován za moderní a bezpečný algoritmus. Cílem realizace bude vytvořit kompaktní a úsporný návrh, který bude možno použít v malých zařízeních, se zaměřením na odolnost proti vnějším útokům.

Kapitola 2

Teoretický rozbor

Tato kapitola se zabývá teoretickou částí zvoleného tématu. Nejprve zde rozeberu princip fungování zvoleného algoritmu, poté stručně popíšu konstrukci hradlového pole a možná úskalí, která plynou z převodu šifrovacích algoritmů do obvodové reprezentace. Nakonec rozeberu několik již existujících řešení zvoleného tématu.

2.1 Advanced Encryption Standard

Zkráceně AES, dříve známý též jako Rijndael je šifrovací algoritmus, výherce soutěže americké vlády o nový šifrovací standard (odtud také jeho současné jméno). Jedná se o symetrickou blokovou šifru s pevnou velikostí bloku 128 bitů. Délka klíče se může měnit mezi 128, 192 a 256 bity. Algoritmus je založen na principu SP sítí (substitution-permutation network). Jedním z hlavních důvodů, proč byl tento algoritmus upřednostněn před ostatními finalisty v soutěži, je jeho rychlost jak v softwarové implementaci, tak při implementaci v hardwaru.

Já jsem si tento algoritmus vybral především proto, že se jedná o jeden z nejrozšířenějších šifrovacích algoritmů současnosti.

2.1.1 Rozbor činnosti algoritmu

Tato část práce je založena na článku [5]. Základem práce algoritmu je provádění tzv. rund (angl. round) nad maticí 4 x 4 byty, které se několikrát opakují v závislosti na délce klíče. V každé rundě se na stavovou matici aplikují 4 různé transformace: 1) provede se substituce daná substituční tabulkou (tzv. S-box), 2) řádky matice se cyklicky posunou, 3) data v každém sloupci se zkombinují s transformační maticí, 4) na sloupce matice se aplikuje podklíč, který je odvozen od hlavního klíče.

Protože jsou všechny operace reverzibilní, lze dešifrování provést pomocí operací inverzních k šifrovacím operacím.

Pro lepší představu, jak algoritmus funguje, se můžeme podívat na pseudokód 1.

Nb zde značí počet sloupců ve stavové matici (standardně 4)
Nr značí počet rund
w je pole obsahující podklíče

```
Cipher(byte in[4*Nb], byte out[4*Nb], word w[Nb*(Nr+1)])
begin
   byte state[4,Nb]
   state = in
   AddRoundKey(state, w[0, Nb-1])
   for round = 1 step 1 to Nr-1
      SubBytes(state)
      ShiftRows(state)
      MixColumns(state)
      AddRoundKey(state, w[round*Nb, (round+1)*Nb-1])
   end for
   SubBytes(state)
   ShiftRows(state)
   AddRoundKey(state, w[Nr*Nb, (Nr+1)*Nb-1])
   out = state
end
```

Zdrojový kód 1: Pseudokód AES šifry [5]

SubBytes transformace

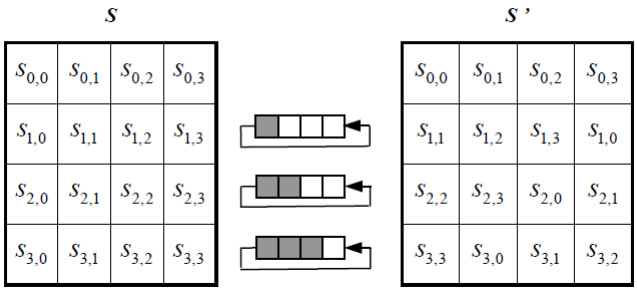
Každý byte matice je zaměněn za hodnotu ze substituční tabulky (tzv. S-box). Tento S-box je vytvořen zkombinováním inverzního zobrazení daného čísla v konečném tělese GF(2^8) a aplikováním afinní transformace dle rovnice 2.1.

$$b_i' = b_i \oplus b_{(i+4)mod8} \oplus b_{(i+5)mod8} \oplus b_{(i+6)mod8} \oplus b_{(i+7)mod8} \oplus c_i, \tag{2.1}$$

kde b_i značí i-tý bit bytu a c_i je i-tý bit bytu c, který je roven hexadecimálnímu číslu 63.

ShiftRows transformace

Spodní tři řádky matice jsou cyklicky posunuty o určený počet bytů. První řádek posunut není. Druhý řádek je posunut o jeden byte vlevo, s tím, že „vysunutý“ byte je zařazen nakonec (cyklické posunutí). Třetí řádek je posunut o dva byty vlevo a poslední řádek je posunut o tři byty vlevo.



Obrázek 1: ShiftRows transformace

***MixColumns* transformace**
Každý sloupec stavové matice je vynásoben pevnou maticí. Výsledkem je sloupec s novými hodnotami.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{kde } c \text{ značí číslo sloupce.}$$

(2.2)

Aby byl výsledek tohoto násobení vidět lépe, můžeme si jej přepsat do rovnicového tvaru:

$$\begin{aligned} s'_{0,c} &= (\{02\} \cdot s_{0,c}) \oplus (\{03\} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \cdot s_{1,c}) \oplus (\{03\} \cdot s_{2,c}) \oplus s_{3,c} \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \cdot s_{2,c}) \oplus (\{03\} \cdot s_{3,c}) \\ s'_{3,c} &= (\{03\} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{03\} \cdot s_{3,c}) \end{aligned}$$

(2.3)

Znak $\oplus$ v tomto případě znamená operaci XOR (exkluzivní disjunkce) a znak $\cdot$ značí násobení v konečném tělese.

***AddRoundKey* transformace**
Ke každému bytu v matici je pomocí operace XOR přidán odpovídající byte z podklíče podle následující rovnice:

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round*Nb+c}] \quad \text{kde } 0 \leq c < 4.$$

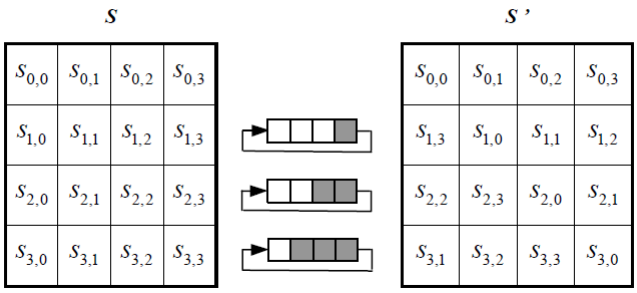
(2.4)

AddRoundKey transformace tedy vezme sloupec stavu *s* a pomocí operace XOR k tomuto sloupci přidá odpovídající podklíč.

***InvSubBytes* transformace**
Tato transformace je totožná se *SubBytes* transformaci, ale využívá inverzní substituční tabulky. Tento inverzní S-box je vytvořen aplikováním afinní transformace inverzní k transformaci popsané rovnicí 2.1 a aplikací inverzního zobrazení daného čísla v GF(2^8).

***InvShiftRows* transformace**
Tato transformace je opakem *ShiftRow* transformace. Spodní tři řádky matice jsou cyklicky posunuty o určený počet bytů. První řádek posunut není. Druhý řádek je posunut o jeden byte

vpravo, s tím, že „vysunutý“ byte je zařazen nakonec (cyklické posunutí). Třetí řádek je posunut o dva byty vpravo a poslední, čtvrtý, řádek je posunut o tři byty vpravo.



Obrázek 2: InvShiftRow transformace

InvMixColumns transformace

Jedná o inverzní transformaci k *MixColumns* transformaci. Každý sloupec stavové matice je vynásoben pevnou maticí. Výsledkem je sloupec s novými hodnotami.

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{kde } c \text{ značí číslo sloupce.} \tag{2.5}$$

Tento zápis můžeme přepsat do rovnicového tvaru takto:

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

Znak $\oplus$ v tomto případě opět znamená operaci XOR a znak $\cdot$ značí násobení v konečném tělese.

AddRoundKey transformace

Inverzní transformace *AddRoundKey* je shodná s *AddRoundKey* transformací. Jediný rozdíl je, že ke každému bytu v matici je pomocí operace XOR přidán odpovídající byte z inverzního podklíče.

Generování podklíčů

Na začátku každého šifrování se z primárního klíče vygenerují podklíče, které se při AddRoundKey transformaci aplikují na jednotlivé byty matice. Tyto klíče se liší pro každou rundu, ale pro daný klíč jsou vždy stejné, takže není nutné toto generování provádět pro každý šifrovaný blok znovu. Při generování těchto podklíčů se nejdříve zkopíruje primární šifrovací

klíč na začátek pole obsahujícího vygenerované podklíče. Každý další podklíč *w[i]* poté vzniká XORováním aktuálního stavu a klíče *w[i – Nk]*. *Nk* značí počet slov primárního klíče (tedy pro 128 bitový klíč se *Nk* rovná 4, pro 192 bitový je *Nk* 6, atd.). Na klíče rovnající se násobku *Nk* se navíc aplikují *SubWord* a *RotWord* transformace a hodnota z pole *Rcon*.
SubWord transformace vezme 32 bitů generovaného podklíče a každý byte nahradí hodnotou z S-box tabulky stejným způsobem jako při *SubBytes* transformaci. *RotWord* transformace taktéž vezme aktuálně generovaný podklíč a cyklicky jej posune o jeden byte doleva. *Rcon[i]* je pole obsahující 32 bitové hodnoty [x^{i-1}, {00}, {00}, {00}]. x^{i-1} je mocnina *x* v konečném tělese GF(2^8) a *x* je rovno {02} hexadecimálně.
Pro provádění inverzní transformace (operace dešifrování) je nutné použít inverzní podklíče. Ty se generují aplikací *InvMixColumns* transformace na již vygenerované podklíče. Na první a poslední rundovní klíč není tato transformace aplikována. Nejlépe proces generování podklíčů ukazuje pseudokód 2.

```
KeyExpansion(byte key[4*Nk], word w[Nb*(Nr+1)], Nk)
begin
   word temp
   i = 0
   while (i < Nk)
      w[i] = word(key[4*i], key[4*i+1], key[4*i+2], key[4*i+3])
      i = i+1
   end while
   i = Nk
   while (i < Nb * (Nr+1)]
      temp = w[i-1]
      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i/Nk]
      else if (Nk > 6 and i mod Nk = 4)
         temp = SubWord(temp)
      end if
      w[i] = w[i-Nk] xor temp
      i = i + 1
   end while
end

for i = 0 step 1 to (Nr+1)*Nb-1
   dw[i] = w[i]
end for
for round = 1 step 1 to Nr-1
   InvMixColumns(dw[round*Nb, (round+1)*Nb-1])
end for
```

Zdrojový kód 2: Pseudokód generování podklíčů [5]

2.2 Postranní kanály

Útok postranními kanály je jeden z druhů útoku na kryptografický systém. Znamená, že se nesnažíme najít slabinu v šifrovacím algoritmu, ale v jeho implementaci. Typů těchto útoků je celá řada, můžeme například měřit spotřebu systému při provádění šifrování, čas, za který dostaneme výsledek výpočtu, nebo elektromagnetické záření, které systém při výpočtu vydává. Další typ útoků se může soustředit na umělé zavedení chyb do výpočtu tak, aby se systém dostal do neočekávaného stavu. Chování systému potom může útočníkovi napovědět o jeho vnitřní stavbě. Tyto chyby se můžeme pokusit zavést například krátkou změnou napětí nebo frekvence, nebo vystavením obvodu hraničním teplotám.

Existuje několik způsobů, jak se bránit těmto útokům. Jednak můžeme snížit množství informace vydávané systémem, nebo můžeme snížit závislost mezi vydávanou informací a šifrovanými daty. Jednou z metod obrany, především proti útokům, které měří čas potřebný k provedení výpočtu, je použití isochronního kódu, tj. kódu, jehož provedení trvá vždy stejnou dobu. Závislost mezi dobou výpočtu a daty můžeme také snížit tím, že do výpočtu vložíme instrukce, které s vlastním výpočtem nemají nic společného. Dalším způsobem, jak snížit množství vydávané informace je použít kód, který má stále stejnou Hammingovu váhu nehledě na zpracovávanou informaci.

2.3 FPGA

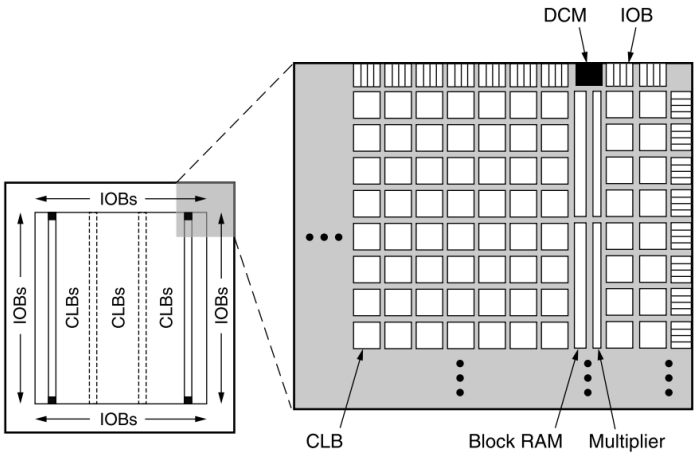
Field-programmable Gate Array (programovatelné hradlové pole) je rekonfigurovatelný číslicový integrovaný obvod. FPGA se skládají z obecných programovatelných bloků a konfigurovatelných propojení mezi nimi, takže jsou velmi flexibilní, co se naprogramovaného obvodu týče a nabízí téměř neomezený počet rekonfigurací. Kromě těchto obecných, tzv. logických, bloků obsahuje většina současných FPGA i jiné specializovanější bloky, jako je paměť RAM, specializované DSP obvody, nebo jednoduché procesorové jádro. Konfigurace pro FPGA se v současnosti definují především pomocí HDL jazyků, ale jsou možné i jiné zápisy, například pomocí schematického popisu. Mezi nejznámější HDL jazyky patří Verilog a VHDL.

2.3.1 VHDL

VHSIC Hardware Description Language je jazyk pro popis hardware, původně vyvinutý v 80. letech pro potřeby americké armády. Jedná se o silně typovaný, obecně použitelný jazyk, který principielně vychází z programovacího jazyku Ada.

2.3.2 FPGA Xilinx Spartan-3

FPGA Spartan-3, konkrétně XC3S50, obsahuje 192 konfigurovatelných logických bloků (CLB) a je složeno z 50 tisíc logických hradel [8]. Dále obsahuje 72 Kb blokové RAM, 12Kb distribuované RAM a čtyři 18ti bitové sčítačky. Každý CLB blok je složen ze 4 tzv. *slice*, které obsahují D klopné obvody, LUT tabulky a další podpůrnou logiku. Jak jsou jednotlivé bloky organizovány, je schematicky zobrazeno na obrázku 3.



Obrázek 3: architektura Spartan 3 FPGA

Toto FPGA je osazeno na školním výukovém kitu FITkit, na kterém budu svůj projekt implementovat. Pokud se mnou navržený obvod do této verze nevejde, použiji jiné dostupné FPGA, které nese označení XC3S400 a obsahuje 8x více logických hradel (400 tisíc). Tato větší verze je osazena na vybraných kusech kitu FITkit.

2.4 Existující implementace

Součástí teoretické přípravy byl i průzkum již existujících řešení. Cílem tohoto průzkumu bylo získat základní přehled o existujících typech řešení, jejich silných a slabých stránkách a další implementační poznatky. Z mnoha těchto řešení jsem vybral několik, která bych zde chtěl v krátkosti představit a poukázat v nich na některé důležité vlastnosti, které ovlivnily tvorbu mého návrhu.

2.4.1 Tim Good a spol.

Tim Good a spol. ve své práci *AES on FPGA from the fastest to the smallest* [2] zkoumá mimo jiné i možnosti velmi kompaktní implementace AES algoritmu. Pro dosažení co nejmenších rozměrů obvod implementoval jako ASIP (application specific instruction processor), tedy procesor s velmi specifickým využitím. Dalšího zmenšení dosáhl tím, že procesor je pouze 8 bitový, je tedy schopný zpracovat pouze jeden byte z 16ti bytové matice za takt. To se odrazilo na výsledné rychlosti šifrování, která dosahuje pouze 2,2 Mb za sekundu. Výsledný design ovšem zabírá pouze 124 *slice* a 2 bloky BRAM kde je uložen S-box a další podpůrná data. Jedná se tak asi o nejmenší implementaci AES obvodu.

2.4.2 Pawel Chodowiec a spol.

Pawel Chodowiec a spol. se ve své práci *Very Compact FPGA Implementations of the AES Algorithm* [3] zaměřují specificky na kompaktní implementaci AES algoritmu. Hlavním znakem tohoto řešení je zmenšení cest na 32 bitů místo plných 128. Dalším znakem je použití tzv. složeného registru (angl. Folded Register) pro uložení stavové matice. To spolu

s promyšleným způsobem čtení a zápisu do tohoto registru umožnilo provést operaci *ShiftRows* během těchto čtení a zápisů bez dalších potřebných prostředků. Druhým efektem tohoto přístupu byla možnost odbourat další registry pro mezivýsledky a snížit tedy celkové nároky na hardware. Dalším aspektem jejich architektury je navržení *MixColumns* a *InvMixColumns* části tak, že *InvMixColumns* transformace využívá velkou část obvodu pro výpočet *MixColumns* transformace a je tedy implementována efektivně. Jejich design dosahuje propustnosti 166 Mb za sekundu při použití 222 *slice* a 3 blokových RAM, což je téměř přesně polovina jimi použitého FPGA Xilinx Spartan-2.

Implementaci a samotnou ideu složeného registru prezentovanou v tomto návrhu považuji za velmi zajímavou a ve svém řešení použiji podobnou implementaci.

2.4.3 Gael Rouvroy a spol.

Rouvroy a spol. se ve svém *Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael* [4] také zaměřuje na implementaci kompaktního AES obvodu. Malých rozměrů dosahuje také zmenšením cest na 32 bitů místo plných 128bitů, ale hlavně použitím vyhledávacích tabulek místo *SubBytes* a *MixColumns* transformací. Pokud totiž spojíme tyto dvě transformace do jedné, tak je možné degradovat výpočet *MixColumns* transformace na vyhledání patřičné hodnoty v předpočítané tabulce, která nahrazuje S-box. Jejich design používá 4 tyto tabulky, aby tak dosáhl vyšší propustnosti na úkor vyššího zatížení blokových RAM.

Takto postavená architektura potom dosahuje propustnosti 208 Mb za sekundu při použití 163 *slice* a 2 paměťových bloků. Zde je ovšem nutno podotknout, že ke svým pokusům použili FPGA Xilinx Spartan-3 XC3S50, které používá velké 18kbit BRAM bloky.

Zde bych chtěl ještě poukázat na fakt, že stejné FPGA ke své práci budu používat i já.

Jednotlivé implementace a jejich výsledky jsem pro porovnání shrnul do přehledné tabulky.

Design	Good a spol.	Chodowiec a spol.	Rouvroy a spol.
Zařízení	Spartan II XC2S15-6	Spartan II XC2S30-6	Spartan III XC3S50-4
Datová šířka (bitů)	8	32	32
Zabraných slice	124	222	163
Použitých BRAM	2	3	2
Velikost BRAM (kbitů)	4	4	18
Max. frekvence (MHz)	67	60	71
Propustnost (Mbitů)	2.2	166	208

Tabulka 1: Porovnání návrhů zaměřených na minimální velikost

Kapitola 3
Návrh řešení

Po provedení výše zmíněného průzkumu jsem se rozhodl pro implementaci šifrovacího a dešifrovacího obvodu s omezením na délku klíče pouze 128 bitů. Již existující implementace naznačují, že takovýto obvod by se měl do mnou zamýšleného FPGA Spartan-3 s 50 tisíci logickými hradly vejít.

Vzhledem k omezené velikosti cílového FPGA jsou datové cesty obvodu pouze 32 bitové. To znamená, že propustnost obvodu bude nanejvýš čtvrtinová oproti implementaci s plnou šířkou, také to ale znamená, že nároky na výpočetní zdroje budou také pouze čtvrtinové.

Druhým hlavním aspektem této implementace je použití předpočítaných vyhledávacích tabulek, které nahrazují *MixColumns* a *InvMixColumns* transformace. Při výpočtu je možné přehodit provádění *SubBytes* a *ShiftRows* transformací, takže na sebe poté *SubBytes* a *MixColumns* transformace navazují a lze je sloučit do jedné vyhledávací tabulky. Pro toto řešení jsem se rozhodl proto, že maticové násobení v konečných tělesech je obvodově náročné a minimální velikost obvodu je hlavním cílem tohoto řešení. Tabulky pro tyto transformace jsou uloženy v blokové RAM zařízení, jejich uložení je tedy řešeno efektivně a jejich implementace nezabírá žádné zdroje navíc.

3.1 Výpočet vyhledávací tabulky

Tato podkapitola se opírá o kapitolu 5 v článku [6].
Označme výstupní sloupec *MixColumns* transformace jako *d* a matici počátečního stavu jako *a*. Poté $a_{i,j}$ značí byte matice *a* v řádku *i* a sloupci *j*. Poté *SubBytes* transformaci můžeme vyjádřit jako:

$$b_{i,j} = S[a_{i,j}]. \tag{3.1}$$

Pro *ShiftRows* transformaci máme:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-C1} \\ b_{2,j-C2} \\ b_{3,j-C3} \end{bmatrix}$$ kde *Cx* značí posunutí v příslušném řádku. (3.2)

A pro *MixColumns* transformaci máme:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} \tag{3.3}$$

Všechny tyto transformace můžeme zkombinovat a vyjádřit jako:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \cdot \begin{bmatrix} S[a_{0,j}] \\ S[a_{1,j-C1}] \\ S[a_{2,j-C2}] \\ S[a_{3,j-C3}] \end{bmatrix} \tag{3.4}$$

Násobení matic můžeme vyjádřit jako lineární kombinaci vektorů takto:

$$\begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = S[a_{0,j}] \cdot \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus S[a_{1,j-C1}] \cdot \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus S[a_{1,j-C2}] \cdot \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus S[a_{3,j-C3}] \cdot \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \tag{3.5}$$

Na základě tohoto můžeme definovat vyhledávací tabulky takto:

$$T_0[a] = \begin{bmatrix} S[a] \cdot 02 \\ S[a] \\ S[a] \\ S[a] \cdot 03 \end{bmatrix} \quad T_1[a] = \begin{bmatrix} S[a] \cdot 03 \\ S[a] \cdot 02 \\ S[a] \\ S[a] \end{bmatrix} \quad T_2[a] = \begin{bmatrix} S[a] \\ S[a] \cdot 03 \\ S[a] \cdot 02 \\ S[a] \end{bmatrix} \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \cdot 03 \\ S[a] \cdot 02 \end{bmatrix} \tag{3.6}$$

S takto definovanými tabulkami můžeme *MixColumns* transformaci zapsat jako:

$$d_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-C1}] \oplus T_2[a_{2,j-C2}] \oplus T_3[a_{3,j-C3}] \tag{3.7}$$

Pokud se blíže podíváme na tabulky T_0 až T_3 zjistíme, že za cenu tří rotací navíc můžeme *MixColumns* transformaci implementovat pomocí pouze jedné tabulky takto:

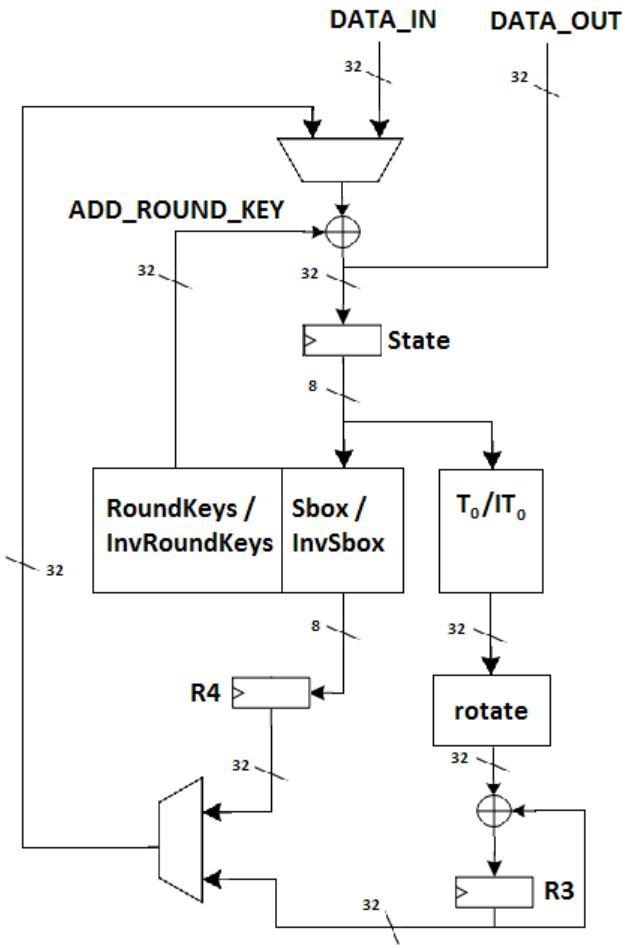
$$d_j = T_0[a_{0,j}] \oplus Rotate\Big(T_0[a_{1,j-C1}] \oplus Rotate\Big(T_0[a_{2,j-C2}] \oplus Rotate(T_0[a_{3,j-C3}])\Big)\Big) \tag{3.8}$$

Výpočet hodnot v tabulce pro dešifrování je postupem obdobný, ale vychází se samozřejmě z inverzních transformací.

Použití takovéto tabulky přináší ovšem také specifické problémy. Jedním z problémů, které vznikají, je nutnost použití inverzních podklíčů při dešifrování, protože se klíč nemůže aplikovat před *InvMixColumns* transformací. Další problém pro toto řešení představuje poslední runda šifrování, při které se již neprovádí *MixColumns* transformace ale pouze *SubBytes* transformace. Je tedy nutné buď implementovat samostatnou S-box tabulku nebo tyto hodnoty vypočítat z existující vyhledávací tabulky.

3.2 Návrh datové části

Návrh datové části, tedy části provádějící samotné šifrovaní a dešifrování, byl již nastíněn na začátku této kapitoly. Datové cesty obvodu jsou 32 bitové a výpočet *SubBytes* a *MixColumns* transformací se provádí přes vyhledání ve vyhledávací tabulce T_0. Registr R3 slouží k akumulaci hodnot z této tabulky. K uložení stavu se používají čtyři 32 bitové registry představující řádky stavové matice. Na schematickém obrázku 4 jsou tyto registry souhrnně označeny jako *State*. *ShiftRows* transformace se provádí nad těmito registry v samostatném kroku vždy na začátku rundy. Poslední runda, která neobsahuje *MixColumns* transformaci, se provádí nad samostatně implementovaným S-boxem. K akumulaci hodnot z tohoto S-boxu slouží registr, který je na tomto schématu označený jako R4. K tomuto řešení jsem přistoupil hlavně kvůli operaci dešifrování. Zde se při *InvMixColumns* transformaci používá násobení maticí s vyššími násobky (jak je vidět z rovnice 2.5) a toto násobení se samozřejmě promítlo do hodnot v předpočítané tabulce. Získat z této tabulky hodnotu shodnou s hodnotou v inverzním S-boxu by vyžadovalo použití další logiky provádějící tento výpočet. Navíc pro uložení těchto dvou tabulek nebylo potřeba alokovat žádnou další RAM. Expandované klíče jsou totiž uloženy v samostatné BRAM, která však jimi není plně využita. Tyto S-boxy jsou proto uloženy na nevyužitých pozicích v této paměti.

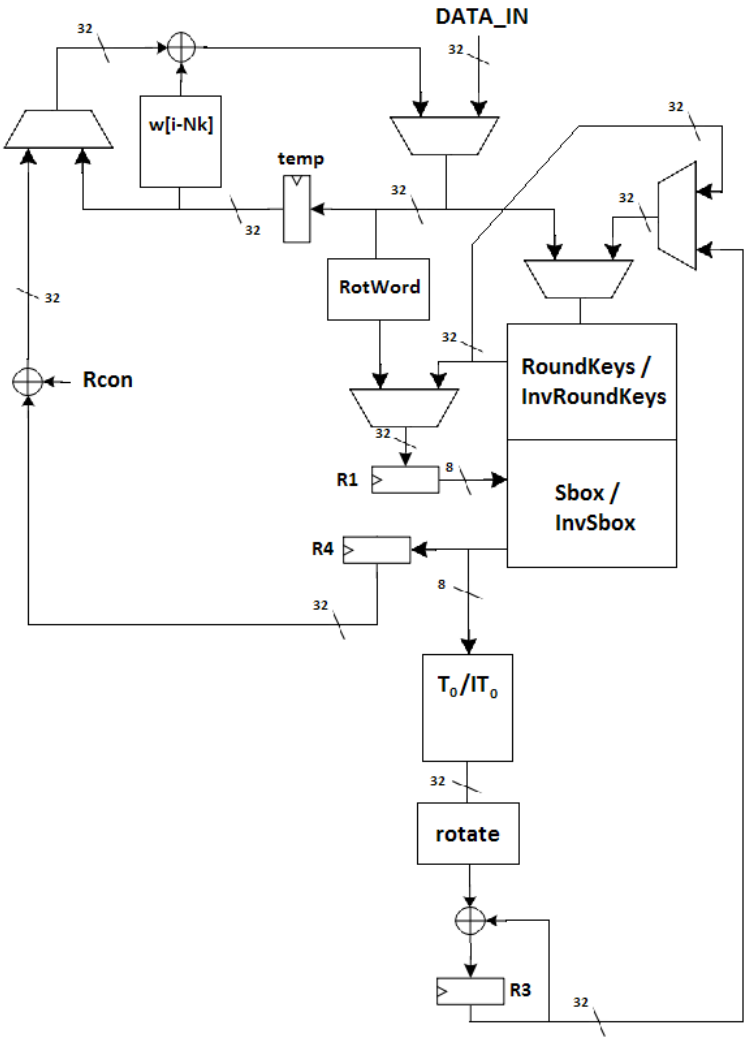


Obrázek 4: Návrh datové části implementovaného obvodu

3.3 Návrh části expandující klíč

Návrh této části je velmi ovlivněn řešeními použitými v datové části obvodu. Použití předpočítané tabulky k výpočtu má totiž negativní dopad na návrh této části. Problémem je především potřeba použití invertovaných podklíčů, které jsou výpočetně náročnější než výpočet neinvertovaných podklíčů. K výpočtu těchto klíčů je potřeba nejprve vypočítat „běžné“ podklíče. Poté, co jsou tyto podklíče vypočteny, jsou všechny byty těchto podklíčů nahrazeny hodnotami z S-box tabulky a poté vyhledány v předpočítané vyhledávací tabulce IT_0 (*InvMixColumns* transformace). Z toho vyplývá, že expanze klíče bude v tomto návrhu potřebovat mnoho výpočetních cyklů. Zde je ovšem nutno podotknout, že pro daný šifrovací klíč bude třeba tuto expanzi provést pouze jednou. Poté budou podklíče uloženy do paměti a samotná operace šifrování/dešifrování již tedy tímto nebude ovlivněna.

Obrázek 5 znázorňuje část pro expanzi klíče. Pro uspoření místa na FPGA by mohly být některé části obvodu sdíleny s datovou částí obvodu, protože obě části nebudou pracovat zároveň.



Obrázek 5: Návrh části expandující klíč

Obě tyto části budou řízeny společnou řídící jednotkou, implementovanou jako stavový automat Moorova typu. Tento automat bude také generovat adresy pro přístup do paměti klíčů a to jak pro jejich zápis v případě expanze klíče, tak i pro jejich čtení v případě šifrování nebo dešifrování dat.

Pro uložení vyhledávacích tabulek T_0 a IT_0 bude použita jednoportová paměť s datovou šířkou 32 bitů. Zařízení Spartan 3 obsahuje syntetizovatelné primitivum RAMB16_S36 [7], které splňuje tyto požadavky. Tato paměť umožňuje uložit až 512 32 bitových hodnot. I když se jedná o paměť RAM, tak se bude tato paměť používat pouze v módu pro čtení. Její obsah bude staticky inicializován při nahrávání konfigurace do zařízení. Pro uložení podklíčů a hodnot S-boxů bude použita jedna dvouportová paměť. První port „A“ bude sloužit opět pouze pro čtení a bude konfigurován s datovou šířkou 8 bitů. Bude sloužit pro uložení hodnot S-boxů. Obsah této paměti bude také inicializován při nahrání konfigurace do zařízení. Druhý port „B“ s datovou šířkou 32 bitů bude sloužit pro uložení expandovaných podklíčů. Tato paměť tedy bude sloužit jak pro čtení, tak pro zápis a nebude nijak inicializovaná. Pro tuto dvouportovou paměť bude použito primitivum RAMB16_S9_S36.

Kapitola 4
Implementace

Tato kapitola popisuje implementační detaily navrženého řešení. Výsledkem práce je šifrovací jednotka ve formě syntetizovatelné entity, kterou lze použít v jakémkoliv dalším projektu. Jednotka se skládá ze tří hlavních částí, které jsou implementovány odděleně. Jedná se o šifrovací blok, blok expandující klíč a řídící jednotku. Díky tomuto oddělení všech jednotek se celý systém dobře programoval, ale za cenu zvýšení celkové velikosti obvodu, protože některé zdroje, které mohly být mezi jednotkami sdíleny, byly implementovány v každém bloku zvlášť.

4.1 Naplnění pamětí

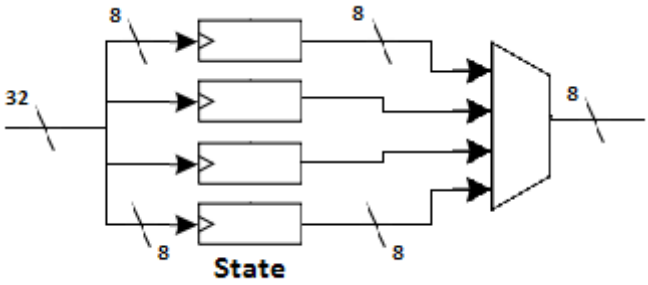
Blokové RAM osazené na zařízení Spartan 3 umožňují zvolit, na jakou hodnotu (případně hodnoty) je jejich obsah při startu inicializován. Výchozí hodnota všech buněk v paměti je 0, ovšem příkazem INIT_xx lze změnit obsah 256 bitů, které se nacházejí na dané adrese.
Následující ukázka ukazuje, jak lze inicializovat prvních 128 bytů paměti.

```
generic map (
     INIT_00 => X"c66363a5f87c7c84ee777799f67b7b8dfff2f20dd66b6bbdde6f6fb191c5c554",
     INIT_01 => X"6030305002010103ce6767a9562b2b7de7fefe19b5d7d7624dababe6ec76769a",
     INIT_03 => X"41adadecb3d4d4675fa2a2fd45afafea239c9cbf53a4a4f7e47272969bc0c05b",
     INIT_04 => X"75b7b7c2e1fdfd1c3d9393ae4c26266a6c36365a7e3f3f41f5f7f70283cccc4f"
    )
```

Tato inicializace paměť nijak nezamyká a je tedy možné ji kdykoliv v zařízení přepsat. Chceme-li takovouto paměť používat jako ROM, musíme dávat pozor, abychom si obsah nepřepsali. Toto platí především pro mnou použitou dvouportovou paměť, kdy část slouží pro uložení vyhledávacích S-boxů, a druhá část pro uložení klíče, protože každý z portů adresuje celou paměť, tedy i tu potenciálně inicializovanou.

4.2 Stavový registr

Stavový registr drží informaci o současném stavu šifry a je tedy nutné, aby měl plnou šířku 128 bitů. Aby se s daty lépe pracovalo, použil jsem místo jednoho 128 bitového registru čtyři 32 bitové registry reprezentující jednotlivé řádky stavové matice. Zapojení registrů je vyobrazeno na obrázku 6.



Obrázek 6: implementace stavového registru

Díky tomuto uspořádání je možné implementovat transformaci *ShiftRows* a *InvShiftRows* pomocí cyklického posunutí v jednotlivých registrech o požadovaný počet míst. Poté po provedení posunutí jsou data reprezentující jednotlivé sloupce matice zarovnaná na stejných pozicích a lze tedy celý sloupec přečíst pouhým přečtením nejvyššího bytu každého registru. Toho v této implementaci není využito a místo toho jsou data vyčítána postupně z každého registru zvlášť a poté zpracována. Výhodou tohoto zapojení je ovšem možnost zapsat nově vypočítaný sloupec bez nutnosti použít další registr na uložení mezistavu, protože po přečtení nejvyššího bytu z posledního registru je celý sloupec zpracován a může být tedy „zapomenut“ (celý registr se může posunout). Tímto vznikne na nejnižších pozicích těchto registrů místo, do kterého může být zapsán nově vypočítaný sloupec. Pro lepší pochopení se lze podívat na následující obrázek, který činnost znázorňuje. Sloupec s hodnotami *[0, 5, A, F]* v tabulce nalevo reprezentuje aktuálně zpracovávaný sloupec. Po jeho zpracování se data posunou tak, jak je znázorněno v pravé tabulce.

3	2	1	0
4	7	6	5
9	8	B	A
E	D	C	F

⇒

0'	3	2	1
5'	4	7	6
A'	9	8	B
F'	E	D	C

Obrázek 7: stav registru před a po zpracování jednoho sloupce

4.3 Rotace výstupu z vyhledávací tabulky

Pro výpočet *MixColumns* pomocí vyhledávacích tabulek jsou potřeba čtyři tabulky s navzájem posunutými hodnotami. Protože však v této implementaci používám pouze jednu tabulku, je potřeba výstup z této tabulky posouvat o určitý počet bytů a tím simulovat vzájemné posunutí tabulek. Správné posunutí je vybráno multiplexorem se čtyřmi vstupy, na které jsou přivedeny správně posunuté výstupy z tabulky. Tato rotace byla podrobněji rozebrána v sekci 3.1.

4.4 Implementace Rcon

Rcon je pole konstant, které se používají při generování podklíčů. Pro klíč s délkou 128 bitů se jedná o pole 10 hodnot s délkou 32 bitů. Jelikož je ale spodních 24 bitů vždy rovno 0, stačí si pamatovat pouze 8 horních bitů. Proto byla pro uložení zvolena paměť, která je generovaná přímo v logice FPGA. Pro paměť takových rozměrů se zdálo toto řešení jako efektivnější, než uložení těchto hodnot do paměti BRAM.

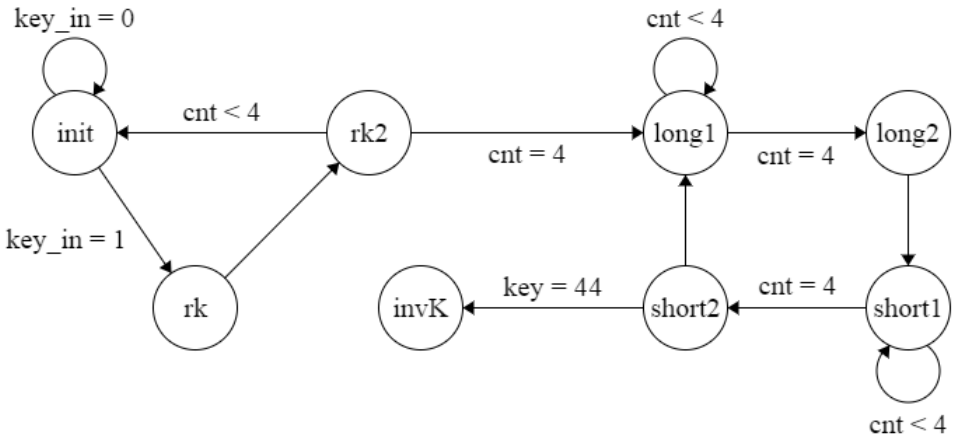
4.5 Řídící jednotka

Celý systém je řízen řídící jednotkou implementovanou jako stavový automat. Jedná se o automat Moorova typu. Tento automat řídí jak generování podklíčů, tak operaci samotného šifrování nebo dešifrování. Z toho důvodu se na řídící jednotku můžeme dívat jako na dva samostatné automaty. Jeden, řídící zpracování šifrovacího klíče, a druhý, řídící šifrování.

4.5.1 Automat řídící generování podklíčů

Tento automat se po inicializaci zařízení nachází ve stavu *init.* V tomto stavu čeká, dokud nepřijde signál oznamující, že na vstupu jsou připravena data obsahující šifrovací klíč. Automat poté přejde do stavu čtení klíče (stavy *rk* a *rk2*). Jelikož je vstup do jednotky pouze 32 bitový, trvá několik taktů, než jsou postupně načtena všechna slova klíče. Když je klíč načten, přejde jednotka k výpočtu podklíčů. Zde automat alternuje mezi prováděním tzv. „dlouhého“ a „krátkého“ cyklu. Dlouhý cyklus sestává ze dvou stavů (*long1* a *long2*) a provádí se zde *SubWord* transformace (substituce bytů za hodnoty z S-box tabulky), přičítání konstanty *Rcon* a přičtení již dříve dekódovaného klíče (viz sekce 3.3). Krátký cyklus sestává také ze dvou stavů (*short1* a *short2*), a provádí se zde pouze přičtení již dříve dekódovaného klíče stejným způsobem jako v případě „dlouhého“ cyklu.

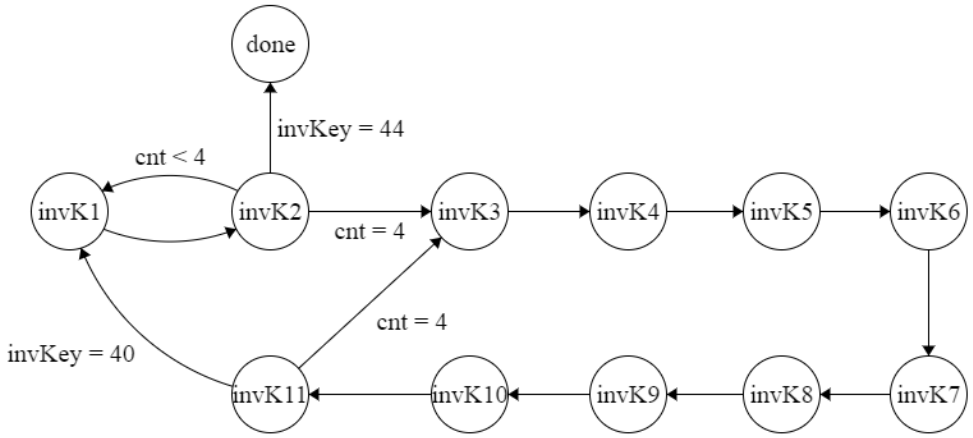
Část zodpovídající za generování podklíčů je zobrazena na obrázku 8.



Obrázek 8: Stavový automat pro generování podklíčů

Po vygenerování všech podklíčů a jejich uložení začne automat s generováním inverzních klíčů (celkem 11 stavů). Generování inverzních klíčů je časově náročnější, proto ve snaze minimalizovat tento čas je počet stavů automatu vyšší než počet stavů pro generování „neinverzních“ podklíčů. Tato část začíná ve stavu *invK1*, kde se zkopírují 4 poslední vygenerované podklíče na začátek paměti obsahující inverzní podklíče. Potom ve stavech *invK3* až *invK8* probíhá inverze ostatních podklíčů, tedy aplikace *SubBytes* a *MixColumns* transformací. Poslední 4 podklíče jsou potom opět pouze zkopírovány do paměti inverzních podklíčů. Generování končí stavem *done*, kde se čeká na data určená k zašifrování nebo dešifrování. Skutečnost, že je generování podklíčů dokončeno, je signalizována vně jednotky nastavením signálu *KEY_READY* na logickou jedna.

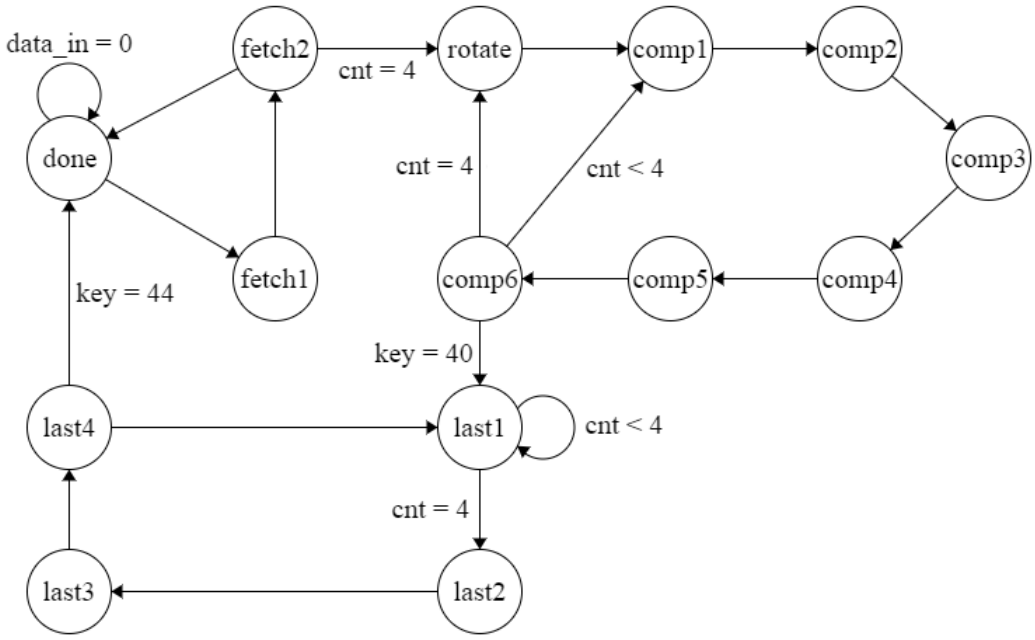
Obrázek 9 zobrazuje část automatu, která zodpovídá za generování inverzních podklíčů.



Obrázek 9: Stavový automat pro generování inverzních podklíčů

4.5.2 Automat řídící proces šifrování

Tento automat řídí šifrování a dešifrování přicházejících dat. Automat začíná ve stavu *done*, kde čeká na příchozí data. Jakmile je do jednotky signalizováno, že jsou data připravena, přesune se automat do stavu *fetch1*. Protože je vstup do jednotky pouze 32 bitový, trvá několik cyklů, než jsou všechna data načtena. Následuje stav *rotate* kde se provede rotace řádků ve stavovém registru. Tato rotace se provádí na začátku každé rundy. Následují stavy *com1* až *comp6* kde se provádí *SubBytes* a *MixColumns* transformace (v tomto řešení implementované formou vyhledání ve vyhledávací tabulce). Tyto stavy se opakují, dokud se neprovede zpracování všech sloupců stavu v dané rundě a všech rund kromě poslední. Po poslední rotaci stavového registru přejde automat do sekce *last*. Ta sestává ze 4 stavů a slouží pro výpočet poslední rundy pomocí jiné vyhledávací tabulky (v poslední rundě chybí *MixColumns* transformace). Výsledná data jsou vyvedena na výstup *DATA_OUT* a připravenost správných dat je vždy signalizována nastavením signálu *DATA_OUT_READY* do logické jedničky. Po dokončení celého výpočtu se automat přesune opět do stavu *done,* kde čeká na další data.



Obrázek 10: Stavový automat šifrovací jednotky

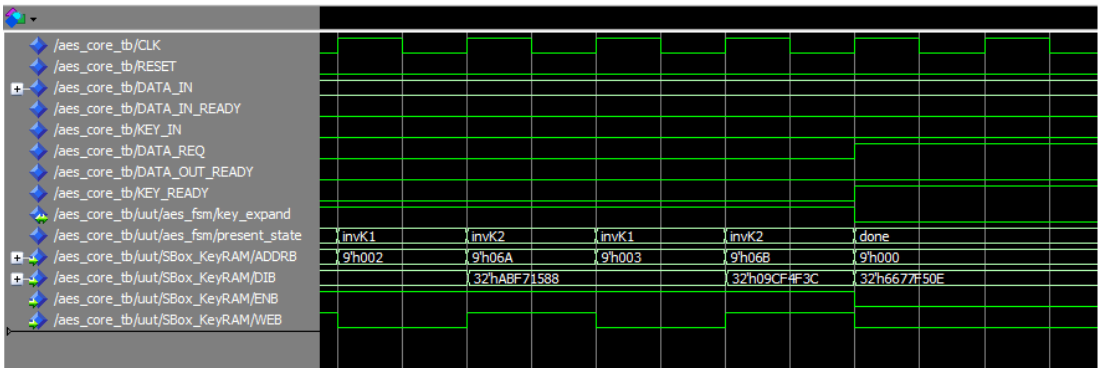
Kapitola 5

Experimenty a naměřené výsledky

Tato kapitola obsahuje výsledky testů, které byly provedeny na implementovaném návrhu k ověření jeho funkčnosti, a obsahuje porovnání s jinými podobně zaměřenými architekturami. Testování bylo provedeno jak v simulovaném prostředí, tak i na reálném hardwaru.

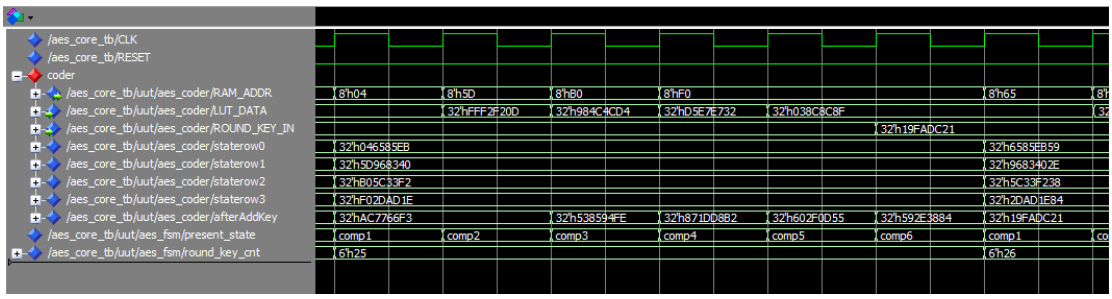
5.1 Výsledky ze simulace

Činnost celé jednotky byla nejprve odsimulována nástrojem ModelSim. Na obrázku 11 můžete vidět dokončení výpočtu podklíčů, jejich ukládání do paměti a signalizaci tohoto stavu a přechod jednotky do stavu, kde čeká na další vstupní data.



Obrázek 11: simulátor ModelSim: dokončení generování podklíčů

Obrázek 12 zachycuje výpočet jednoho sloupce dat ze stavového registru. Jde zde vidět, jak je výpočet *SubBytes* a *MixColumns* transformací implementován pomocí vyhledávání dat v tabulce a jak jsou poté výsledná data zapsána do stavového registru.



Obrázek 12: simulátor ModelSim: výpočet jednoho sloupce stavového registru

Ze simulace lze jednoduchým výpočtem vyčíst, že vygenerování všech klíčů trvalo jednotce 441 cyklů. Z toho 340 cyklů trval výpočet inverzních podklíčů. To potvrzuje počáteční odhady, že tento výpočet v takto implementovaném systému zabere hodně času. Bohužel tento čas je přímo závislý na pouze 8 bitovém přístupu do vyhledávacích tabulek.

Samotné šifrování potom potřebuje 244 cyklů pro vypočítání prvních 32 bitů výsledku. Během dalších 21 cyklů je potom dostupných i zbylých 96 bitů šifry. Za výhodu tohoto designu můžeme považovat to, že operace šifrování i dešifrování zabere stejný počet cyklů a využívá stejných jednotek. Takto implementovaný obvod by tedy mohl být odolný vůči některým druhům postranních útoků.

5.2 Výsledky z experimentů na zařízení FITkit

Pro otestování funkčnosti na reálném zařízení byla jednotka vložena do jednoduchého testovacího obvodu. Tento celek byl otestován na zařízení FITkit s pomocí aplikace QDevKit. Přes terminál této aplikace bylo zjištěno, zda výsledky šifrování a měření odpovídají výsledkům simulace. Protože se jednalo o velmi jednoduchý testovací obvod, byly k dispozici pouze údaje, zda se vypočítané hodnoty shodovali s hodnotami uloženými v zařízení a počet cyklů, indikující dobu výpočtu klíče a šifrovaných hodnot. Další testování by sice bylo možné, předpokládám ale, že pro ověření funkčnosti takovýto obvod postačuje.

5.3 Spotřeba zdrojů a zhodnocení výsledků

Pomocí výpisu z programu Map můžeme zjistit využití zdrojů FPGA. Využití zdrojů pro jednotku implementovanou na cílové zařízení Spartan 3 jsem shrnul v tabulce 2.

Zdroje	Využitých	Dostupných
Slice	525	768
Flip Flop	535	1 536
4 vstupové LUT	859	1 536
RAMB16	2	4

Tabulka 2: Spotřeba zdrojů jednotky na zařízení Spartan 3

Při syntéze implementace byla nástrojem XST stanovena maximální frekvence jednotky na 104 MHz. I když je rozumné předpokládat, že skutečná dosažitelná frekvence bude nižší, jedná se o dobrý výsledek. Tato frekvence dává jednotce teoretickou propustnost 48 Mbitů.

Jelikož cílem bylo implementovat kompaktní obvod, můžeme dát tyto výsledky do relace s již existujícími podobně zaměřenými implementacemi, které byly podrobněji rozebrány v kapitole 2.

Design	Good a spol.	Chodowiec a spol.	Rouvroy a spol.	Tento design
Zařízení	Spartan II XC2S15-6	Spartan II XC2S30-6	Spartan III XC3S50-4	Spartan III XC3S50-4
Datová šířka (bitů)	8	32	32	32
Zabraných slice	124	222	163	525
Použitých BRAM	2	3	2	2
Velikost BRAM (kbitů)	4	4	18	18
Max. frekvence (MHz)	67	60	71	104
Propustnost (Mbitů)	2.2	166	208	48

Tabulka 3: porovnání vlastní implementace s jinými podobnými implementacemi

Z porovnání s ostatními architekturami je patrná vysoká náročnost tohoto obvodu na zdroje. Ta vyplývá především z oddělené implementace generátoru podklíčů a samotné šifrovací části. Přepsáním těchto částí tak, aby více sdílely výpočetní prostředky, by bylo možné ušetřit množství zdrojů, protože tyto jednotky nepracují nikdy současně. Jsem si jistý, že snížení náročnosti na zdroje by bylo značné a zcela by ospravedlnilo tento krok z hlediska snížené maximální frekvence a zvýšené latence obvodu z důvodu přepínací logiky navíc. Dalším problémem této implementace je dle prezentovaných výsledků propustnost, kterou by bylo možno jednoduše zvýšit přidáním druhé vyhledávací tabulky. To by sice znamenalo nutnost použití další BRAM paměti, ale propustnost by se tímto krokem mohla zdvojnásobit. Možností jak mírně zvýšit výkon jednotky při zachování současného rozložení jednotek by také bylo přepracovat existující stavový automat tak, aby se minimalizovaly prodlevy a zvýšilo se využití dostupných zdrojů.

Kapitola 6
Závěr

Cílem této práce bylo navrhnout a implementovat šifrovací jednotku kompatibilní se standardem AES v zařízení FPGA. Jednotka implementovaná v rámci této práce dokáže produkovat šifrovaná data plně vyhovující tomuto standardu, stejně tak jako dokáže dešifrovat data zašifrovaná tímto algoritmem. Funkčnost jednotky byla otestována jak v simulaci, tak na vývojové desce FITkit pomocí jednoduchého testovacího obvodu. Jediným omezením jednotky je omezení maximální podporované délky šifrovacího klíče na 128 bitů.

Na základě prezentovaných výsledků získaných z testování můžeme říct, že se nejedná o nejlepší dostupnou implementaci a pokud by tato měla být konkurenceschopná v souboji s ostatními podobně zaměřenými architekturami, bylo by nutné zapracovat na efektivitě celé jednotky. Musela by být snížena náročnost obvodu na zdroje a zvýšena propustnost tak, jak to bylo prezentováno v závěru předchozí kapitoly. Tímto směrem by se asi taky měly ubírat případné další práce na této implementaci.

Literatura

[1] RODRIGUEZ-HENRÍQUEZ, FRANCISCO et al. *Cryptographic Algorithms on Reconfigurable Hardware*. New York: Springer, 2006. ISBN 0-387-33883-7.

[2] GOOD, Tim a BENAISSA, M. *AES on FPGA from the fastest to the smallest* [online]. 2005, [cit. 10. května 2016]. Dostupné na: `https://www.iacr.org/archive/ches2005/031.pdf`

[3] CHODOWIEC, P., GAJ, K. *Very Compact FPGA Implementations of the AES Algorithm* [online]. 2003, [cit. 10. května 2016]. Dostupné na: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.3697&rep=rep1&type=pdf`

[4] ROUVROY, G., STANDAERT, F. X., QUISQUATER, J. J., LEGAT, J. D. *Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael Very Well Suited for Small Embedded Applications* [online]. 2004, [cit. 10. května 2016]. Dostupné na: `http://perso.uclouvain.be/fstandae/PUBLIS/15.pdf`

[5] NIST. *Federal Information Processing Standards Publication 197* [online]. 2001, [cit. 10. května 2016] Dostupné na: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[6] RIJMEN V., DEAMEN J. *AES Proposal: Rijndael.* [online]. 1999, [cit. 10. května 2016] Dostupné na: `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`

[7] Xilinx, Inc. *Spartan-3 Libraries Guide for HDL Designs* [online]. Dostupné na: `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/spartan3_hdl.pdf`

[8] Xilinx, Inc. *Spartan-3 FPGA Family Data Sheet.* [online]. Dostupné na: `http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf`

Příloha A

Obsah CD

Přiložené CD má následující strukturu:

/aes – adresář testovacího projektu pro QDevKit

/src – adresář se zdrojovými soubory a projektem pro ISE Design Suite

/doc – tato dokumentace v elektronické podobě