VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2020 Sanjin Kek

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ
DEPARTMENT OF TELECOMMUNICATIONS

POSTKVANTOVÁ KRYPTOGRAFIE NA FPGA
POSTQUANTUM CRYPTOGRAPHY ON FPGA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
Sanjin Kek

VEDOUCÍ PRÁCE
SUPERVISOR
Ing. David Smékal

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Sanjin Kek ***ID:*** 186614
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Postkvantová kryptografie na FPGA

POKYNY PRO VYPRACOVÁNÍ:

V rámci bakalářské práce prostudujte možnosti hardwarové implementace na platformě FPGA v jazyce VHDL, seznamte se s FPGA kartami NFB a platformou Xilinx Vivado. Prostudujte postkvantová kryptografická primitiva a schémata, která budou implementována na platformě FPGA. Zaměřte se a popište implementaci postkvantového schéma digitálního podpisu CRYSTALS-Dilithium na platformě FPGA. Cílem bakalářské práce je funkční hardwarová implementace dílčích částí a procedur algoritmu Dilithium (SHAKE, NTT, MakeHint, UseHint, Power2Round a další) a jejich simulací ověřte funkčnost návrhu. Závěrem diskutujte dosažené výsledky.

DOPORUČENÁ LITERATURA:

[1] ALAGIC, Gorjan, a spol. Status Report on the First Round of the NIST Post-Quantum Cryptography Standardization Process. US Department of Commerce, National Institute of Standards and Technology, 2019, DOI: 10.6028/NIST.IR.8240. Dostupné z: https://nvlpubs.nist.gov/nistpubs/ir/2019/NIST.IR.8240.pdf

[2] DUCAS, Léo, et al. Crystals-dilithium: A lattice-based digital signature scheme. IACR Transactions on Cryptographic Hardware and Embedded Systems, 2018, 238-268.

Termín zadání: 3.2.2020 ***Termín odevzdání:*** 8.6.2020

Vedoucí práce: Ing. David Smékal

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Předmětem bakalářské práce je postkvantová kryptografie na FPGA. Teoretická část práce se zabývá seznámením čtenáře s technologií FPGA, základními principy jazyka VHDL, současnou situací na poli postkvantové kryptografie a postkvantovým digitálním podpisovým schématem CRYSTALS-Dilithium. Tomuto schématu je věnována zvýšená pozornost. Jsou rozebrány algoritmy nutné k jeho fungování, jako SHAKE, NTT a dílčí operace pro kompresi veřejného klíče. Praktická část obsahuje hardwarovou implementaci funkce rozšířitelného výstupu SHAKE a dílčích operací algoritmu Dilithium, jako jsou Decompose, UseHint, Power2Round a další.

KLÍČOVÁ SLOVA

Postkvantová kryptografie, FPGA, VHDL, SHAKE, NTT, Digitální podpis, CRYSTALS-Dilithium, Komprese veřejného klíče

ABSTRACT

Subject of this bachelor thesis is postquantum cryptography on FPGA. Focus of theoretical part is to acquaint the reader with FPGA technology, basic principles of VHDL language, current situation in the field of postquantum cryptography and postquantum digital signing scheme CRYSTALS-Dilithium. Increased attention is paid to this scheme. Algorithms needed for function of the scheme, such as SHAKE, NTT and smaller operations used for public key compression, are described. Practical part contains hardware implementation of expandable output function SHAKE and smaller operations, such as Decompose, UseHint, Power2Round and others.

KEYWORDS

Postquantum cryptography, FPGA, VHDL, SHAKE, NTT, Digital signature, CRYSTALS-Dilithium, Public key compression

KEK, Sanjin. *Postkvantová kryptografie na FPGA*. Brno, 2020, 62 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Vedoucí práce: Ing. David Smékal

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Postkvantová kryptografie na FPGA“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Davidu Smékalovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci. Dále bych rád poděkoval paní M.Sc. Sáře Ricci Ph.D. za pomoc s matematickými principy v souvislosti s realizací bakalářské práce. Další poděkování patří Ing. Davidu Kudeláskovi za opakované korektury a přísun pozitivní energie v kritických momentech napříč realizací výsledné podoby bakalářské práce.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Tématem této bakalářské práce je hardwarová implementace dílčích částí postkvantového schématu digitálního podpisu CRYSTALS-Dilithium. Největší předností tohoto algoritmu je odolnost proti útokům kvantovými počítači. Ty v dnešní době sice nejsou ještě dostupné, ale výzkumu na tomto poli je věnována velká pozornost a vznik dostatečně výkonných kvantových počítačů je jenom otázkou času. Už dnes existují algoritmy, připravené pro nasazení na kvantových počítačích, které mají potenciál doslova znemožnit použitelnost mnoha hojně využívaných kryptografických algoritmů. Cílovou platformou pro implementaci je technologie programovatelných hradlových polí (FPGA). Využití této technologie poskytuje výhody rychlosti provádění algoritmu přímo na úrovni hardware a flexibilitu možnosti modifikace, bez nutnosti vyrábět zbrusu nový čip. Cílem bakalářské práce je seznámení s technologií FPGA, jazykem VHDL (jazyk popisující hardware) a funkční hardwarové řešení operací a algoritmů nutných pro realizaci podpisového schématu CRYSTALS-Dilithium.

Teoretické kapitoly práce se zabývají přiblížením technologie FPGA, jazyka implementace, dále stručnou charakteristikou současné situace v oblasti postkvantové kryptografie a obecným popisem principu digitálního podpisu. Čipy FPGA mají v oblasti kryptografie veliký potenciál. Hlavní výhodou je možnost modifikace jejich funkcionality a dnešní modely nacházejí širokou škálu uplatnění na poli vysokorychlostních síťových komunikací, které je s aplikovanou kryptografií úzce spjato. Hojně používaným jazykem pro popis funkcionality těchto čipů je jazyk VHDL. Jedná se o plnohodnotný programovací jazyk, ale vzhledem k tomu, že jeho primárním účelem je popis hardwarových komponent, má specifika s tímto spojená. Jedním z nich je vysoká míra paralelizace prováděných instrukcí, což vyžaduje zcela jiný přístup k problémům, než u vyšších programovacích jazyků. Soudobá postkvantová kryptografie se dělí do pěti odvětví, přičemž každé z nich je postaveno na odlišných matematických principech. Součástí této práce je stručný rozbor těchto pěti přístupů.

V páté kapitole je proveden rozbor podpisového schématu CRYSTALS-Dilithium. Je předveden zjednodušený postup generace klíčů, podepisování a verifikace podpisu. Zvláštní pozornost je věnována kompresi přenášených parametrů a rozboru operací nutných pro úspěšnou implementaci Dilithia, jako jsou funkce rozšířitelného výstupu SHAKE, postavená na algoritmu KECCAK nebo operace NTT, která je variantou diskrétní Fourierovy transformace použitelná v prostoru konečných polí. Kapitola je také věnována dalším dílčím operacím, které jsou předmětem této bakalářské práce.

Poslední dvě kapitoly jsou věnovány samotné implementaci funkce rozšířitelného výstupu SHAKE a dílčích operací. Součástí je názorná ukázka výstupů práce a testování dosažených výsledků.

1 Cílová platforma

Technologie FPGA, neboli programovatelné hradlové pole je jedním z druhů logického integrovaného obvodu. Mezi základní charakteristiky této technologie patří způsob zavedení samotné logiky na obvod. Narozdíl od obvodů ASIC neboli zákaznických integrovaných obvodů, kde jsou logické bloky napevno propojeny vrstvami metalizace, nám technologie FPGA umožňuje tuto logiku měnit pomocí programovatelných propojek. Tento fakt umožňuje této technologii uplatnění v široké škále aplikací. Čip FPGA může zastávat funkci jednoduchého logického prvku, ale i velice složitých integrovaných obvodů. Pro tyto funkční definice se zpravidla používají HDL jazyky jako jsou VHDL nebo Verilog. Technologie je využívána především v situacích, kdy by se ASIC přístup nevyplatil kvůli například malému množství potřebných jednotek nebo v oblasti vývoje, kde je potřeba strukturu čipu měnit. Při využití FPGA také uživatel získá výhodu rychlosti, požadované operace se totiž odehrávají přímo v hardware. Architektura FPGA sestává ze tří základních stavebních kamenů.

- programovatelné logické bloky
- programovatelné I/O bloky
- programovatelné propojky mezi nimi

Programovatelné logické bloky umožňují FPGA čipu implementovat samotné logické operace. Dnešní čipy jsou osazovány kombinací bloků s různými účely, například dedikované paměťové bloky nebo samotné logické komponenty (sčítače, násobiče, apod.). Programovatelné I/O bloky slouží k propojení logických bloků s okolním světem, tedy pro vstupně-výstupní operace. Komunikaci logických bloků s I/O bloky zprostředkovávají programovatelné propojky.

Princip fungování FPGA je sekvenční, je tedy nezbytný hodinový signál. Tento může být generován přímo na FPGA čipu nebo přiveden zvenčí pomocí výše zmiňovaných I/O bloků.

1.1 Návrh

V rámci implementace funkcionality na FPGA čip je nejprve nutné tuto popsat. K tomu se používají výše zmiňované HDL jazyky. V dnešní době se mimo HDL jazyků přistupuje i k jazykům s vyšší mírou abstrakce, jako je například jazyk C nebo Java. Při použití vyšších programovacích jazyků je v závěrečném kroku vždy nutný překlad do HDL jazyka, tento krok je ale pro pohodlí programátorů automatizován.

Dalším krokem je převod HDL do netlist formátu, který popisuje zapojení jednotlivých elektronických komponent, tento proces se nazývá syntéza.

V následujícím kroku se funkcionalita překládá z netlistů. Logické porty jsou v tomto kroku přiřazovány na fyzické elementy FPGA čipu. Poté následuje mapování na samotné bloky FPGA čipu a následné propojení těchto bloků.

Součástí procesu je i verifikace, která slouží k ověření samotné funkcionality. Pokud celý proces proběhne hladce, je nakonec vygenerován kód srozumitelný pro cílový FPGA čip.

1.2 Konfigurace

Tento FPGA čitelný kód je nutné do samotného čipu nahrát. Podle způsobu nahrávání konfigurace rozdělujeme FPGA čipy do dvou kategorií, buď s volatilní nebo nevolatilní pamětí. Obě varianty mají své výhody i nevýhody.

V případě použití volatilní paměti je konfigurace nahrána v paměti typu SRAM. Tento přístup nám umožňuje konfiguraci měnit i za běhu. Nevýhodou ale je, že FPGA čip musí být po zapnutí nakonfigurován, což nějakou dobu trvá. Dalším problémem je, že konfigurace může být z paměti vyčtena, tím pádem je složitější zajistit ochranu intelektuálního vlastnictví.

Při použití nevolatilní paměti je k nahrání konfigurace použita flash paměť, což přináší výhodu většího zabezpečení intelektuálního vlastnictví, nevýhodou je ale omezená možnost rekonfigurace.

1.3 Síťová karta NFB-200G2QL

V rámci realizace této práce bude použita síťová karta společnosti NETCOPE TECHNOLOGIES NFB-200G2QL. Jedná se o vysokorychlostní síťovou kartu osazenou FPGA čipem Virtex UltraScale+ společnosti Xilinx. Karta disponuje dvěma síťovými rozhraními typu QSFP28, která slouží pro připojení vysokorychlostních optických transcieverů. Se serverem komunikuje karta pomocí PCI Express x16 rozhraní třetí generace. Karta je stavěná na rychlosti komunikace až 200 Gb/s, je také osazena pamětí SRAM, jedná se tedy o FPGA řešení s volatilní pamětí. Karta je díky své rychlosti použitelná pro širokou škálu využití. Hodí se například pro předzpracování síťového provozu, vývoj hardwarově akcelerovaných protokolových řešení nebo elektronické finanční transakce, kde je zapotřebí minimální zpoždění.

Informace ke zpracování této kapitoly pocházejí ze zdrojů [1], [2] a [3].

2 Jazyk implementace

Jazyk VHDL je plnohodnotný programovací jazyk. Oproti vyšším programovacím jazykům jsou ale instrukce psané ve VHDL zpravidla prováděny současně. Toto chování vyplývá z primárního účelu jazyka, kterým je popisování hardwarových komponent, jako jsou například FPGA čipy.

2.1 Specifika VHDL kódu

- není citlivý na rozdíl mezi velkými a malými písmeny
- není citlivý na bílé znaky
- komentáře jsou uvozeny symbolem "–"a končí koncem řádku
- převážně není vyžadováno používání závorek
- řádky s vykonávaným kódem jsou zakončeny znakem ";"
- rozdíly v konstrukci cyklů oproti vyšším programovacím jazykům
- omezení možných znaků obsažených v identifikátorech

Necitlivost na velká či malá písmena nebo bílé znaky může být pozitivem při psaní kódu, kdy není nutné dávat pozor na přesné dodržení těchto pravidel jako u jiných programovacích jazyků, například Pythonu. Tento fakt však může mít i negativní důsledky na čitelnost kódu. Stejná situace je i u závorek, kdy jejich použití sice není jazykem vyžadováno, ale může být přínosem pro čitelnost kódu.

Na rozdíl od většiny vyšších programovacích jazyků se v případě jazyka VHDL váže ke konstrukci cyklů nebo větvení v kódu několik specifik. Při použití podmínky uvozené rezervovaným slovem `if` musí následovat rezervované slovo `then`. Každá takováto konstrukce pak musí být zakončena pomocí `end if;`. Další rezervované slovo, které se váže k této konstrukci, je `else`, v jazyce VHDL má ale tvar `elsif`. Stejně jako v případě podmínek, je nutné uzavírat i jiné konstrukce jako `case` pomocí `end case;` a `loop` pomocí `end loop;`.

Při vymýšlení identifikátorů pojmenovávajících například proměnné je nutné dbát na omezení, které umožňuje v těchto situacích použít pouze velká či malá písmena, číslice a znak "_". Identifikátory také musí začínat písmenem, naopak nesmí končit podtržítkem.

2.2 Základní bloky a struktura VHDL kódu

Jazyk VHDL využívá pro popis funkce hardwarových komponent takzvaný Black-Box přístup, který umožňuje jednoduché znovupoužití již definovaných funkčních

celků, a také zjednodušuje proces návrhu z hlediska přehlednosti. Základními bloky jazyka jsou entita, označovaná `entity` a architektura, označovaná `architecture`.

Entity slouží pro popis funkčního celku směrem ven, tedy kolik a jakých vstupů a výstupů daný celek má. K deklaraci vstupu nebo výstupu entity slouží rezervované slovo `port`. U každého z nich je potřeba definovat, zda se jedná o vstup nebo výstup, případně vstupně-výstupní port pomocí rezervovaných slov `in`, `out` nebo `inout`. Každý port také musí být pojmenován a mít definovaný datový typ, se kterým pracuje.

Výpis 2.1: Příklad entity v jazyce VHDL

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL
3
4  entity ukazka is
5  port(
6      input1 : in std_logic;
7      input2 : in std_logic;
8
9      output : out std_logic
10 );
11 end ukazka;
```

Výpis 2.1 ukazuje jednoduchou entitu pojmenovanou ukázka, která má dva vstupní porty a jeden port výstupní. Porty pracují s datovým typem `std_logic`, který nabývá standardních hodnot bitové logiky, ale navíc umožňuje použití několika zvlášť definovaných stavů, které se váží k poli hardwarového designu na této úrovni, například stav 'Z', který značí vysokou impedanci. Tento datový typ je definován v balíčku `IEEE.STD_LOGIC_1164.ALL`, který je součástí knihovny `IEEE`. Pro jeho použití je tedy nutné deklarovat příslušnou knihovnu a balíček na začátku souboru, obdobně jak je tomu s hlavičkovými soubory a knihovnami v programovacích jazycích jako například C++ nebo Java. Z ukázky je patrný Black-Box přístup, není zde definováno žádné vnitřní uspořádání komponenty, pouze jakým způsobem komunikuje se svým okolím.

Architektura popisuje vnitřní strukturu funkčního celku, tedy co a jakým způsobem daný celek dělá. Při návrhu architektury je nutné zohlednit účel a cíl, v jazyce VHDL je totiž možné docílit stejné funkcionality různými cestami. Optimalizovat kód lze pro různé účely, například pro rychlost nebo pro nejúspornější možné řešení z hlediska počtu využitých fyzických komponent. V deklaraci architektury je na prvním místě deklarace proměnných, signálů, případně konstant, k čemuž slouží rezervovaná slova `variable`, `signal` a `constant`. Proměnné slouží k uchování infor-

mace, mají název a definovaný datový typ. Konstanty jsou proměnné, jejichž hodnotu nelze po přiřazení nadále měnit. Signály jsou reprezentacemi fyzických spojení mezi různými funkčními celky. Stejně jako proměnné nebo konstanty mají název a definovaný datový typ, na rozdíl od nich jsou ale reprezentacemi skutečných vodivých elementů na čipu. Následuje popis samotné funkce, kde je možné například pomocí logických operátorů pracovat s daty na portech entity, přiřazovat hodnoty proměnným nebo signálům pomocí k tomu určených operátorů a pomocí výstupního portu entity poslat výsledek dál.

Výpis 2.2: Příklad architektury v jazyce VHDL

```
1 architecture Behavioral of ukazka is
2
3 variable var : integer;
4
5 begin
6     var := 3;
7     output <= input1 and input2;
8 end Behavioral;
```

Předmětem výpisu 2.2 je ukázka jednoduché architektury v jazyce VHDL. Samotná funkce modulu je popsána mezi klíčovými slovy `begin` a `end`. Nad tímto tělem architektury je prostor pro deklaraci konstant, proměnných nebo signálů se kterými architektura pracuje. V případě ukázky je zde deklarována proměnná `var` typu `integer`, tedy celé číslo. V těle architektury je do proměnné uloženo číslo 3. Na dalším řádku architektura pracuje s porty entity, kdy na výstup posílá výsledek operace `and` provedené na dvou vstupech.

2.3 Vývojové prostředí použité pro realizaci

K implementaci návrhu podpisového schématu CRYSTALS-Dilithium je využito vývojové prostředí ISE WebPACK Design Suite 14. Jedná se o bezplatné, široké společnosti dostupné řešení společnosti Xilinx, která se zabývá mimo jiné výrobou a vývojem FPGA čipů. Jde o plnohodnotnou alternativu k prostředí Vivado Design Suite HLx od stejné společnosti, které je určeno pro komerční účely a jedinou bezplatnou variantou jeho použití je třiceti denní zkušební licence. Prostředí ISE poskytuje všechny nástroje nutné pro návrh FPGA řešení v jazyce VHDL. Pro účely testování je použita aplikace ISim, která je integrovaná v prostředí ISE. Samotné prostředí je realizováno formou virtuálního počítače s operačním systémem Oracle Linux Server 6.4, který je distribuován formou .ova souboru.

Informace ke zpracování této kapitoly pocházejí ze zdrojů [4] a [5].

3 Postkvantová kryptografie

Postkvantová kryptografie je souhrnný název pro oblast kryptografie, která se zabývá odolností proti útokům kvantovými počítači. V dnešní době není technologie kvantové výpočetní techniky zatím dostupná. Riziko, které potenciální dostupnost této technologie v budoucnosti představuje, je pro dnešní kryptografii natolik ohrožující, že práce na postkvantové kryptografii probíhá již dnes.

Běžné výpočetní systémy využívají k reprezentaci informace binární soustavu. V této číselné soustavě jsou k dispozici číslice 0 a 1, každý řád binárního čísla reprezentuje jeden bit informace. Počet bitů, se kterými může výpočetní systém pracovat tedy přímo ovlivňuje množství informace, které může daný systém zpracovávat.

Kvantové výpočetní systémy reprezentují informaci pomocí qubitů, které se od bitů zásadně liší. Informace je reprezentována pomocí superpozice dvou hodnot, které mohou nabývat známých binárních stavů s různými pravděpodobnostmi. Tento přístup k výpočetní technice bude mít v budoucnu veliký dopad na oblasti kryptografie, které dnes spoléhají na matematické problémy faktorizace nebo problém diskrétního logaritmu.

V dnešní době používané kryptosystémy spoléhají v mnoha případech na složitost matematických problémů faktorizace velkých čísel nebo problému diskrétního logaritmu, které je s pomocí Shorova algoritmu možné na dostatečně výkonném kvantovém počítači řešit v polynomiálním čase. V současnosti používané protokoly jako například RSA nebo ECDH jsou tímto algoritmem přímo ohroženy a s nástupem kvantových počítačů budou nepoužitelné.

Dalším algoritmem, který hrozí dnešním kryptosystémům je Groverův algoritmus, který je použitelný pro hledání inverzní funkce. To představuje hrozbu pro symetrické kryptosystémy nebo jednosměrné kompresní funkce používané při tvorbě otisků dat. Nebezpečí reprezentované tímto algoritmem však není natolik výrazné jako v případě Shorova algoritmu. V současnosti používané kryptografické systémy jako symetrická šifra AES nebo hashovací funkce SHA-3 mohou být upraveny pro zvýšenou odolnost proti Groverovu algoritmu. V případě symetrických kryptosystémů pomůže v tomto směru větší velikost klíče, v případě hashovacích funkcí větší velikost výstupu. Z hlediska odolnosti vůči kvantovým útokům lze výzkum kryptografie kategorizovat do pěti základních odvětví.

- Kryptografie založená na teorii kódování
- Kryptografie založená na hashovacích funkcích
- Kryptografie založená na polynomiálních rovnicích
- Kryptografie založená na supersingulárních eliptických křivkách
- Kryptografie založená na mřížkách

3.1 Typy postkvantové kryptografie

Kryptografie založená na teorii kódování

Teorie kódování popisuje přenos zpráv a zachování jejich integrity při přenosu libovolným komunikačním kanálem. Příkladem je v dnešní době hojně využívaný Hammingův kód, který je zástupcem z rodiny lineárních samoopravných kódů. Pro odolnost proti kvantovým útokům lze použít Goppa kód. Jedná se o lineární samoopravný kód, který je konstruován pomocí algebraické křivky nad konečným polem. Mezi jeho výhody patří rychlost použití, nevýhodou je velikost klíčů. Bezpečnost systémů využívajících tohoto přístupu je zajištěna pouze při dodržení určitých nutných parametrů. Mezi postkvantové kryptosystémy využívající teorie kódování patří například asymetrický algoritmus McEliece.

Kryptografie založená na hashovacích funkcích

Hashovací funkce je matematická funkce, jejíž nejdůležitější vlastností je jednosměrnost. Vstupem funkce je řetězec libovolné délky, ze kterého se na výstupu stává řetězec konstantní délky. Jednosměrností je myšlen fakt, že není možné jednoduchým způsobem tuto funkci invertovat. U hashovacích funkcí je důležitým faktorem pojem kolize, který označuje situaci, kdy při různých vstupech dosáhneme stejného výstupu. Toto chování je sice nežádoucí, ale bohužel z principu nevyhnutelné. Z hlediska odolnosti proti kvantovým útokům je potřeba u těchto funkcí zvětšit velikost výstupních řetězců.

Kryptografie založená na polynomiálních rovnicích

Tento přístup k postkvantové kryptografii využívá algebraickou geometrii a s ní spojenou náročnost řešení soustavy rovnic o více neznámých nad konečným polem. Pro účely kryptografie jsou používány jednosměrné funkce se zadními vrátky, na poli postkvantové kryptografie jsou významné rovnice skrytých polí, které jsou založeny na polynomech různých velikostí nad konečným polem. Bezpečnost zajišťuje problém řešení vícerozměrných kvadratických rovnic.

Kryptografie založená na supersingulárních eliptických křivkách

Dnešní kryptografie, která využívá eliptických křivek je založena na problému diskrétního logaritmu. Důležitou vlastností pro odolnost proti kvantovým útokům je izogenita. Ta zobrazuje body jedné křivky do druhé, se zachováním vrcholů. Výhodou tohoto přístupu je malá velikost klíčů, jejich generování je ale výpočetně náročné. Mezi kvantově odolné protokoly využívající tento přístup patří například protokol SIDH, který je modifikovanou verzí protokolu ECDH.

Kryptografie založená na mřížkách

Mřížku si lze představit jako množinu uspořádaných bodů v n-rozměrném prostoru. V případě dvourozměrného prostoru mluvíme o vektorovém prostoru s omezením násobení vektorů celými čísly. V této oblasti kvantově odolné kryptografie jsou využívány dva základní matematické problémy.

Prvním z nich je SVP. Tento problém spočívá v nalezení nejkratšího nenulového vektoru mřížky, což je složité z důvodu existence velkého množství bází. Samotná mřížka je také definována pomocí bázových vektorů mnohem delších, než je ten nejkratší. Druhým problémem je CVP, který spočívá v nalezení nejbližšího vektoru k libovolnému vektoru mřížky. Ten oproti SVP může být nulový, ale musí být součástí mřížky.

Na těchto dvou problémech jsou postaveny další dva problémy relevantní pro postkvantovou kryptografii. Problém LWE vypadá následovně. Jsou zapotřebí dvě matice $\boldsymbol{X}$ a $\boldsymbol{Y}$. Následovně s pomocí matice náhodných čísel $\boldsymbol{A}$ spočteme $\boldsymbol{B} = \boldsymbol{AX} + \boldsymbol{Y}$. Z matice $\boldsymbol{X}$ se stává soukromý klíč, veřejným jsou matice $\boldsymbol{A}$ a $\boldsymbol{B}$. Výpočty se odehrávají v množině reálných čísel. Problém RLWE je obdobný, místo reálných čísel je ale použito polynomiálních okruhů nad konečným polem. Tímto přístupem se snižuje výpočetní náročnost. Algoritmus CRYSTALS - Dilithium, jehož implementace je předmětem této práce, patří do tohoto odvětví postkvantové kryptografie.

Informace pro zpracování této kapitoly pocházejí ze zdrojů [6], [7], [8] a [9].

4 Digitální podpis

Jedná se o aplikaci asymetrické kryptografie, která je dostupná široké veřejnosti. Existence digitálního podpisu a jeho používání je podchyceno nejen v českém právním řádu, dnes je již nedílnou součástí elektronického styku například s bankami, ale právě díky jeho uznávání, i se státními institucemi. Je používán ve formě certifikátu a má tři základní funkce.

- Zaručení autentičnosti odesílatele
- Zaručení integrity zprávy
- Zaručení nepopiratelnosti

Autentičnost se v tomto kontextu váže k nepopiratelnosti. Odesílatel digitálně podepsané zprávy nemůže popřít fakt, že zprávu odeslal právě on. Naopak příjemce digitálně podepsané zprávy má jistotu, že zprávu odeslal právě odesílatel. Pro splnění těchto funkcí je ale nutnou podmínkou tajnost soukromého klíče. Pokud dojde k jeho ztrátě nebo odcizení, samotná kryptografická primitiva, na kterých jsou schémata digitálních podpisů postavená už v takové situaci autentičnost ani nepopiratelnost nedokáží zaručit. Tvar digitálního podpisu přímo závisí na obsahu zprávy, pokud by tedy potenciální útočník zprávu zachytil, nějakým způsobem ji zmanipuloval a následně poslal příjemci, byl by tento fakt patrný při ověřování digitálního podpisu. Tato fakta přímo vyplývají z matematického postupu podepisování zprávy, proces podepisování a verifikace podpisu vypadá následovně.

- Odesílatel zprávy podepíše zprávu s pomocí svého soukromého klíče
- Odesílatel pošle zprávu společně s digitálním podpisem
- Příjemce ověří přijatý digitální podpis pomocí přijaté zprávy a veřejného klíče odesílatele

Tento proces, tak jak je popsán by byl pro reálné využití nevhodný. Pro zvýšení zabezpečení zprávy je možné ji nejprve zašifrovat vhodným kryptografickým systémem, v praxi se také nepodepisuje samotná zpráva, ale pouze její otisk, který je získán pomocí jednosměrné hashovací funkce. Podepisování otisků zpráv má však také svá úskalí. Pokud by pro jeho získání byla použita zastaralá hashovací funkce se známými kolizemi, mohlo by to ohrozit i samotný digitální podpis.

5 Algoritmus CRYSTALS - Dilithium

Algoritmus slouží k tvorbě digitálních podpisů a jejich verifikaci, řadí se mezi postkvantové kryptografické algoritmy založené na mřížkách a je součástí druhého kola soutěže institutu NIST, jejímž předmětem je standardizace postkvantové kryptografie. Algoritmus je založen na modifikovaném problému RLWE. Autory tohoto algoritmu jsou Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seidler a Damien Stehlé. Cílem autorů bylo zohlednit čtyři základní kritéria[10].

- Jednoduchost bezpečné implementace
- Konzervativita výběru parametrů
- Minimalizace velikosti veřejného klíče a digitálního podpisu
- Modularita

Ke generování náhodnosti pro zabezpečení v kompaktních algoritmech založených na mřížkách bylo v minulosti používáno diskrétní Gaussovské distribuce. Autoři Dilithia z důvodů složitosti touto cestou dosáhnout výsledků odolných vůči útokům postranními kanály zvolili cestu použití uniformního vzorkování.

Vzhledem k časté nutnosti přenosu páru veřejného klíče i digitálního podpisu je v algoritmu kladen důraz na co nejmenší velikost těchto parametrů. Za tímto účelem je algoritmus postaven tak, že není nutné veřejný klíč ani digitální podpis posílat celý.

Modularity algoritmu je dosaženo tím, že polynomiální okruh nad konečným polem zůstává vždy stejný. Pro míru zabezpečení implementace algoritmu není tedy důležitý samotný okruh, ale počet operací nad ním. Dilithium používá polynomiální okruh $\mathbb{Z}_q[X]/(X^n + 1)$, s parametry $q = 2^{23} - 2^{13} + 1$ a $n = 256$. Dvě pro algoritmus důležité operace jsou funkce rozšiřitelného výstupu a násobení v polynomiálním okruhu. K první jsou použity funkce SHAKE-128 a SHAKE-256, k druhé je využito NTT[10].

5.1 Ukázka zjednodušeného postupu

Prvním krokem algoritmu je generování páru veřejného a soukromého klíče (5.1). Výchozím krokem je matice $\boldsymbol{A}$, která má v ukázce rozměr 5×4. Elementy matice $\boldsymbol{A}$ jsou polynomy z okruhu $R_q = \mathbb{Z}_q[X]/(X^n+1)$, přičemž parametry q a n jsou vždy stejné. Dalším krokem je výběr dvou krátkých vektorů $\boldsymbol{s_1}$ a $\boldsymbol{s_2}$, následuje výpočet

vektoru $\boldsymbol{t}$. Výstupem jsou veřejný klíč, který se skládá z matice $\boldsymbol{A}$ spolu s vektorem $\boldsymbol{t}$ a soukromý klíč, složený z matice $\boldsymbol{A}$ a vektorů $\boldsymbol{t}$, $\boldsymbol{s_1}$ a $\boldsymbol{s_2}$.

$$\begin{aligned}
&\boldsymbol{A} \leftarrow R^{5\times 4} \\
&\boldsymbol{s_1} \leftarrow S_5^4,\ \boldsymbol{s_2} \leftarrow S_5^5 \\
&\boldsymbol{t} = \boldsymbol{A}\boldsymbol{s_1} + \boldsymbol{s_2} \\
&pk = (\boldsymbol{A},\ \boldsymbol{t}),\ sk = (\boldsymbol{A},\ \boldsymbol{t},\ \boldsymbol{s_1},\ \boldsymbol{s_2})
\end{aligned} \tag{5.1}$$

Proces podepisování (5.2) začíná výběrem krátkého vektoru $\boldsymbol{y}$, s pomocí kterého je v dalším kroku znásobením s maticí $\boldsymbol{A}$ získán vektor $\boldsymbol{w}$. Ten vstupuje spolu se zprávou M do hashovací funkce $H()$. Notace $High$ značí fakt, že vektor $\boldsymbol{w}$ nevstupuje do funkce celý, ale jenom jeho bity horních řádů. Výsledkem této operace je výzvový polynom c, který se skládá z 60 $\pm$ jedniček, zbytek jsou nuly. S pomocí parametrů $\boldsymbol{y}$, c a $\boldsymbol{s_1}$ je spočten vektor $\boldsymbol{z}$. Pokud vektor $\boldsymbol{z}$ nebo výraz $\boldsymbol{w} - c\boldsymbol{s_2}$, u kterého jsou posuzovány spodní řády, odhalují skrytou informaci, je potřeba proces podepisování restartovat. V situaci, kdy tomu tak není, je získán podpis, kterým jsou vektor $\boldsymbol{z}$ a výzvový polynom c.

$$\begin{aligned}
&\boldsymbol{y} \leftarrow S_\gamma^4 \\
&\boldsymbol{w} = \boldsymbol{A}\boldsymbol{y} \\
&c = H(High(\boldsymbol{w}),\ M) \in B_{60} \\
&\boldsymbol{z} = \boldsymbol{y} + c\boldsymbol{s_1} \\
&\texttt{If } \|\boldsymbol{z}\|_\infty > \gamma - \beta \texttt{ or } \|Low(\boldsymbol{w} - c\boldsymbol{s_2})\|_\infty > \gamma - \beta, \texttt{ restart} \\
&sig = (\boldsymbol{z},\ c)
\end{aligned} \tag{5.2}$$

K verifikaci podpisu (5.3) je potřeba spočítat výzvový polynom c', k čemuž je potřeba znát horní řády vektoru $\boldsymbol{w}$, které verifikátor nemá k dispozici. Druhá část podmínky na konci podepisování ale zajišťuje, že výraz $\boldsymbol{w} - c\boldsymbol{s_2}$, u kterého se posuzují pouze bity spodních řádů, posuzovaný při procesu podepisování a výraz $\boldsymbol{A}\boldsymbol{z} - c\boldsymbol{t}$, který má k dispozici verifikátor, jsou ekvivalentní. Je tedy možné z dostupných parametrů ověřit, zda se oba výzvové polynomy rovnají a pokud ano, je podpis úspěšně ověřen.

$$\begin{aligned}
&c' = H(High(\boldsymbol{A}\boldsymbol{z} - c\boldsymbol{t}),\ M) \\
&\texttt{If } \|\boldsymbol{z}\|_\infty \leq \gamma - \beta \texttt{ and } c' = c, \texttt{ accept}
\end{aligned} \tag{5.3}$$

5.2 Komprese veřejného klíče

Výše popsaný postup má podstatný nedostatek. Součástí veřejného klíče je matice $\boldsymbol{A}$, která se skládá z dvaceti polynomů a její reprezentace přidává na velikosti veřejného klíče. Autoři tento problém řeší tím, že matici $\boldsymbol{A}$ generují ze zrna, k čemuž je využita hashovací funkce SHAKE-128. Toto zrno je přidáno do soukromého klíče, a je použito spolu se zprávou ke generovaní náhodnosti při definování vektoru $\boldsymbol{y}$ v procesu podepisování.

Dalšího zmenšení veřejného klíče je dosaženo rozkladem vektoru $\boldsymbol{t}$ na části obsahující vyšší a nižší řády $\boldsymbol{t_1}$ a $\boldsymbol{t_0}$, přičemž do veřejného klíče je vložena pouze část $\boldsymbol{t_1}$. Při procesu verifikace je potřeba spočítat $High(\boldsymbol{Az} - c\boldsymbol{t}) = High(\boldsymbol{Az} - c\boldsymbol{t_1}2^{14} - c\boldsymbol{t_0})$, ale nyní verifikátor nemá k dispozici hodnotu $-c\boldsymbol{t_0}$. Tento problém je řešen přidáním zbytků z přičítání $-c\boldsymbol{t_0}$ do samotného podpisu, verifikátor tedy může svůj výsledek opravit.

5.3 Důležité operace pro implementaci algoritmu

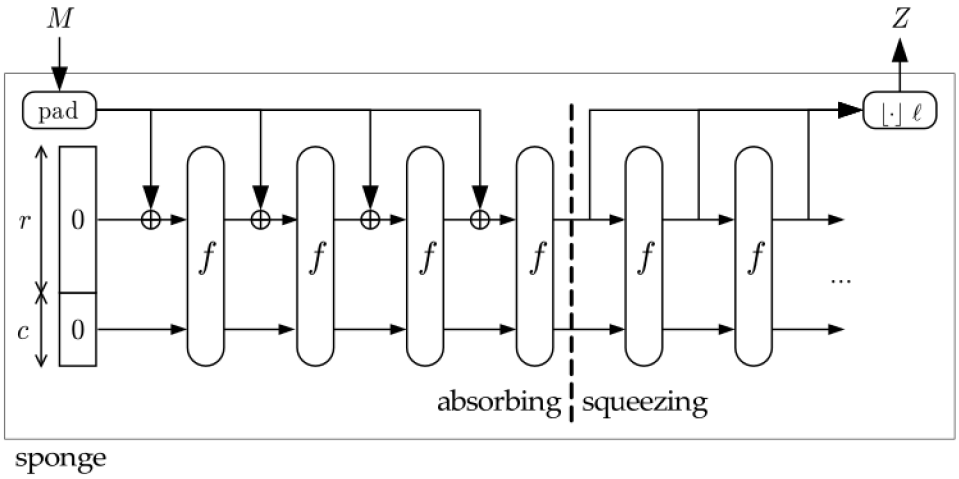
Operace nutné pro úspěšnou implementaci Dilithia jsou SHAKE a NTT. Algoritmus využívá také řadu dílčích algoritmů za účelem snížení velikosti přenášených parametrů nutných pro ověření podpisu.

5.3.1 SHAKE

Jedná se o funkci rozšiřitelného výstupu, která vychází ze specifikace FIPS PUB 202 a je založena na algoritmu KECCAK, který se stal vítězem soutěže institutu NIST o standard SHA-3. Algoritmus SHAKE je postaven na principu houby. [11]

SPONGE

Funkce slouží k transformaci vstupního binárního řetězce na výstupní binární řetězec libovolné délky. Její chování je definováno počtem parametrů, formální zápis má tvar `SPONGE[f,pad,r](M,l)`. Parametr `f` značí funkci, která bude uvnitř houby použita k transformaci binárního řetězce, `pad` je pravidlo pro výplň, `r` je přenosová rychlost, `M` je samotný bitový řetězec, který bude transformován a `l` je požadovaná délka výstupu.



Obr. 5.1: Konstrukce houby [12]

Na obrázku 5.1 je schematicky znázorněno chování funkce SPONGE. Na levé straně je patrný parametr `r` spolu s parametrem `c`, což je kapacita. Součtu délek těchto dvou parametrů se říká šířka `b` a udává velikost bloků se kterými pracuje funkce `f`. Bitový řetězec `M` vstupuje do houby a podle pravidla `pad` je vyplněn do násobku přenosové rychlosti. Poté je rozdělen do bloků velikosti `r` a tyto bloky postupně vstupují do funkce `f`, přičemž procházejí logickou operací XOR spolu s řetězcem velikosti `r`, který je naplněn nulami, stejně jako řetězec velikosti `c`. Tento proces se nazývá absorpce, a trvá dokud jsou k dispozici části řetězce `M`. Po absorbování celého řetězce přechází houba do procesu vymačkávání, kdy jsou výstupy funkce `f` vedeny na výstup houby. Počet iterací tohoto procesu je definován parametrem `l`. Bloky výstupu jsou nakonec spojeny do bitového řetězce `Z`. V případě algoritmu SHAKE je jako funkce `f` použita permutace `KECCAK-p[1600,24]`.

KECCAK-p[1600,24]

Formální zápis permutace má tvar `KECCAK-p[b,n](S)`, kde parametr `b` udává délku vstupního řetězce `S` a parametr `n` udává počet kol permutace. V případě algoritmu SHAKE jsou tyto parametry předem dané specifikací. Hlavní charakteristikou této permutace je převod vstupního binárního řetězce `S` do trojrozměrného stavového pole rozměru $5 \times 5 \times 64$. Nad tímto stavovým polem jsou v rámci permutace prováděny dílčí operace, které budou předmětem implementace.

5.3.2 NTT

Jedná se o zobecnění diskrétní Fourierovy transformace pro použití v prostoru konečných polí. Operace je v rámci algoritmu použita k snížení výpočetní náročnosti

násobení polynomů. Diskrétní Fourierova transformace slouží k převodu časových průběhů do frekvenčního spektra. Oproti klasické Fourierově transformaci, kdy vstupem je spojitá funkce, vstupem diskrétní podoby této operace je množina hodnot. Základními stavebními kameny je vektor vstupních hodnot, transformační matice a vektor výstupních hodnot. Toto platí i pro NTT, ale hodnotami, na kterých je operace prováděna jsou místo komplexních čísel elementy uzavřeného celočíselného pole. Stejně tak transformační matice, která v klasické podobě obsahuje mocniny čísla ω, které splňuje podmínku $\omega^n = 1$ a zároveň $\omega^k \neq 1$ pro $1 \leq k \leq n$. Nejčastějším vyjádřením je $\omega = \mathrm{e}^{-2\pi i/n}$, kde n značí délku vstupního vektoru. Notace je v případě NTT stejná, ale vyjádření má tvar $\omega \equiv g^k \mod N$. Vstupem je $n-$tice elementů, k je celé číslo vyhovující $N = kn + 1$ a N je prvočíslo vyjadřující velikost pole nad kterou je operace prováděna.[13]

Operace NTT je výpočetně náročná. Jedná se o sumu součinů elementů vstupní $n-$tice s elementy transformační matice, složitost této operace je kvadratická. Pro dopřednou transformaci je použit algoritmus Cooley-Tukey, který využívá symetrie v transformační matici k rozdělení transformace na dvojnásobný počet operací s polovičním objemem dat iterativní metodou, tímto je dosaženo snížení složitosti z n^2 na $n\ log\ n$. K násobení výsledků dopředné transformace tvůrci Dilithia využívají Montgomeryho násobení a ke zpětné transformaci algoritmus Gentleman-Sande.

5.3.3 Operace pro kompresi veřejného klíče

Hlavní předností algoritmu CRYSTALS-Dilithium oproti ostatním kryptografickým algoritmům založeným na mřížkách je komprese veřejného klíče. K tomuto účelu je použita řada dílčích algoritmů, které se starají o dělní reprezentací polynomů na vyšší a nižší řády. Dalším předmětem těchto pomocných algoritmů je tvorba nápověd, které jsou odesílány za účelem úspěšného ověření podpisu namísto celých parametrů nutných pro tyto výpočty.

Tvůrci Dilithia zavádějí termín centralizované modulární redukce, která má symbol $\mod^{\pm}$ a její definice zní následovně. Hodnota $r' = r \mod^{\pm} \alpha$ je jedinečným elementem r' v intervalu $-\frac{\alpha}{2} < r' \leq \frac{\alpha}{2}$, respektive $-\frac{\alpha-1}{2} \leq r' \leq \frac{\alpha-1}{2}$ pro liché hodnoty α taková, že $r' \equiv r \mod \alpha$. Pro kladná celá čísla α také definují $r' = r \mod^{+} \alpha$ jako jedinečný element v intervalu $0 \leq r' \leq \alpha$ takový, že $r' \equiv r \mod \alpha$.

Decompose

První z těchto operací je `Decompose(r,`α`)` (5.4), která slouží k dělení vstupu na vyšší a nižší řády. Vstupními parametry jsou element vektoru koeficientů polynomu r a celé číslo α. Návratovou hodnotou jsou rozdělené části vstupního parametru r.

$$
\begin{aligned}
&r := r \mod^{+} q \\
&r_0 := r \mod^{\pm} \alpha \\
&\texttt{If } r - r_0 = q - 1 \\
&\quad \texttt{then } r_1 := 0; r_0 := r_0 - 1 \\
&\texttt{else } r_1 := (r - r_0)/\alpha \\
&\texttt{return } (r_1, r_0)
\end{aligned}
\tag{5.4}
$$

HigBits a LowBits

Tyto operace používají volání výše popsané operace `Decompose` k získání buď vyšších nebo nižších řádů elementů vstupního parametru $\boldsymbol{r}$. Operace `HighBits(r,`$\boldsymbol{\alpha}$`)` vrací hodnotu $\boldsymbol{r_1}$, `LowBits(r,`$\boldsymbol{\alpha}$`)` naopak hodnotu $\boldsymbol{r_0}$. Vstupní parametry těchto operací jsou vektor koeficientů $\boldsymbol{r}$ a celé číslo α.

MakeHint a UseHint

Operace `MakeHint(z,r,`$\boldsymbol{\alpha}$`)` (5.5), kde vstupními parametry jsou vektory polynomů $\boldsymbol{r}$ a $\boldsymbol{z}$ a celé číslo α, má za úkol vytvoření nápovědy, která je posílána spolu s podpisem a je nutná pro jeho ověření.

$$
\begin{aligned}
&\boldsymbol{r_1} := \texttt{HighBits}(\boldsymbol{r}, \alpha) \\
&\boldsymbol{v_1} := \texttt{HighBits}(\boldsymbol{r} + \boldsymbol{z}, \alpha) \\
&\texttt{return } [\![\boldsymbol{r_1} \neq \boldsymbol{v_1}]\!]
\end{aligned}
\tag{5.5}
$$

Notace `return` $[\![\boldsymbol{r_1} \neq \boldsymbol{v_1}]\!]$ znamená vrať hodnotu 1, pokud je výraz ve zdvojených závorkách pravdivý, pokud ne tak vrať hodnotu 0. Je patrné, že vytvořená nápověda má velikost jednoho bitu.

Vstupními parametry operace `UseHint(h,r,`$\boldsymbol{\alpha}$`)` (5.6) jsou opět vektor $\boldsymbol{r}$, celé číslo α a výše vytvořená nápověda h, která slouží k dosažení stejného výsledku sčítání vektorů $\boldsymbol{r}$ a $\boldsymbol{z}$, avšak bez nutnosti použití vektoru $\boldsymbol{z}$.

$$
\begin{aligned}
&m := (q - 1)/\alpha \\
&(r_1, r_0) := \texttt{Decompose}(r, \alpha) \\
&\texttt{if } h = 1 \texttt{ and } r_0 > 0 \texttt{ return } (r_1 + 1) \mod^{+} m \\
&\texttt{if } h = 1 \texttt{ and } r_0 \leq 0 \texttt{ return } (r_1 - 1) \mod^{+} m \\
&\texttt{return } r_1
\end{aligned}
\tag{5.6}
$$

Power2Round

Druhou metodou používanou k dělení elementu vektoru koeficientů polynomu na vyšší a nižší řády je operace `Power2Round(r,d)` (5.7). Jedná se o přímočařejší verzi operace `Decompose`, která neošetřuje situaci, kdy rozdíl výsledků modulárních redukcí je rovný výrazu $q - 1$.

$$\begin{aligned} &r := r \mod^{+} q \\ &r_0 := r \mod^{\pm} 2^d \\ &\texttt{return } ((r - r_0)/2^d, r_0) \end{aligned} \tag{5.7}$$

6 Vlastní implementace algoritmu SHAKE

Jazyk VHDL není klasickým programovacím jazykem, ale jazykem sloužícím k popisu hardware. Instrukce se ve většině případů provádějí současně, i zde ale existují způsoby, jak vyjádřit posloupnost instrukcí. První z těchto možností je použití procesu, který je součástí architektury. Mezi další nástroje použitelné k vyjádření posloupnosti instrukcí patří funkce a procedury. Tyto konstrukce jazyka VHDL jsou velice podobné funkcím nebo metodám v programovacích jazycích. K implementaci je využit sekvenční přístup, hlavně z důvodu zvýšení přehlednosti výsledného algoritmu. V kryptosystému CRYSTALS-DILITHIUM jsou využity dvě varianty algoritmu, SHAKE-128 a SHAKE-256. Obě používají permutaci `KECCAK-p[1600,24]`, pouze s rozdílnou hodnotou parametru přenosové rychlosti r.

6.1 KECCAK

Permutace je postavena na převodu vstupního bitového řetězce do trojrozměrného stavového pole. Na tomto stavovém poli je poté prováděna série pěti permutačních algoritmů.

- θ - theta
- ρ - rho
- π - pi
- χ - chi
- ι - iota

Jeden průchod těmito pěti dílčími algoritmy je nazýván rundou, kterých je v tomto případě 24. Na konci tohoto procesu jsou data převedena zpět z trojrozměrného stavového pole do bitového řetězce.

6.1.1 Převod binárního řetězce do stavového pole

Jedná se o první krok ještě před první rundou permutací. Stavové pole má v tomto případě rozměr $5 \times 5 \times 64$, přičemž do každého elementu pole je umístěn jeden bit binárního řetězce. Jazyk VHDL umožňuje definici vlastních datových typů, pro zvýšení přehlednosti byl definován datový typ `state` (6.1), který je trojrozměrným polem datového typu `std_logic` potřebných rozměrů.

Výpis 6.1: Datový typ state

```
1 type state is array (0 to 4,0 to 4,0 to 63)   of std_logic;
```

O samotný převod se stará procedura `string2state()` (6.2), která má jako vstupní parametr proměnnou typu `std_logic_vector` délky 1600 bitů a jako výstupní parametr proměnnou typu `state`. Převod je realizovaný pomocí tří vnořených `for` cyklů, procedura využívá pomocnou proměnnou `count1600` typu `integer` k přístupu k jednotlivým bitům vstupního bitového řetězce.

Výpis 6.2: Procedura string2state()

```
1  procedure string2state (
2          variable inString : in std_logic_vector(0 to 1599);
3          variable outState : out state) is
4    variable count1600: integer := 0;
5  begin
6
7  for i in 0 to 4 loop
8    for j in 0 to 4 loop
9      for k in 0 to 63 loop
10       outState(j,i,k) := inString(count1600);
11       count1600 := count1600 + 1;
12     end loop;
13   end loop;
14 end loop;
15
16 end string2state;
```

6.1.2 Permutace theta

Smyslem této permutace (6.3) je provést operaci XOR na každém bitu stavového pole s paritami dvou sloupců stavového pole. Pro pomocné proměnné `thetaC` a `thetaD` byl pro přehlednost implementován datový typ `xzArr`, který vyjadřuje řez stavovým polem v rovině xz. V první dvojici vnořených `for` cyklů jsou ustanovovány parity sloupců stavového pole. Ve druhé části procedury jsou ustanovené parity kombinovány vždy pro souřadnici $x + 1$ a $x - 1$ vzhledem k souřadnici příslušné hodnoty x. Nakonec je nad těmito mezihodnotami prováděna operace XOR s každým elementem stavového pole.

Výpis 6.3: Procedura theta()

```
1  procedure theta (variable inState   : in state;
2                   variable outState  : out state) is
3          variable thetaC   :   xzArr;
4          variable thetaD   :   xzArr;
5  begin
6
7    for i in 0 to 4 loop
8      for j in 0 to 63 loop
9        thetaC(i,j) := inState(i,0,j) xor inState(i,1,j)
10                                     xor inState(i,2,j)
11                                     xor inState(i,3,j)
12                                     xor inState(i,4,j);
13     end loop;
14   end loop;
15
16   for i in 0 to 4 loop
17     for j in 0 to 63 loop
18       thetaD(i,j) := thetaC((i-1)mod 5,j) xor
19                      thetaC((i+1)mod 5,(j-1)mod 64);
20     end loop;
21   end loop;
22
23   for i in 0 to 4 loop
24     for j in 0 to 4 loop
25       for k in 0 to 63 loop
26         outState(j,i,k) := inState(j,i,k) xor thetaD(j,k);
27       end loop;
28     end loop;
29   end loop;
30
31 end theta;
```

6.1.3 Permutace rho

V této permutaci (6.4) dochází ke změně souřadnice z každého bitu o předem stanovenou hodnotu. Hodnoty posunu jsou čerpány z dvourozměrného pole konstant `offArr`. Procedura je opět realizována pomocí vnořených `for` cyklů, kterými je hodnota posunu aplikovaná na každý element stavového pole.

Výpis 6.4: Procedura rho()

```
procedure rho (variable inState : in state;
               variable outState: out state) is
begin

  for i in 0 to 4 loop
    for j in 0 to 4 loop
      for k in 0 to 63 loop
        outState(j,i,k) :=
        inState(j,i,((offArr(j,i)mod 64)+k)mod 64);
      end loop;
    end loop;
  end loop;

end rho;
```

6.1.4 Permutace pi

Tato permutace (6.5) mění souřadnice x a y každé z pětadvaceti 64 bitových řad bitů ve stavovém poli. Výsledným stavem je zpřeházení pozic řad bitů stavového pole dle vzorce $pole'(x, y, z) = pole((x + 3y) \mod 5, x, z)$. Procedura opět prochází stavové pole pomocí vnořených `for` cyklů.

Výpis 6.5: Procedura pi()

```
procedure pi (variable inState : in state;
              variable outState: out state) is
begin

  for i in 0 to 4 loop
    for j in 0 to 4 loop
      for k in 0 to 63 loop
        outState(j,i,k) :=
        inState(((j+(3*i))mod 5),j,k);
      end loop;
    end loop;
  end loop;

end pi;
```

6.1.5 Permutace chi

Tato permutace (6.6) provádí operaci XOR každého bitu ve stavovém poli s nelineární funkcí dvou dalších bitů ve stejné řadě vzhledem k souřadnici x, kdy s bitem na souřadnici $x + 1$ je provedena operace XOR s hodnotou '1' a výsledek následně vstupuje do operace AND s bitem na souřadnici $x + 2$.

Výpis 6.6: Procedura chi()

```
1  procedure chi (variable inState : in state;
2                 variable outState: out state) is
3  begin
4
5    for i in 0 to 4 loop
6      for j in 0 to 4 loop
7        for k in 0 to 63 loop
8          outState(j,i,k) := inState(j,i,k) xor
9           ((inState((j+1)mod 5,i,k) xor '1') and
10          inState((j+2)mod 5,i,k));
11       end loop;
12     end loop;
13   end loop;
14
15 end chi;
```

6.1.6 Permutace iota

Pátá permutace (6.7) se od ostatních liší v její závislosti na konkrétní rundě algoritmu KECCAK. Nad bity řady stavového pole na souřadnici $0, 0, z$ a bity konstant, kdy pro každou rundu je jedna definována, je prováděna operace XOR. Tyto konstanty je možné získat implementací algoritmu pro jejich výpočet za běhu algoritmu. Druhou variantou, která byla zvolena v tomto případě je jejich vyjádření pomocí konstantního dvourozměrného pole `rcArr`. Vstupním parametrem této procedury je na rozdíl od ostatních, kdy jsou předávána pouze vstupní a výstupní stavová pole, také proměnná `roundIndex`, ve které je udržována informace o aktuální rundě algoritmu KECCAK. Procedura pomocí této proměnné přistupuje k příslušné hodnotě v poli konstant `rcArr` a následně ji inkrementuje. Parametr z tohoto důvodu musí mít identifikátor `inout`, protože je kromě čtení uvnitř procedury také modifikován.

Výpis 6.7: Procedura iota()

```
1  procedure iota (variable inState  : in state;
2                  variable outState : out state;
3                  variable roundIndex : inout integer) is
4  begin
5
6    for i in 0 to 63 loop
7      outState(0,0,i) := inState(0,0,i) xor rcArr(roundIndex,i);
8    end loop;
9
10   roundIndex := roundIndex + 1;
11
12 end iota;
```

6.1.7 Převod stavového pole do binárního řetězce

O zpětný převod dat ze stavového pole se stará procedura `state2string()`, která má stejný tvar jako procedura `string2state()` s jediným rozdílem, kterým je obrácený směr přiřazování elementů v trojici vnořených `for` cyklů.

6.2 SPONGE

Pro účely použití uvnitř konstrukce houby byla definována procedura `KECCAKp()`, která zapouzdřuje výše popsané procedury do jednoho funkčního celku. Vyznačuje se vstupním parametrem datového typu `std_logic_vector` o rozměru 1600 bitů a výstupním parametrem stejného typu i rozsahu. Tímto je docíleno univerzality použití. Algoritmus CRYSTALS-Dilithium využívá dvě varianty algoritmu SHAKE, které se liší v rozsahu parametru přenosové rychlosti r respektive kapacity c. Takovouto definicí procedury `KECCAKp()` je starost o podobu vstupů přenesena o úroveň výše a obě varianty algoritmu tedy mohou využívat stejnou implementaci. Procedura má pětici pomocných proměnných datového typu `state`, které jsou postupně předávány výše popsaným procedurám jako vstupní a výstupní parametry. Další pomocnou proměnnou typu `integer` procedura využívá k udržování informace o aktuální rundě algoritmu, tato proměnná je také předávána proceduře `iota()`, která na ní závisí. Obě varianty algoritmu SHAKE jsou postaveny na čtyřiadvaceti rundách algoritmu KECCAK, tohoto chování je docíleno umístěním volání výše popsaných procedur do cyklu `for`. O reprezentaci vstupních dat přijatelnou formou se starají procedury `string2state()` a `state2string()`.

Při samotné implementaci houby došlo na problém "předpovídání budoucnosti". Pro dodržení standardu [11], který definuje algoritmus SHAKE jako funkci rozšiřitelného výstupu, by bylo nutné zajistit variabilní délku výstupu, tedy předem neurčený počet fází vymačkávání dat z houby. Obdobný problém nastává u fáze absorpce, spojený s předem neurčeným počtem iterací. Po konzultaci s vedoucím bakalářské práce byla zvolena varianta implementace testovací houby, kde objem vstupních dat je předem daný, čímž je docíleno stavu, kdy objem výstupních dat je také předem známý. Byla určena testovací hodnota velikosti parametru r dané verze algoritmu SHAKE.

6.2.1 Pravidlo pro výplň pad10*1

Vstupní data musí být po vstupu do houby vyplněna do násobku parametru přenosové rychlosti r. Za tímto účelem byla implementována v tomto případě funkce `pad10_1()` (6.8), která má dva vstupní parametry datového typu `integer`. Prvním je hodnota přenosové rychlosti r a druhým délka vyplňovaného bitového řetězce. Funkce podle těchto dvou parametrů vyhodnotí nutnou délku výplně, kterou následně vygeneruje ve formě návratové hodnoty datového typu `std_logic_vector`. Výplň má tvar dvou hodnot '1', mezi kterými je potřebný počet hodnot '0'.

Výpis 6.8: Funkce pad10_1()

```
1  function pad10_1 (rate : integer;
2                    len  : integer) return std_logic_vector is
3        variable j : integer := (-len-2)mod rate;
4        variable pad : std_logic_vector(0 to j+1);
5  begin
6
7    for i in 1 to (j) loop
8      pad(i) := '0';
9    end loop;
10
11   pad(0) := '1';
12   pad(j+1):= '1';
13
14   return pad;
15
16 end function;
```

6.2.2 Testování implementace SHAKE

Pro účely testování byl vytvořen modul `testSHAKE128` (6.9). Struktura entity modulu vychází z potřeby známé velikosti vstupu a výstupu. Na vstupním portu `bitIN` je očekáván bitový řetězec délky 1344 bitů, což odpovídá hodnotě parametru přenosové rychlosti pro variantu algoritmu SHAKE-128. Dle specifikace je před vstupem dat do samotných permutací KECCAK ke vstupnímu řetězci připojena čtveřice bitů hodnoty '1'. Následkem je situace, kdy je nutné využít funkci `pad10_1()` pro výplň vstupu do násobku hodnoty přenosové rychlosti. Nyní je také známá velikost výstupu modulu na portu `bitOUT`, která je dvojnásobkem velikosti vstupu. Testovací modul je realizován pomocí procesu závislém na portu `en`, který zprostředkovává funkcionalitu chip-enable. Uvnitř procesu jsou data ze vstupního portu `bitIN` předávána pomocné proměnné stejné délky. Jedna je také definována pro výstupní data funkce `test_sponge()` (6.10), zapouzdřující testovací konstrukci houby, stavěné dle konstrukce na obrázku 5.1.

Výpis 6.9: Modul testSHAKE128

```
1  entity testSHAKE128 is
2  port (en  : in std_logic;
3      bitIN : in std_logic_vector(0 to 1343);
4      bitOUT: out std_logic_vector(0 to 2687));
5  end testSHAKE128;
6
7  architecture TS128_BEH of testSHAKE128 is
8  begin
9
10 sequence: process(en) is
11    variable dataIN  : std_logic_vector(0 to 1343);
12    variable dataOUT : std_logic_vector(0 to 2687);
13 begin
14
15   dataIN := bitIN;
16   dataOUT := test_sponge(1344, dataIN, 2687);
17   bitOUT <= dataOUT(0 to 2687);
18
19 end process sequence;
20 end TS128_BEH;
```

Funkce je členěna do tří částí, v první z nich dochází k přípravě dat. Dvě pomocné proměnné slouží k rozšíření dat o bitový řetězec skládající se ze čtyř hodnot '1' a následně jsou data doplněna do délky dvojnásobku přenosové rychlosti

pomocí funkce `pad10_1()`. Proměnná `dataOUT` je návratovou hodnotou celé funkce, `dataParts` vlastního datového typu `str128Arr` je určena pro části vstupu délky přenosové rychlosti. Poslední dvě proměnné `initData` a `nextData` slouží pro předávaní dat proceduře `KECCAKp()` uvnitř konstrukce houby. Na začátku vykonávání funkce jsou proměnné `dataParts` předávána vyplněná vstupní data z proměnné `dataPad`.

Výpis 6.10: Funkce test_sponge() - fáze přípravy dat

```
1  function test_sponge(rate  : integer;
2                       dataIN : std_logic_vector;
3                       len    : integer)
4                       return std_logic_vector is
5
6      variable data    : std_logic_vector(0 to dataIN'length+3)
7                       := dataIN & ('1','1','1','1');
8
9      variable dataPad : std_logic_vector(0 to 2687)
10                      := data & pad10_1(rate, data'length);
11
12     variable dataOUT : std_logic_vector(0 to 2687);
13     variable dataParts : str128Arr;
14     variable initData : std_logic_vector(0 to 1599);
15     variable nextData : std_logic_vector(0 to 1599);
16 begin
17   for i in 0 to 1343 loop
18     dataParts(0,i) := dataPad(i);
19   end loop;
20   for j in 0 to 1343 loop
21     dataParts(1,j) := dataPad(j + 1344);
22   end loop;
23
24 ...
```

V druhé části funkce (6.11) dochází k absorpci vstupních dat s využitím procedury `KECCAKp()`. Proměnné `initR128` a `initC128` jsou inicializačními vektory houby plné nulových hodnot. Nad první částí vstupu z proměnné `dataParts` je prováděna operace XOR spolu s inicializačním vektorem, výsledek je předáván proměnné `initData` v cyklu `for`. Druhý cyklus provádí doplnění proměnné `initData` nulovými hodnotami z druhého inicializačního vektoru. Proceduře `KECCAKp()` je předána proměnná s předzpracovanými daty spolu s proměnnou `nextData`, která bude výstupem procedury. Následuje druhá absorpce, která se od té první liší operandy operace XOR. Místo inicializačního vektoru `initR128` vstupuje do operace výstup

první absorpce v rozsahu 0 až 1343. Proměnná `initData`, která je vstupem pro proceduru `KECCAKp()` i pro druhou absorpci je posléze doplněna taktéž výstupem té první.

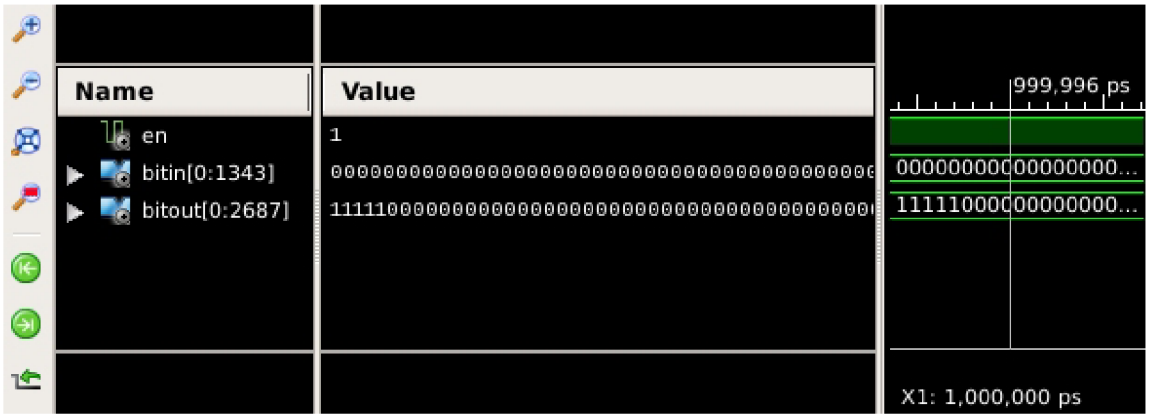
Výpis 6.11: Funkce test_sponge() - fáze absorbce a vymačkávání

```
...

  for i in 0 to 1343 loop
    initData(i) := dataParts(0,i) xor initR128(i);
  end loop;
  for i in 1344 to 1599
      initData(1344 to 1599) := initC128(i - 1344);
  end loop;

  KECCAKp(initData, nextData);

  for i in 0 to 1343 loop
    initData(i) := dataParts(1,i) xor nextData(i);
  end loop;
  for i in 1344 to 1599 loop
    initData(i) := nextData(i);
  end loop;

  KECCAKp(initData, nextData);

  for i in 0 to 1343 loop
    dataOUT(i) := nextData(i);
  end loop;

  KECCAKp(nextData, initData);

  for i in 0 to 1343 loop
    dataOUT(i + 1344) := initData(i);
  end loop;

  return dataOUT;
end test_sponge;
```

Výstup procedury `KECCAKp()` je po druhé absorpci předáván výstupní proměnné `dataOUT` jako první část výstupu funkce. Třetí fází je vymačkávání, pro které je opět využito procedury `KECCAKp()`, jen s prohozením vstupních parametrů. V zá-

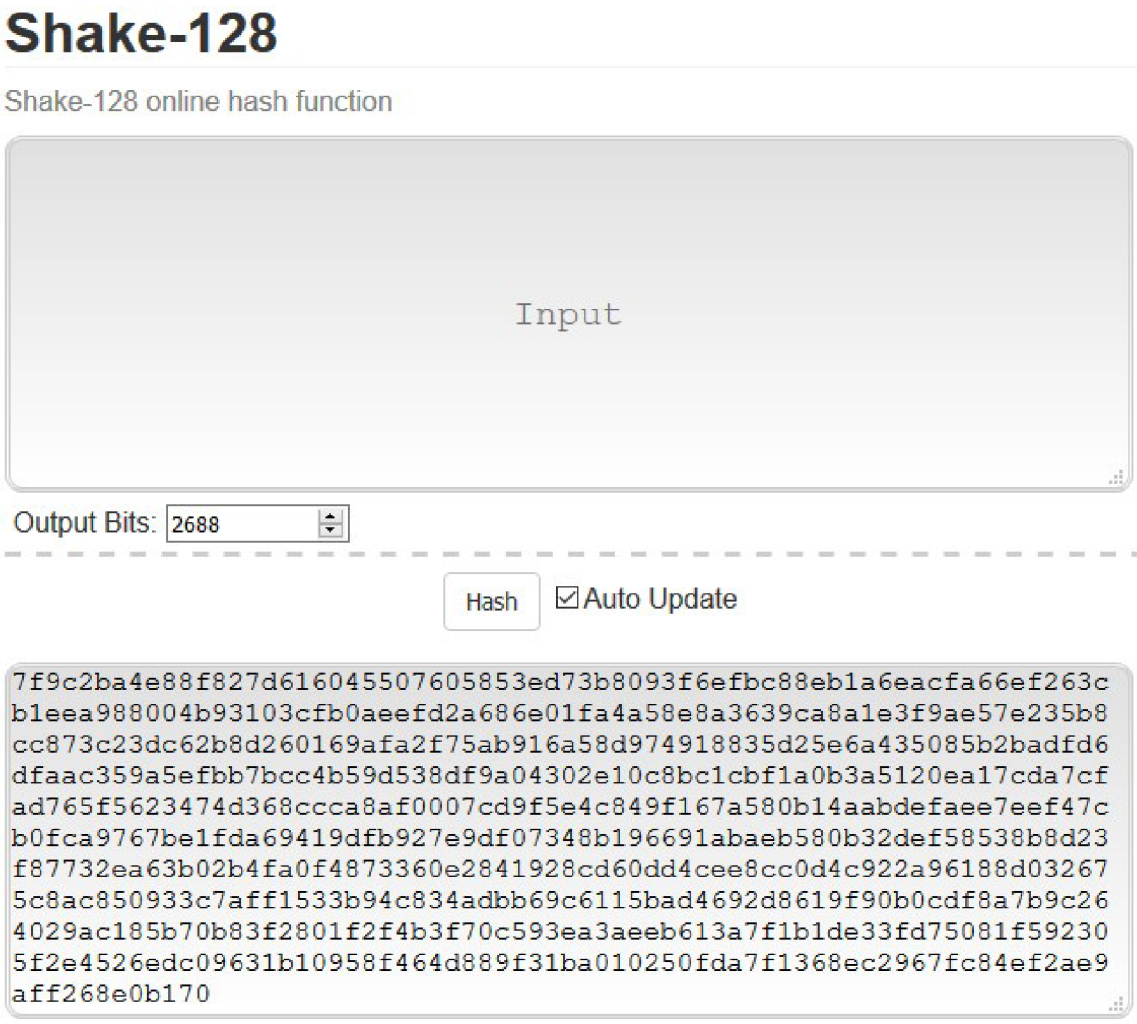
věru funkce je výstup procedury velikosti parametru přenosové rychlosti předáván do druhé poloviny proměnné `dataOUT`, která je návratovou hodnotou funkce.

K testování bylo využito aplikace ISim, která je součástí vývojového prostředí ISE ve spojení s funkcionalitou TestBench, což je standardní přístup k testování modulů pomocí simulace. K ověření výsledků bylo použito veřejně dostupné implementace SHAKE-128 ve formě webové aplikace, která umožňuje nastavit požadovanou délku výstupu. Nejjistějším výsledkem by bylo porovnávat implementovaný modul s jinou, funkční implementací SHAKE-128 v jazyce VHDL, ale v danou chvíli nebyla tato možnost dostupná. Vstupem pro testování modulu je binární řetězec délky 1344 plný nulových hodnot, do webové aplikace byl poslán prázdný řetězec.

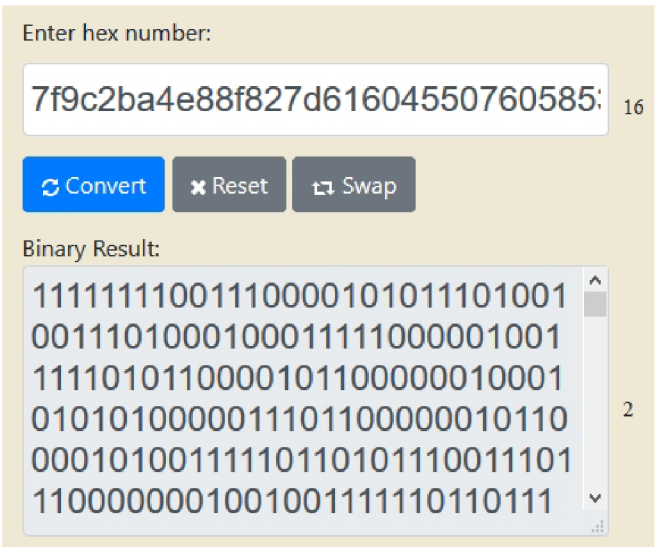


Obr. 6.1: Výřez snímku obrazovky aplikace ISim - testování implementace SHAKE

Na obrázku 6.1 je výsledek simulace modulu `testSHAKE128`, obrázek 6.2a obsahuje výsledek webové implementace SHAKE-128 s požadovanou délkou výstupu. Výsledek konverze hexadecimálního výstupu do binární soustavy je na obrázku 6.2b. Už na první pohled je patrné, že výsledky nesouhlasí. Možný problém v souvislosti znakové sady používané webovou aplikací je vyřešen použitím prázdného řetězce coby vstupu. Jisté je, že vstupem do konstrukce houby v obou implementacích je řetězec délky parametru přenosové rychlosti, dle specifikace funkce SHAKE. Možný rozdíl ve vstupu je pozice přidávaného binárního řetězce ve formě čtyř hodnot '1'. Toto by bylo možné ošetřit pokud by vstupem do VHDL implementace byl také prázdný řetězec, což ale není možné. Naopak pokud by vstupem do webové implementace byl řetězec plný nul, nastává problém spojený se znakovou sadou. Metoda zvolená pro testování implementace funkce SHAKE rozhodně není dokonalá. Tvar dvou porovnávaných výstupů je výrazně odlišný, což nasvědčuje tomu, že by problém mohl být v samotných permutacích uvnitř procedury `KECCAKp()`. To by bylo možné ověřit důkladnou analýzou cesty precizně strukturovaných vstupů vnitřními permutacemi.



(a) Webová implementace SHAKE-128 [14]



(b) Převod do binární soustavy [15]

Obr. 6.2: Výřezy snímků webových aplikací - testování implementace SHAKE

7 Vlastní implementace dílčích operací

V rámci implementace dílčích operací kryptosystému CRYSTALS-Dilithium byl uplatněn, na rozdíl od implementace algoritmu SHAKE, strukturální přístup k definici funkčních bloků. Jedná se tedy o samostatné moduly jazyka VHDL, které jsou později zapouzdřovány do větších celků, pomocí instanciace komponent. Jak bylo zmíněno v kapitole 5, parametry q, který je prvočíslem definujícím maximální hodnotu koeficientů polynomů a n, určujícím počet těchto koeficientů, jsou předem definované a vždy stejné.

7.1 DECOMPOSE

Operace slouží k dělení vstupních elementů. V této implementaci je vstupem jediný element r vektoru koeficientů polynomu spolu s celočíselnou hodnotou α, kterou operace využívá k modulární redukci. Ze specifikace Dilithia vyplývají pro hodnotu α určitá omezení. Musí vyhovovat podmínkám $\alpha \mid q-1$, $q > 2\alpha$ a $\alpha \mod 2 = 0$. Operace také využívá centralizované modulární redukce $\mod^+$ a $\mod^\pm$. Realizace těchto nestandardních modulárních redukcí je provedena pomocí dvou funkcí `mod_pm()` (7.1) a `mod_p()`.

Výpis 7.1: Funkce mod_pm()

```
1  function mod_pm(operand : integer;
2                  modulus : integer) return integer is
3        variable result : integer;
4  begin
5
6    if (isEven(modulus)) then
7      result := (operand mod modulus);
8
9      if (-(modulus/2) < result and result <= (modulus/2)) then
10       return result;
11     else
12       return result - modulus;
13     end if;
14   else
15     result := 0;
16   end if;
17
18   return result;
19 end function;
```

Ze specifikace Dilithia vyplývá, že výsledné chování funkce `mod_p()` je stejné jako v případě standardní matematické operace modulo. Funkce `mod_pm()` specifikuje centralizovaný prostor výsledků popsaný v kapitole 5.3.3, pokud výsledek do tohoto rozsahu nepatří, je od něj odečtena hodnota proměnné `modulus`, vzhledem ke které je prováděna operace modulo. Funkce je definována pouze pro sudé hodnoty proměnné `modulus`, jelikož touto proměnnou je vždy hodnota α, která je z definice sudá.

Výpis 7.2: Modul DECOMPOSE

```
1  entity DECOMPOSE is
2  port(   clk     : in std_logic;
3          coef    : in integer;
4          alpha   : in integer;
5          pr1     : out integer;
6          pr0     : out integer);
7  end DECOMPOSE;
8
9  architecture DEC_BEH of DECOMPOSE is
10 begin
11 sequence: process(clk) is
12   variable r : integer;
13   variable r0: integer;
14   variable r1: integer;
15 begin
16   if rising_edge(clk) then
17     r := mod_p(coef, q);
18     r0:= mod_pm(r, alpha);
19     if (r-r0 = q-1) then
20       r1 := 0;
21       r0 := r0 - 1;
22     else
23       r1 := (r - r0)/alpha;
24     end if;
25     pr1 <= r1;
26     pr0 <= r0;
27   end if;
28 end process sequence;
29 end DEC_BEH;
```

Modul `DECOMPOSE` (7.2) je realizovaný pomocí procesu, tedy sekvence příkazů. Jedná se o nejnižší modul v hierarchii, který je používán ostatními moduly. Vstupy modulu jsou mimo port `clk`, kterým je poskytován hodinový signál, port `coef` na kterém modul očekává koeficient, který je potřeba rozdělit a `alpha`, který slouží pro vstup hodnoty α. Výstupy modulu jsou porty `pr1` a `pr0`, kterými modul vrací rozdělené části vstupního koeficientu. Porty pracují s datovým typem `integer`, tedy celým číslem. Hodnota parametru q tady maximální hodnota vstupních koeficientů je nižší než rozsah tohoto datového typu, druhou předností je přehlednost. Uvnitř procesu `sequence` jsou deklarovány tři pomocné proměnné, které slouží k ukládání mezivýsledků operace.

Prvním krokem je modulární redukce vstupu `coef` a uchování výsledku v proměnné `r`. Tento krok je nutný, protože existují situace, kdy vstupem je součet koeficientů, druhým důvodem je zajištění předvídatelného chování uvnitř modulu. Poté je výsledek předchozího kroku poslán do funkce `mod_pm()`, kde dochází k centralizované modulární redukci $r \mod {}^{\pm}\alpha$, k uchování výsledku slouží proměnná `r`. Dalším krokem je posouzení podmínky `if r-r0 = q-1`. Pokud je tato vyhodnocena jako pravdivá, proměnná `r1` nabývá předem dané hodnoty 0 a hodnota proměnné `r0` je dekrementována. Při nepravdivosti podmínky zůstává proměnná `r0` nezměněná, naopak proměnná `r1` nabývá hodnoty `(r - r0)/alpha`. Posledním krokem je předání výsledných hodnot portům `pr1` a `pr0`.

7.1.1 Testování modulu DECOMPOSE

K testování modulu byla využita metoda TestBench. Jedná se o standardní přístup k testování VHDL modulů, kdy je chování simulováno. V tomto případě byla využita aplikace ISim, která je součástí volně dostupného vývojového prostředí ISE společnosti Xilinx. TestBench je samostatným VHDL modulem bez entity, což znamená, že sám o sobě nemá žádné vstupní nebo výstupní porty. Testovaný modul je uvnitř architektury modulu TestBench instanciován v podobě komponentu a pomocí signálů je možné na vstupy tohoto komponentu poslat testovací data. Aplikace ISim zprostředkovává grafickou reprezentaci jak vstupních, tak výstupních hodnot testovaného modulu. V rámci testování byly zvoleny různé testovací hodnoty prvočíselného parametru q, za účelem ověření matematického chování modulu `DECOMPOSE` v souladu se specifikací Dilithia.

Hodnoty parametru α byly zvoleny tak, aby vyhovovaly podmínkám $\alpha \mid q - 1$, $q > 2\alpha$ a $\alpha \mod 2 = 0$, nutným pro správné fungování modulu. Tabulka 7.1 znázorňuje výsledky testování. Sloupec $\texttt{coef}_{\texttt{if}}$ vyjadřuje vstupní koeficient zvolený tak, aby vyhovoval podmínce v modulu, sloupec $\texttt{coef}_{\texttt{else}}$ naopak takový, aby jí nevyhověl. Výsledné hodnoty jsou vždy napravo od příslušného koeficientu. U vyšších

hodnot parametru q byl $\texttt{coef}_{\texttt{if}}$ zvolen jako $q - 1$. Takto zvolený koeficient nemá šanci podmínce v modulu nevyhovět, je to dáno podmínkou soudělnosti kladenou na parametr α.

Tab. 7.1: Tabulka testovacích hodnot modulu DECOMPOSE

q	α	$\texttt{coef}_{\texttt{if}}$	r1	r0	$\texttt{coef}_{\texttt{else}}$	r1	r0
19	6	17	0	-2	23	1	-2
4001	2000	4000	0	-1	8027	0	25
112909	37636	112908	0	-1	56454	1	18818
3010349	97108	3010348	0	-1	714265	7	34509
8380417	349184	8380416	0	-1	417256	1	68072

Ukázka ověření správnosti výsledků

Pro účely ukázky jsou zvoleny hodnoty $q = 19$, $\alpha = 6$ a vstupní koeficienty $c_1 = 17$, $c_2 = 23$.

V případě c_1 má úvodní operace tvar $r = 17 \mod^{+} 19$, vychází $r = 17$. Následuje operace $r_0 = 17 \mod^{\pm} 6$, jejíž centralizovaný prostor výsledků je $-\frac{6}{2} < r_0 \leq \frac{6}{2}$. Výsledek $r_0 = 17 \mod 6$ je 5, nepatří tedy do centralizovaného prostoru a je nutno odečíst hodnotu 6, kdy konečný výsledek je $r_0 = -1$. Podmínka $r - r_0 = q - 1$ je splněna, konečnými výsledky jsou $r_1 = 0$ a $r_0 = -2$.

Pro hodnotu c_2 je výsledkem první operace $r = 23 \mod^{+} 19$ hodnota $r = 4$. Mezivýsledek $r_0 = 4 \mod 6$ je $r_0 = 4$, stejně jako v případě c_1 nepatří do centralizovaného prostoru a je tedy nutno odečíst hodnotu 6, kdy konečný výsledek operace $r_0 = 4 \mod^{\pm} 6$ nabývá hodnoty $r_0 = -2$. Podmínka $r - r_0 = q - 1$ nyní není splněna, r_1 tedy nabývá hodnoty $\frac{4-(-2)}{6} = 1$, hodnota r_0 zůstává nezměněna. Výsledné hodnoty r_1 i r_0 v obou případech souhlasí s výstupy modulu `DECOMPOSE` v tabulce 7.1.

7.2 HIGHBITS

Tento modul umožňuje provést operaci zprostředkovanou modulem `DECOMPOSE` na vstupním vektoru koeficientů délky n, přičemž výstupem modulu `HIGHBITS` (7.3) je $n-$tice hodnot r_1. Operace je opět realizována pomocí VHDL modulu, který má vstupní porty `coefs` a `alpha`. Port `alpha` má identické charakteristiky se stejnojmenným portem modulu `DECOMPOSE`, jelikož hodnoty tímto přijímané jsou přímo

přenášeny na vstupní porty vnořených modulů `DECOMPOSE`. K reprezentaci dat portu `coefs` slouží vlastní datový typ `coef_vector`, který je jednorozměrným polem datového typu `integer` o délce $n - 1$. Výstupní port také pracuje s tímto datovým typem, obsahem portu `highCoefs` jsou hodnoty r_1 získané z modulů `DECOMPOSE`.

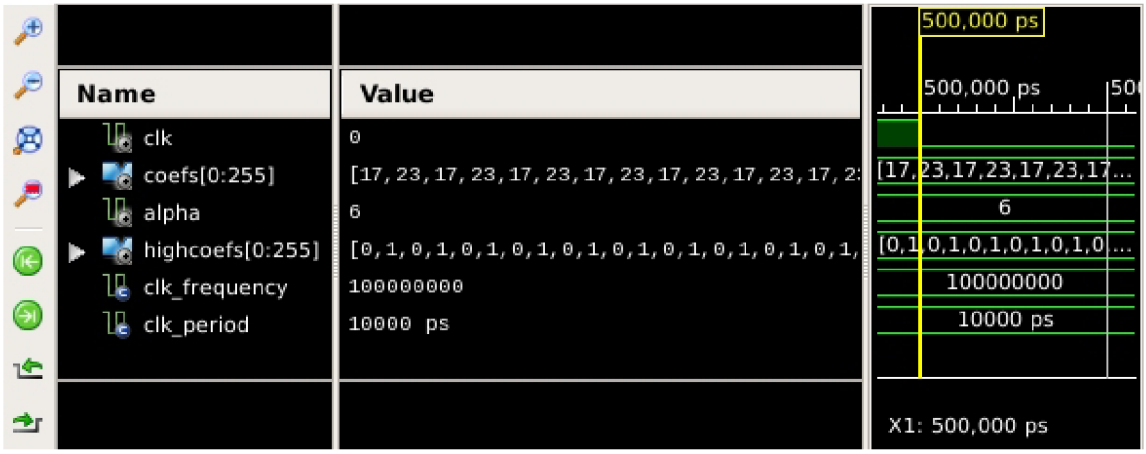
Výpis 7.3: Modul HIGHBITS

```
1  entity HIGHBITS is
2  port (  clk          : in std_logic;
3          coefs        : in coef_vector;
4          alpha        : in integer;
5          highCoefs    : out coef_vector);
6  end HIGHBITS;
7
8  architecture HB_BEH of HIGHBITS is
9  component DECOMPOSE
10         port (clk    : in std_logic;
11               coef   : in integer;
12               alpha  : in integer;
13               pr1    : out integer;
14               pr0    : out integer);
15 end component;
16 begin
17   GEN_DECs : for i in 0 to n - 1 generate
18     D: DECOMPOSE port map
19     ( clk => clk,
20       coef => coefs(i),
21       alpha => alpha,
22       pr1 => highCoefs(i));
23   end generate GEN_DECs;
24 end HB_BEH;
```

V deklarativní části architektury modulu je uvedena definice komponentu se kterým modul pracuje. Jedná se o popsání entity vnořovaného modulu, `HIGHBITS` potřebuje vědět jaké porty jsou k dispozici a jaké datové typy jsou očekávány. V samotném těle architektury dochází k instanciaci potřebného počtu vnořených modulů. Pomocí konstrukce `port map` jazyka VHDL je určeno, které porty modulu budou propojeny s porty vnořeného modulu. V případě `HIGHBITS` zůstává výstupní port `pr0` modulu `DECOMPOSE` nezapojený, jelikož není potřebný. Instanciace správného počtu komponent je realizována cyklem `for`, kdy počet iterací je určen parametrem `n`. Tento parametr je konstantou definovanou v knihovně `UTILS` spolu s prvočíselným parametrem `q` a funkcemi využívanými moduly popsanými v této kapitole.

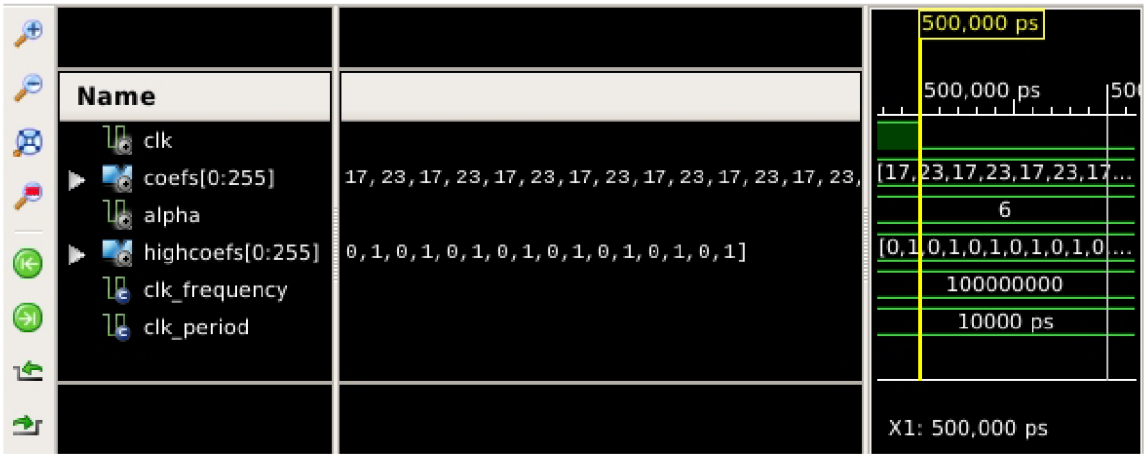
7.2.1 Testování modulu HIGHBITS

K simulaci byla opět využita aplikace ISim spolu s funkcionalitou TestBench vývojového prostředí Xilinx ISE. Pro ověření správné funkce modulu `HIGHBITS` byly použity hodnoty z ukázky ověření správnosti výsledků modulu `DECOMPOSE`, jelikož funkce modulu je pouze zprostředkování této již otestované funkcionality. Hodnota parametru `n` byla vybrána dle specifikace Dilithia, tedy 256.



Obr. 7.1: Výřez snímku obrazovky aplikace ISim - testování HIGHBITS

Testovací vstupní signál datového typu `coef_vector`, plný opakujících se číselných hodnot 17 a 23, by za předpokladu správného fungování modulu měl vyústit ve výstupní signál `highCoefs`, plný opakujících se hodnot 0 a 1 dle tabulky 7.1. Z ukázky na obrázku 7.1 je patrné, že tomu tak skutečně je. Správnost použití parametru `n` je ilustrována na obrázku 7.2, kde je výstupní signál `highCoefs` o rozměru 0 až 255 je očekávanými výstupními hodnotami zaplněn v celém svém rozsahu.



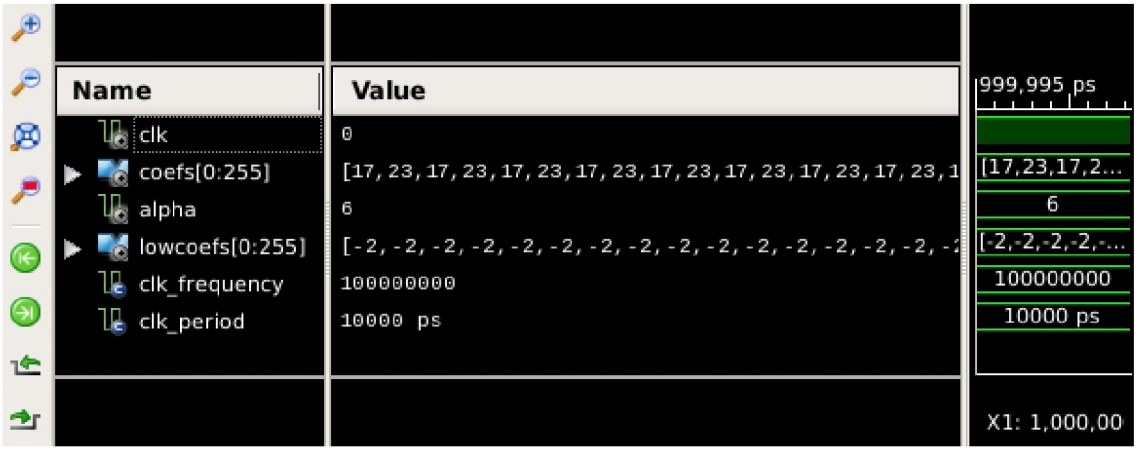
Obr. 7.2: Výřez snímku obrazovky aplikace ISim - 2. testování HIGHBITS

7.3 LOWBITS

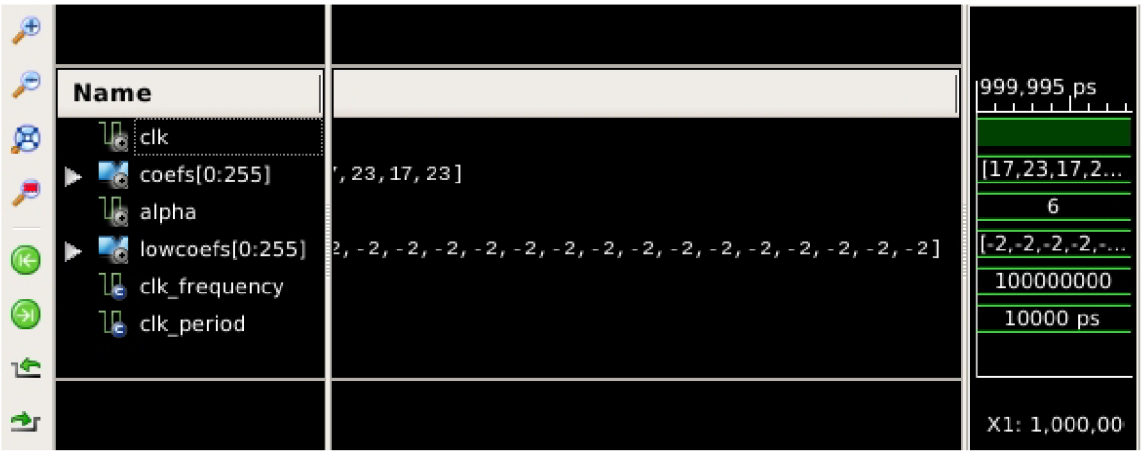
Tato operace je opět realizována samostatným modulem, který pro svou funkci využívá stejných principů jako v případě modulu `HIGHBITS`. Jediným rozdílem mezi těmito dvěma moduly je, že v konstrukci `port map`, pomocí které dochází k propojení portů mezi modulem `LOWBITS` a vnořenými `DECOMPOSE` je místo portu `pr1` mapován port `pr0`.

7.3.1 Testování modulu LOWBITS

Metoda testování zůstává i v tomto případě stejná jako u ostatních simulovaných modulů. Při použití dosavadních testovacích parametrů nastává situace, kdy očekávané výsledné hodnoty na portech `pr0` vnořených modulů `DECOMPOSE` jsou pro obě hodnoty vstupních koeficientů stejné.



Obr. 7.3: Výřez snímku obrazovky aplikace ISim - testování LOWBITS



Obr. 7.4: Výřez snímku obrazovky aplikace ISim - 2. testování LOWBITS

Jelikož funkčnost vnořeného modulu byla již otestována, není tento fakt z hlediska testování `LOWBITS` známkou nefunkčnosti. Při zachování testovacích parametrů jako při simulaci modulu `HIGHBITS` je očekávaná hodnota ve výstupním signálu `lowCoefs` pro oba vstupní koeficienty -2, dle tabulky 7.1. Z obrázků 7.3 a 7.4 je patrné, že očekávané hodnoty získáme a jejich počet také souhlasí.

7.4 MAKEHINT

Výstupem této operace je jediný bit, který nabývá jedné ze dvou možných hodnot na základě porovnání koeficientů dvou vektorů, přičemž prvním z nich je vstupní vektor r a druhým součet obou vstupních vektorů $r+z$. Realizace je provedena opět pomocí samostatného modulu, který využívá dvojice výše popsaného modulu `HIGHBITS`. Pro sčítání vstupních vektorů byla implementována funkce `add_coef_vectors()` (7.4), a k porovnávání vektorů funkce `eval_coef_vectors()` (7.5).

Výpis 7.4: Funkce add_coef_vectors()

```
1  function add_coef_vectors(  vector1 : coef_vector;
2                              vector2 : coef_vector)
3                              return coef_vector is
4        variable result : coef_vector;
5  begin
6    for i in 0 to n - 1 loop
7      result(i) := vector1(i) + vector2(i) mod q;
8    end loop;
9    return result;
10 end function;
```

Sčítání vektorů koeficientů je realizováno `for` cyklem, který má opět vazbu na parametr `n`. Uvnitř cyklu jsou po indexech procházeny vstupní proměnné datového typu `coef_vector` a jejich součet je po modulární redukci parametrem `q` přiřazován do pomocné proměnné, která je zároveň návratovou hodnotou funkce. Porovnávání vektorů je také realizováno cyklem `for`, kdy informaci o počtu rovností koeficientů na stejných indexech uchovává pomocná proměnná `equalCount` datového typu `integer`, která je na začátku funkce `eval_coef_vectors()` vždy inicializována na hodnotu 0. Pokud se na konci vykonávání funkce všechny koeficienty rovnají, má tato proměnná hodnotu `n - 1` a návratová hodnota funkce nabývá stavu '0'.

Výpis 7.5: Funkce eval_coef_vectors()

```
1  function eval_coef_vectors( vector1 : coef_vector;
2                              vector2 : coef_vector)
3                              return std_logic is
4       variable equalCount : integer := 0;
5  begin
6    for i in 0 to n - 1 loop
7      if (vector1(i) = vector2(i)) then
8        equalCount := equalCount + 1;
9      end if;
10   end loop;
11
12   if(equalCount = n - 1) then
13     return '0';
14   else
15     return '1';
16   end if;
17
18 end function;
```

Entita modulu `MAKEHINT` (7.6) má dva vstupní porty datového typu `coef_vector`, `rVec` a `zVek`, které jsou určeny pro vektory koeficientů. Dalším vstupem je port očekávající datový typ `integer` určený pro parametr α, který je předáván dvojici vnořených komponent `MAKEHINT`. Kromě všudypřítomného portu `clk` má entita ještě výstupní port `hint`, který slouží pro výstup operace. V deklarativní části architektury je specifikován komponent `HIGHBITS` a tři pomocné signály. První je součtem vstupních vektorů koeficientů pomocí funkce `add_coef_vectors()`, další dva slouží pro návratové hodnoty vnořených komponent `HIGHBITS`. V samotném těle architektury se nachází instanciace dvojice vnořených modulů, první je určen pro vstup `rVec` a na výstup modulu je přiveden pomocný signál `rHcoefs`. Druhý vnořený modul je určen pro součet dvojice vstupních vektorů zprostředkovaný pomocným signálem `rzVec` a pro výstup tohoto modulu je určen pomocný signál `rzHcoefs`. Posledním krokem nutným pro správnou funkci modulu je porovnání výstupů vnořených komponent `HIGHBITS`, k tomu je použita funkce `eval_coef_vectors()`, které jsou jako parametry předány pomocné signály `rHcoefs` a `rzHcoefs`. Návratová hodnota funkce je přímo přiřazena na výstupní port `hint`.

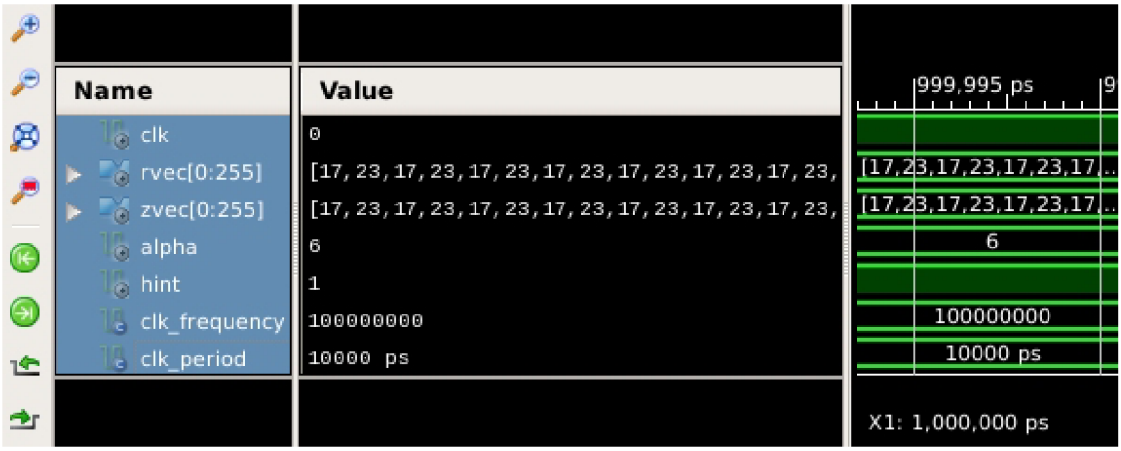
Výpis 7.6: Modul MAKEHINT

```
1  entity MAKEHINT is
2    port( clk    : in std_logic;
3          rVec   : in coef_vector;
4          zVec   : in coef_vector;
5          alpha  : in integer;
6          hint   : out std_logic);
7  end MAKEHINT;
8
9  architecture MH_BEH of MAKEHINT is
10 component HIGHBITS
11     port (clk    : in std_logic;
12           coefs  : in coef_vector;
13           alpha  : in integer;
14       highCoefs  : out coef_vector);
15 end component;
16 signal rzVec    : coef_vector := add_coef_vectors(rVec, zVec);
17 signal rHcoefs  : coef_vector;
18 signal rzHcoefs : coef_vector;
19
20 begin
21   rHB: HIGHBITS port map
22   (      clk      => clk,
23          coefs    => rVec,
24          alpha    => alpha,
25          highCoefs => rHcoefs);
26   rzHB: HIGHBITS port map
27   (      clk      => clk,
28          coefs    => rzVec,
29          alpha    => alpha,
30          highCoefs => rzHcoefs);
31   hint <= eval_coef_vectors(rHcoefs, rzHcoefs);
32 end MH_BEH;
```

7.4.1 Testování modulu MAKEHINT

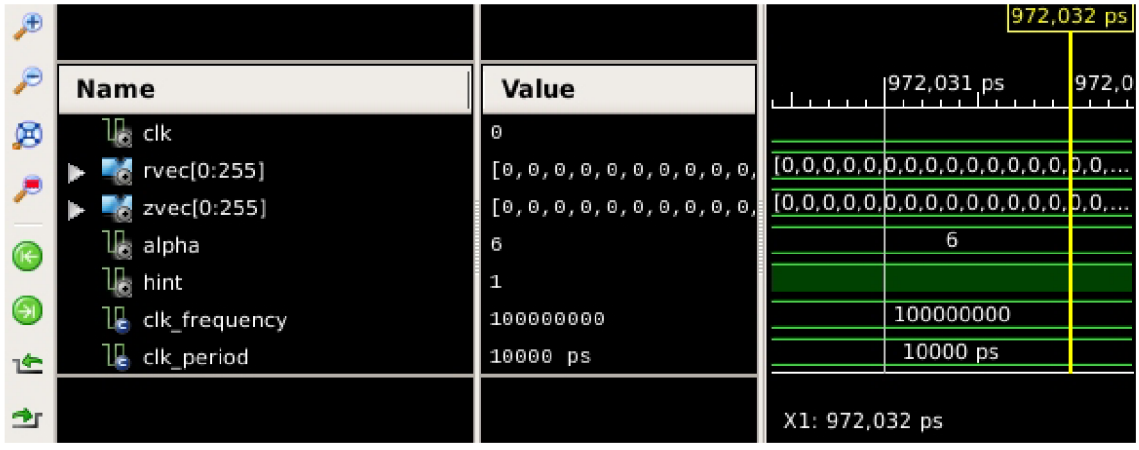
Metodou testování je i v tomto případě simulace modulu pomocí aplikace ISim a funkce TestBench vývojového prostředí ISE. Testovací parametry zůstávají i v tomto případě stejné. Pokud je na vstupní port `rVec` přiveden vektor koeficientů skládající se z opakovaných dvojic čísel 17 a 23 a na port `zVec` modulu přiveden stejný

vektor, výsledkem funkce `add_coef_vectors()` bude vektor opakujících se dvojic čísel 15 a 8. S těmito vstupními koeficienty vrací modul `HIGHBITS` jejich hodnoty r_1 2 a 1. Hodnoty r_1 pro čísla 17 a 23 dle tabulky 7.1 jsou 0 a 1. Do funkce `eval_coef_vectors()` tedy vstupují opakující se dvojice čísel 0 a 1 obsažené v signálu `rHcoefs` a opakující se dvojice čísel 2 a 1 v signálu `rzHcoefs`. Tyto vektory se rozhodně nerovnají, na výstupním portu `hint` by měla být hodnota '1', což simulace potvrzuje na obrázku 7.5.



Obr. 7.5: Výřez snímku obrazovky aplikace ISim - testování MAKEHINT

Pro ověření druhého možného stavu je nejjednodušším způsobem přivést na oba vstupní porty modulu vektory plné nulových hodnot. V tomto případě jejich součet bude také plný nul a výstupy modulů `HIGHBITS` budou také nulové vektory.



Obr. 7.6: Výřez snímku obrazovky aplikace ISim - 2. testování MAKEHINT

Z výsledku simulace na obrázku 7.6 je patrné, že v tomto případě zůstává hodnota na výstupním portu `hint` stejná jako při předchozí simulaci. Vzhledem k přímočarosti provedení funkcí používanými modulem `MAKEHINT` usuzuji, že jejich funkčnost je

správná. Problémem je s největší pravděpodobností vlastnost jazyka VHDL, kdy instrukce jsou prováděny současně, tedy problém s časováním.

Domnívám se, že by řešením mohlo být nasazení zpožďovacích prvků v závislosti na hodinovém signálu `clk`, které by zajistily vykonání operací zprostředkovaných funkcemi `add_coef_vectors()` a `eval_coef_vectors()` tak, aby byl na port `hint` přiřazen očekávaný výsledek.

7.5 USEHINT

Modul `USEHINT` (7.7) využívá jednobitového výstupu modulu `MAKEHINT`. V rámci Dilithia je pomocí těchto dvou modulů dosaženo snížení velikosti veřejného klíče. Modul má kromě vstupního portu pro hodinový signál dva porty pracující s datovým typem `integer` a jeden očekávající datový typ `std_logic`. Celočíselné vstupy jsou pro parametr α a koeficient, které jsou předávány vnořenému modulu `DECOMPOSE`. Tento je v deklarativní části architektury modulu definován jako komponenta, spolu s dvěma pomocnými signály, které slouží pro jeho výstupy.

Výpis 7.7: Architektura modulu USEHINT

```
1  ...
2
3  architecture UH_BEH of USEHINT is
4  component DECOMPOSE
5      port (clk   : in std_logic;
6            coef  : in integer;
7            alpha : in integer;
8            pr1 : out integer;
9            pr0 : out integer);
10 end component;
11     signal sr0  : integer;
12     signal sr1  : integer;
13 begin
14     D: DECOMPOSE port map
15     ( clk => clk,
16       coef => coef,
17       alpha => alpha,
18       pr0 => sr0,
19       pr1 => sr1);
20
21 ...
```

Po instanciaci vnořeného modulu je součástí těla architektury proces (7.8), uvnitř kterého jsou výstupy modulu `DECOMPOSE` předány pomocným proměnným `r0` a `r1`. Třetí proměnnou je celočíselná hodnota `m`, která slouží uvnitř procesu k modulárním redukcím konečných výsledků. Výstup modulu je závislý na dvojici složených podmínek `if`, které pracují s jednobitovou hodnotou nápovědy `hint` generovanou modulem `MAKEHINT` a hodnotou signálu `r0`, která je výstupem vnořeného modulu `DECOMPOSE`. Na základě vyhodnocení těchto podmínek je na výstup vedena buď modifikovaná a posléze modulárně redukovaná nebo nezměněná hodnota signálu `r1`.

Výpis 7.8: Proces v modulu USEHINT

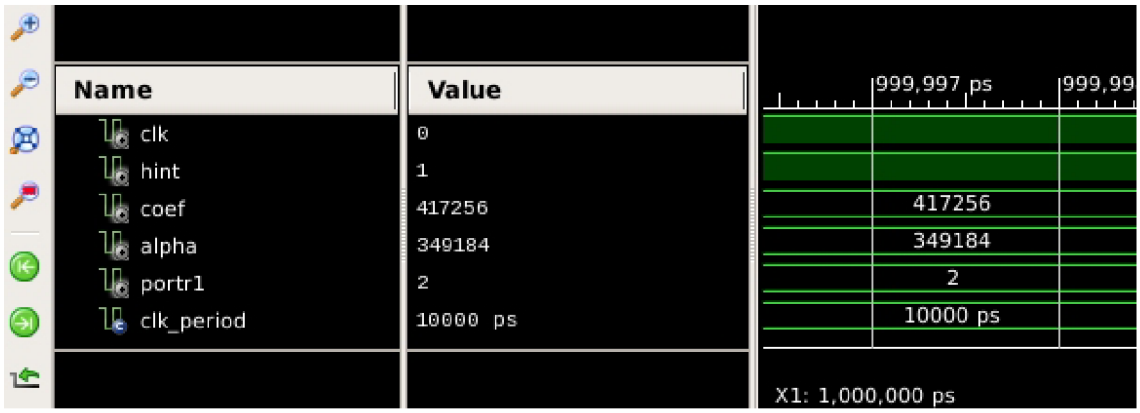
```
...

sequence: process(clk) is
    variable m : integer;
    variable r0: integer;
    variable r1: integer;
begin
  if rising_edge(clk) then
    m := (q - 1)/alpha;
    r0:= sr0;
    r1:= sr1;

    if (hint = '1' and r0 > 0) then
        portR1 <= mod_p((r1 + 1),m);

    elsif (hint = '1' and r0 <= 0) then
        portR1 <= mod_p((r1 - 1),m);

    else portR1 <= r1;
    end if;
  end if;
end process sequence;
end UH_BEH;
```

7.5.1 Testování modulu USEHINT

Testovací metoda zůstává i u tohoto modulu stejná jako doposud. Hodnoty parametrů q a α jsou pro tuto simulaci zvoleny jako $q = 8380417$ a $\alpha = 349184$. Tato hodnota parametru q vychází ze specifikace Dilithia, navíc už testované hodnoty vstupních koeficientů modulu `DECOMPOSE`, spjaté s těmito hodnotami, umožňují si-

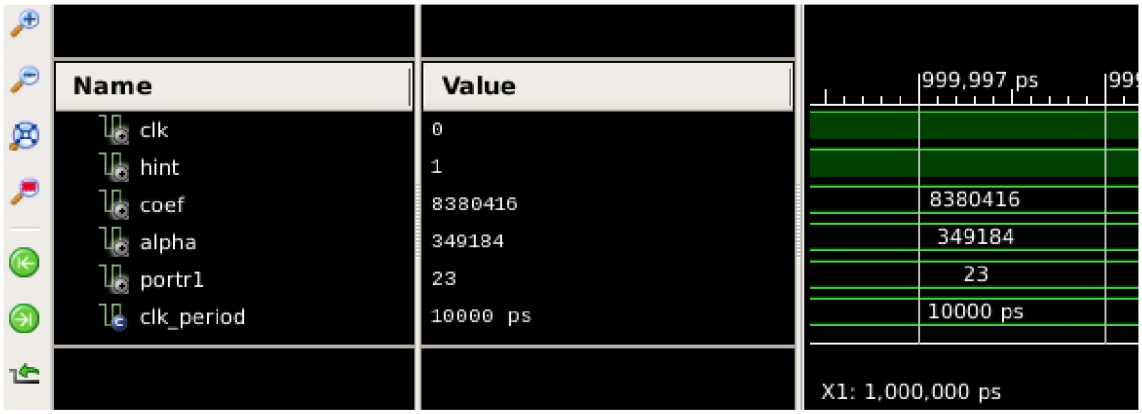
mulaci všech stavů určených složenými podmínkami v modulu `USEHINT`.

Pro první simulaci je tedy hodnota vstupního koeficientu `coef` rovna 417256 a na port `hint` je přivedena hodnota '1', pro splnění první podmínky v modulu. Hodnota proměnné `m` je ve všech případech rovna 24. Výstupy modulu `DECOMPOSE` v proměnných `r0` a `r1` jsou dle tabulky 7.1 v tomto případě 68072 a 1. První složená podmínka je splněna a na výstupním portu očekáváme hodnotu výrazu $(r_1 + 1) \bmod^{+} 24$, což je 2. Z obrázku 7.7 je patrné, že výstup modulu souhlasí s očekáváním.



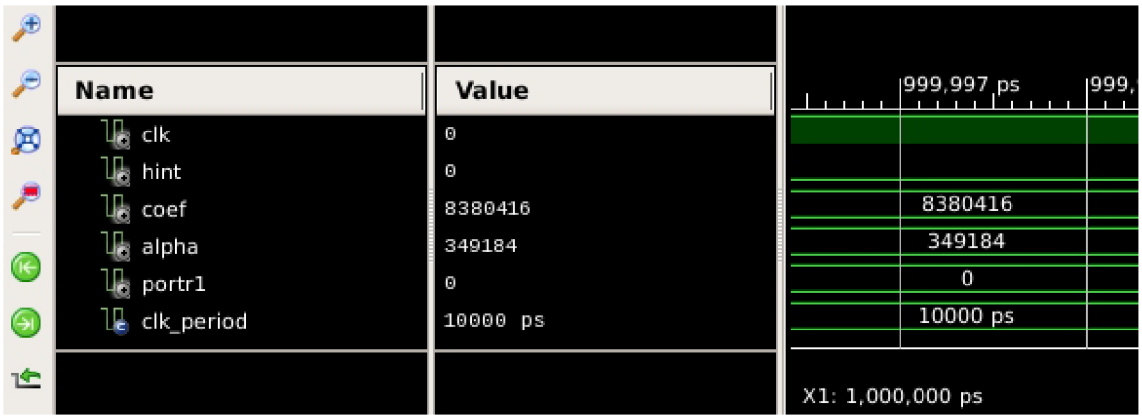
Obr. 7.7: Výřez snímku obrazovky aplikace ISim - testování USEHINT

Pro vyhovění druhé složené podmínce je potřeba změnit vstupní hodnotu na portu `coef`. V druhé simulaci je tato hodnota 8380416. V proměnných `r0` a `r1` jsou výstupy vnořeného modulu `DECOMPOSE` dle tabulky 7.1 -1 a 0. Hodnota jednobitového vstupu `hint` zůstává nezměněna a na základě vyhovění podmínce předpokládáme na výstupu modulu výsledek výrazu $r_1 - 1 \bmod^{+} 24$, tedy 23. Výsledek druhé simulace na obrázku 7.8 potvrzuje výstup modulu dle očekávání.



Obr. 7.8: Výřez snímku obrazovky aplikace ISim - 2. testování USEHINT

Pro simulaci třetího možného stavu je potřeba změnit hodnotu jednobitového vstupu na portu `hint` na '0', v této situaci není ani jedna podmínka vyhodnocena jako pravdivá a výstupem modulu je hodnota proměnné `r1`, která je dle tabulky 7.1 pro nezměněnou hodnotu vstupního koeficientu 0. Na obrázku 7.9 je potvrzena třetí očekávaná hodnota výstupu modulu `USEHINT`.



Obr. 7.9: Výřez snímku obrazovky aplikace ISim - 3. testování USEHINT

7.6 POWER2ROUND

Poslední implementovaná dílčí operace Dilithia je také řešena pomocí samostatného modulu. Funkce modulu `POWER2ROUND` (7.9) je velice podobná modulu `DECOMPOSE`. Využívá centralizované modulární redukce a také dělí vstupní koeficient na dvě části, ale jak vyplývá z názvu, je specializován pro situaci, kdy je tato operace prováděna vzhledem k mocnině čísla 2. Vstupem modulu není parametr α, který nahradila celočíselná reprezentace exponentu, na který je umocňováno číslo 2. Kromě vstupního portu pro exponent a nutného portu pro hodinový signál, má modul port `coef`, určený pro vstupní celočíselný koeficient a dva výstupní porty `pr` a `pr0`, které také operují s datovým typem `integer`. Samotná funkce modulu je řešena pomocí procesu závislém na hodinovém signálu a jsou definovány dvě celočíselné pomocné proměnné `r` a `r0` pro mezivýsledky. Do první z nich je ukládána výsledná hodnota výrazu $r \mod^{+} q$, druhá z nich slouží pro výsledek centralizované modulární redukce $r \mod^{\pm} 2^d$, kde d je exponent ze vstupního portu modulu. Výstupem modulu jsou dvě hodnoty datového typu `integer` na výstupních portech, kdy první je výsledkem výrazu $(r - r_0)/2^d$ a druhá je hodnota proměnné `r0`.

7.6.1 Testování modulu POWER2ROUND

Modul byl testován formou simulace za použití parametrů převzatých ze specifikace Dilithia. Hodnota parametru q zůstává stejná jako při simulaci modulu `USEHINT`, parametr d pužívaný tvůrci Dilithia má hodnotu 14. Byly zvoleny dvě náhodné hodnoty vstupního koeficientu vedeného na port `coef`, první z nich je 15672, která je menší než 2^{14}, druhou hodnotou je číslo 8112357.

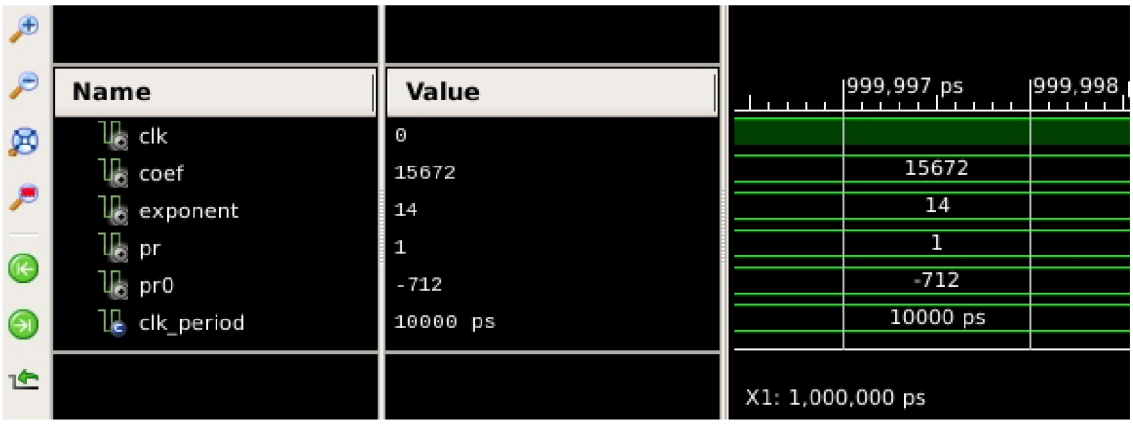
Výpis 7.9: Modul POWER2ROUND

```
1  entity POWER2ROUND is
2  port( clk      : in std_logic;
3        coef     : in integer;
4        exponent : in integer;
5        pr       : out integer;
6        pr0      : out integer);
7  end POWER2ROUND;
8
9  architecture P2R_BEH of POWER2ROUND is
10 begin
11 sequence: process(clk) is
12     variable r : integer;
13     variable r0: integer;
14 begin
15   if rising_edge(clk) then
16     r := mod_p(coef, q);
17     r0:= mod_pm(r, 2**exponent);
18     pr <= (r-r0)/2**exponent;
19     pr0<= r0;
20   end if;
21 end process sequence;
22 end P2R_BEH;
```

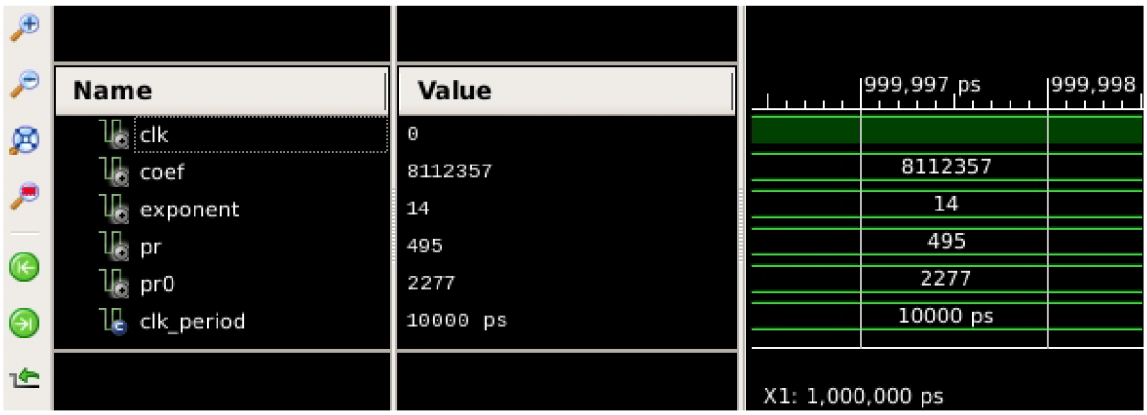
Hodnota prvního vstupního koeficientu je menší než parametr q, prvním výrazem tedy prochází nezměněn. Následuje centralizovaná modulární redukce hodnotou $2^{14} = 16384$. Prostor výsledků pro tuto hodnotu má tvar $-\frac{16384}{2} < r_0 \leq \frac{16384}{2}$, výsledek $15672 \mod 16384 = 15672$ do tohoto prostoru nepatří, je tedy nutné odečíst 16384, kdy konečný výsledek v proměnné `r0` je -712. První z dvojice výstupních hodnot je výsledkem výrazu $\frac{(15672-(-718))}{16384}$ tedy 1. Druhou výstupní hodnotou je obsah proměnné `r0`, což je -712.

Druhý vstupní koeficient také přeskakuje první výraz, ze stejného důvodu jako první. Výsledek operace $8112357 \mod 16384$ je 2277. Tato hodnota patří do výše

zmíněného centralizovaného prostoru výsledků, není ji tedy nutné měnit. Výstupy modulu by měly být $\frac{8112357-2277}{16384} = 495$ a 2277. Z výsledku simulací na obrázcích 7.10 a 7.11 je patrné, že očekávané hodnoty souhlasí s výstupy modulu.



Obr. 7.10: Výřez snímku obrazovky aplikace ISim - testování POWER2ROUND



Obr. 7.11: Výřez snímku obrazovky aplikace ISim - 2. testování POWER2ROUND

Závěr

Teoretická část bakalářské práce je členěna do pěti kapitol. První z nich se věnuje technologii FPGA, jejímu popisu, procesu návrhu funkcionality a stručnému seznámení se síťovou kartou, která je cílovým zařízením implementace praktické části. Druhá kapitola je věnována seznámení s principy a specifiky jazyka VHDL, spolu s popisem vývojového prostředí použitého pro realizaci praktické části. Třetí kapitola pojednává o pěti kategoriích dnešní postkvantové kriptografie, stejně jako o základních principech fungování kvantových počítačů a dvou nejznámějších algoritmech, které ohrožují dnes hojně využívané kryptografické algoritmy. Následuje kapitola věnovaná digitálnímu podpisu jako takovému, která obsahuje definice jeho funkcí a fungování. Poslední kapitola teoretické části je věnována algoritmu CRYSTALS-Dilithium. Jedná se o postkvantové schéma digitálního podpisu založené na mřížkách ve spojení s matematickými principy uzavřených polynomiálních polí. Je předveden zjednodušený postup tří standardních fází fungování podpisových schémat a posléze jsou představeny algoritmy a operace nutné pro realizaci funkční implementace.

Praktickou částí bakalářské práce jsou hardwarová implementace funkce rozšiřitelného výstupu SHAKE spolu s implementací dílčích operací podpisového schématu CRYSTALS-Dilithium.

K implementaci funkce SHAKE bylo přistupováno pomocí funkcí a procedur jazyka VHDL. Tento přístup umožnil přemýšlet o prováděných instrukcích jako o posloupnostech, dalším důvodem tohoto přístupu byla přehlednost. Prvním krokem byla permutace KECCAK, která se skládá z pěti dílčích permutačních algoritmů θ, ρ, π, χ a ι. Permutace KECCAK i dílčí permutace byly realizovány pomocí procedur jazyka VHDL, kdy pro reprezentaci dat ve formě trojrozměrného stavového pole, nad kterým je definována operace dílčích permutací, byl definován vlastní datový typ a konverzní procedury pro převod bitového řetězce do stavového pole. Tato posloupnost procedur byla později zapouzdřena do funkce testovací konstrukce houby, s předem známým objemem vstupních i výstupních dat, a tedy známým počtem absorpčních a vymačkávacích fází. Toto řešení bylo zvoleno z důvodu vlastnosti návrhu v jazyce VHDL, kdy není možné, jako v klasických programovacích jazycích, určovat velikosti vstupních dat nebo počet komponent nutných pro vykonání algoritmu za běhu programu. Nejedná se totiž o vykonávání programu, ale návrh hardwarových komponent. I přes předem známou délku vstupních dat byla implementována funkce pro výplň, dle standardu FIPS PUB 202 [11] a v rámci testovací konstrukce houby využita. Bylo dosaženo modularity návrhu, jelikož obě varianty funkce rozšiřitelného výstupu SHAKE využívané v rámci Dilithia mohou využívat stejné implementace KECCAK. Rozdíl je pouze ve velikosti parametru přenosové rychlosti, respektive kapacity, se kterým pracuje nadřazená konstrukce houby.

K testování implementace SHAKE byla zvolena metoda simulace modulu VHDL, zapouzdřujícího hierarchii testovací konstrukce houby a vnořených permutačních procedur. Výstupy simulace byly porovnány s výstupem volně dostupné implementace SHAKE v podobě webové aplikace. Výsledky testování prokazují nedostatky vlastní implementace. Vzhledem k nedokonalosti testovací metody to sice není naprosto jisté, ale už tvar výstupu vlastní implementace poukazuje na problém, buď uvnitř testovací konstrukce houby nebo hierarchie permutačních procedur. Vzhledem k obtížnosti sledování průběhu vstupních dat napříč implementací, v kombinaci s časovým hlediskem, nebyla provedena další analýza.

V rámci implementace dílčích operací Dilithia byly dokončeny VHDL moduly zprostředkovávající operace pro kompresi veřejného klíče. V tomto případě byl kombinován strukturální a sekvenční přístup. Moduly operací Decompose, HighBits, LowBits, Usehint a Power2Round byly úspěšně otestovány pomocí simulace a porovnáním výsledků s předpokládanými hodnotami. Simulace modulu MakeHint odhalila nedostatky jeho návrhu.

Cíle bakalářské práce, kterým byla funkční hardwarová implementace dílčích částí a procedur podpisového schéma CRYSTALS-Dilithium, bylo z velké části dosaženo. V rámci realizace došlo k seznámení s technologií FPGA, stejně jako s jazykem VHDL, v míře umožňující realizaci výsledků bakalářské práce. Byla také prozkoumána oblast postkvantové kryptografie a podpisové schéma CRYSTALS-Dilithium coby jejího zástupce. Výstupy bakalářské práce jsou VHDL moduly operací určených ke kompresi veřejného klíče. U všech byla prokázána funkčnost, s vyjímkou modulu operace MakeHint. Dále hardwarové řešení funkce rozšířitelného výstupu SHAKE, kde proces testování odhalil nedostatky v implementaci.

Literatura

[1] AGARWAL, Tarun.: *Basic FPGA Architecture and its Applications*[online].[cit.29. 2 2020]. Dostupné z URL: <https://www.edgefx.in/fpga-architecture-applications/>

[2] ARAR, Steve.: *Clock Signal Management: Clock Resources of FPGAs*[online].2018,[cit.2. 3 2020]. Dostupné z URL: <https://www.allaboutcircuits.com/technical-articles/clock-management-clock-resources-of-fpgas/>

[3] Netcope Technologies.: *NFB-200G2QL Product Brief*[online].[cit.3. 3 2020]. Dostupné z URL: <https://www.netcope.com/getattachment/bb2b8efa-9925-438d-b895-897d7c1e4745/NFB-200G2QL-product-brief.aspx>

[4] MEALY, Bryan, TAPPERO, Fabrizio.: *Free Range VHDL*. Free Range Factory, 2012.

[5] University of California.: *VHDL Guides*[online].[cit.5. 3 2020]. Dostupné z URL: <https://www.ics.uci.edu/~jmoorkan/vhdlref/>

[6] BURDA, Patrik.: *Možnosti postkvantové kryptografie* Brno, 2019, 38 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací

[7] POPELOVÁ, Lucie.: *Metody postkvantové kryptografie* Brno, 2018, 55 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací

[8] HOVANOVÁ, Tatiana.: *Kvantově bezpečná kryptografie* Brno, 2019, 61 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací

[9] MATULA, Lukáš.: *Post-kvantová kryptografie na omezených zařízeních* Brno, 2019, 57 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací

[10] L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, D. Stehlé.: *CRYSTALS-Dilithium algorithm Specifications and Supporting Documentation*[online].[cit.3. 4 2020]. Dostupné z URL: <https://pq-crystals.org/dilithium/resources.shtml>

[11] Information Technology Laboratory, National Institute of Standards and Technology.: *Federal Information Processing Standards Publication: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* Gaithersburg, 2015[online].[cit.12. 4 2020]. Dostupné z URL: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>

[12] *Obrázek konstrukce houby*[online].[cit. 15. 4. 2020]. Dostupné z URL: <https://keccak.team/images/Sponge-150.png>

[13] Nayuki.: *Number-theoretic transform (integer DFT)*[online].2017,[cit. 16. 4. 2020]. Dostupné z URL: <https://www.nayuki.io/page/number-theoretic-transform-integer-dft?fbclid=IwAR1Sg9UKryzQn3zIAB2TbyDyoHh4EsrQ32PYqT1r1fhZYHnoQlJIG_0VS3U>

[14] *Implementace funkce SHAKE-128 ve formě webové aplikace*[online].[cit. 5. 6. 2020]. Dostupné z URL: <https://emn178.github.io/online-tools/shake_128.html>

[15] *Převodník z hexadecimální do binární číselné soustavy ve formě webové aplikace*[online].[cit. 5. 6. 2020]. Dostupné z URL: <https://www.rapidtables.com/convert/number/hex-to-binary.html>

Seznam symbolů, veličin a zkratek

FPGA programovatelné hradlové pole – Field Programmable Gate Array

VHDL programovací jazyk VHDL – Very High Speed Integrated Circuilt Hardware Description Language

NTT forma diskrétní Fourierovy transformace zobecněná na použití v konečných polích – number theoretic transform

ASIC zákaznický integrovaný obvod – Application Specific Integrated Circuit

HDL jazyk popisující hardware – Hardware Description Language

C programovací jazyk C – C programming language

C++ programovací jazyk C++ – C++ programming language

I/O vstup/výstup – Input/Output

SRAM statická paměť – Static Random Access Memmory

PCI sběrnice PCI – Peripheral Component Interconnect

RSA asymetrický kryptosystém RSA – Rivest-Shamir-Adleman

ECDH protokol ustanovení klíče Diffie-Hellman postavený na eliptických křivkách – Eliptic-curve Diffie-Hellman

AES symetrický kryptosystém AES – Advanced Encryption Standard

SHA-3 standardizovaný hashovací algoritmus – Secure Hash Algorithm 3

SIDH protokol ustanovení klíče Diffie-Hellman postavený na supersingulárních eliptických křivkách – Supersingular Isogeny Diffie-Hellman

SVP problém nejkratšího vektoru – Shortest Vector Problem

CVP problém nejbližšího vektoru – Closest Vector Problem

LWE učení s chybami – Learning with errors

RLWE učení s chybami nad polynomiálním kruhem – Ring LWE

SHAKE funkce rozšířitelného výstupu v rámci SHA-3 – SHA-3 Extendable Output Function

FIPS kolekce standardizovaných postupů pro zpracování informací federální vlády spojených států amerických – Federal Information Processing Standards

PUB publikace – Publication

KECCAK hashovací algoritmus používaný SHA-3

NIST národní úřad pro standardizaci spojených států amerických – National Institute of Standards and Technology

A Obsah příloh

Přílohy obsahují výstupy praktické části bakalářské práce. V obou adresářích je soubor UTIL.vhd (respektive UTILS.vhd) který obsahuje definice funkcí a procedur nutných k správnému fungování implementací. Dále jsou v nich definovány vlastní datové typy a konstanty využívané napříč implementacemi. Soubory, jejichž názvy končí řetězcem "tb" jsou moduly TestBench využívané pro testování vlastních modulů.

K realizaci a testování výstupů bakalářské práce bylo použito vývojové prostředí ISE WebPACK Design Suite 14 společnosti Xilinx. Součástí příloh je i zdrojový kód samotné bakalářské práce.

```
/ ..... kořenový adresář příloh
├── Dílčí operace ..... implementace dílčích operací
│   ├── DECOMPOSE.vhd
│   ├── DECOMPOSEtb.vhd
│   ├── HIGHBITS.vhd
│   ├── HIGHBITStb.vhd
│   ├── LOWBITS.vhd
│   ├── LOWBITStb.vhd
│   ├── MAKEHINT.vhd
│   ├── MAKEHINTtb.vhd
│   ├── POWER2ROUND.vhd
│   ├── POWER2ROUNDtb.vhd
│   ├── USEHINT.vhd
│   ├── USEHINTtb.vhd
│   └── UTILS.vhd
├── SHAKE ..... implementace funkce SHAKE
│   ├── SHAKEtb.vhd
│   ├── testSHAKE128.vhd
│   └── UTIL.vhd
└── Zdrojový kód a PDF bakalářské práce
    ├── Latex.zip
    ├── Postkvantová kryptografie na FPGA - Sanjin Kek.pdf
    └── Postkvantová kryptografie na FPGA - Sanjin Kek(tisk).pdf
```