VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

BAKALÁŘSKÁ PRÁCE

Brno, 2020 Petr Ohnút

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY
DEPARTMENT OF CONTROL AND INSTRUMENTATION

IMPLEMENTACE ALGORITMU SHA-3 DO FPGA
IMPLEMENTATION OF SHA-3 ALGORITHM IN FPGA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **Petr Ohnút**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Soběslav Valach**
SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Automatizační a měřicí technika**

Ústav automatizace a měřicí techniky

Student: Petr Ohnút ***ID:*** 203310
Ročník: 3 ***Akademický rok:*** 2019/20

NÁZEV TÉMATU:

Implementace algoritmu SHA-3 do FPGA

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je seznámení se s hašovací funkcí SHA-3 (algoritmus Keccak) a jazykem VHDL.
1) Seznamte se s hašovací funkcí SHA-3 (algoritmus Keccak)
2) Prostudujte existující metody implementace algoritmu pro FPGA
3) Vyberte vhodnou implementaci algoritmu pro PC a verifikujte funkcionalitu oproti zlatému standardu
4) Posuďte výpočetní náročnost existujících implementací a požadavky na zdroje v FPGA
5) Navrhněte vlastní model implementace algoritmu pro FPGA (zvažujte využití klopných obvodů, mohutnost LUT, blokových pamětí, DSP bloků a jiných zdrojů existujících v FPGA)
6) Navržený model implementujte ve vyšším programovacím jazyku a ověřte funkcionalitu řešení
7) Proveďte převod do HDL, uvažujte s parametrizací modulu
8) Simulací na úrovni RTL ověřte funkcionalitu
9) Implementujte modul pro cílovou platformu
10) Ověřte funkcionalitu v reálné aplikaci a dosažené výsledy z hlediska pracovní frekvence, spotřeby zdrojů FPGA a rychlosti výpočtu – propustnosti modulu (latence a zpoždění)

DOPORUČENÁ LITERATURA:

[1] DWORKIN, Morris J. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. 2015.

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

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

Vedoucí práce: Ing. Soběslav Valach

doc. Ing. Václav Jirsík, CSc.
předseda rady studijního programu

ABSTRAKT

Bakalářské práce je zaměřena na popis algoritmu SHA3, technologie FPGA a možností implementovat algoritmus SHA3 do FPGA. Dále se zabýváme vlastním návrhem a implementací v jazyce Python a VHDL.

KLÍČOVÁ SLOVA

FPGA, Kryptografie, SHA-3

ABSTRACT

This Bachelors's thesis is focused on the description of SHA3 algorithm, an FPGA technology, and the possibility to implement the SHA3 algorithm into FPGA. It also deals with our design and implementation in Python and VHDL.

KEYWORDS

FPGA, Cryptography, SHA-3

OHNÚT, Petr. *Implementace algoritmu SHA-3 do FPGA*. Brno, 2020, 57 s. Bakalářská práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce: Ing. Soběslav Valach

PROHLÁŠENÍ

Prohlašuji, že svou bakalářskou práci na téma „Implementace algoritmu SHA-3 do 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 8.6.2020 .....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu bakalářské práce panu Ing. Soběslavu Valachovi, rodině a kolegům z Netcope Technologies za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno 8.6.2020 .....

podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů

Úvod

Bakalářská práce se zabývá návrhem implementace algoritmu SHA3 do FPGA.

Téma bylo zadáno externí firmou Netcope Technologies a.s. v rámci projektu Modulární hardwarový akcelerátor pro kryptografické operace.

Cílem práce je se seznámit podrobně s funkcí algoritmu SHA3, technoligí FPGA a jejími dostupnými zdroji a provést návrh komponenty, kterou bude možno převést do HDL jazyka, korektně ji odsimulovat a následně implementovat do FPGA čipu. Pro implementaci byla vybrána funkce SHA3-512. Funkce byla zvolena převážně proto, že šance kolize je nejmenší a zároveň nejbezpečnější.

V práci jsme seznámeni s blokovým návrhem implementace, s výsledky již navržených implementací, vlastní Python implementací a následnou námi navrženou implementací v jazyce VHDL. Je popsána funkcionalita blokového návrhu společně se všemi signály, které návrh obsahuje, simulace jednotlivých komponent a následně simulace komponenty společně s již existující aplikací.

Pro implementaci byla zvolena vysokorychlostní síťová karta na bázi FPGA technologie, která slouží ke zpracovávání datového provozu. Dochází k zahašování paketů putujících kartou. Jak již bylo zmíněno tak komponenta vznikla v rámci projektu a vyžívá další externí komponenty. Chování externí komponenty je odsimulována simulacemi. Zvolenou platformou je karta NFB-200G2QL, která je osazena čipem Vitex 7 Ultrascale+ od firmy Xilinx.

V poslední kapitole jsme seznámeni s dosaženými výsledky, které jsou okomentovány a zhodnoceny.

1 Hašovací funkce

Základní vlastností hašovací funkce h je vytvořit z libovolně dlouhé vstupní zprávy z takzvaný haš **H** o pevně dané délce. Tenhle haš musí být pro každou zprávu jedinečný. Pro dosažení eliminace co nejvíce kolizí hašovací funkcí je vhodné, aby haš byl dlouhý nejméně 128-bitů. Pro menší délku by byla velká pravděpodobnost kolize. Kolizi chápeme jako stejný výsledek pro více rozdílných zpráv. Tohle platí pro aplikace, kde není potřeba vysokého zabezpečení. Typická délka výstupu je pak 160-bitů. Kryptografická hašovací funkce dle [2] musí mít následující vlastnosti :

- Z vytvořeného haše nelze zjistit vstupní zpráva
- Nelze najít rozdílné vstupy pro které by byl vytvořen stejný haš
- Ze znalosti zprávy a jejího haše nelze najít rozdílnou zprávu o stejném haši

K velmi oblíbenémuu vytváření kryptografickách hašovacích funkcí patří takzvaná Merkle–Damgårdova konstrukce. Tímhle způsobem vznikly například hašovací funkce MD5 nebo SHA-1.

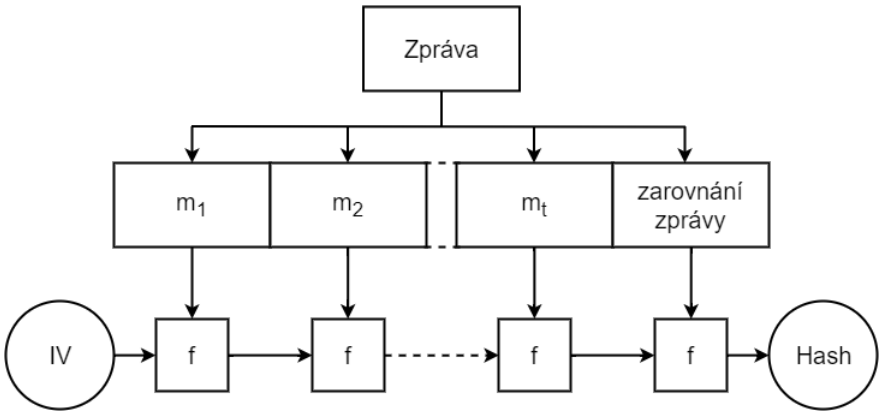
Merkle–Damgårdova konstrukce
Předpokládejme funkci f, která je kompresí z s bitů na n bitů. Funkce f se použije ke konstrukci hašovací funkce h, která je schopna přijmout vstup o libovolné délce a vytvořit haš o délce n bitů. Princip konstrukce je popsán z [2] a je následující :

1. Nechť $l = s$ - n
2. Vstupní zpráva m se zarovná 0 tak, aby celková délka byla násobkem délky l
3. Vstup m se rozdělí do t bloků o délce l
4. Nechť H je řetězec o délce n
5. Pro $i \in \langle 1, t \rangle : H = f(H||m_i)$
6. Vrať H

Na obrázku 1.1 si můžeme konstrukci prohlédnout. V obrázku symbol IV značí inicializační vektor. V algoritmu je hodnota H většinou nazývána *vnitřním stavem* haš funkce. V každé iteraci se *vnitřní stav* mění pomocí aktuálního *vnitřního stavu* a dalšího bloku zprávy m, na které se aplikuje kompresní funkce. Na konci cyklu je *vnitřní stav* H výstupem hašovací funkce. [2]

1.1 Secure hash Algorithm-3

Jsou definovány čtyři hašovací funkce SHA-3 pomocí algoritmu nazvaného KECCAK[c]. Zpráva M je zarovnána dvouma bity 01 a kapacita c je dvojnásobkem požadované délky haše d, takže platí $c = 2d$. V rovnicích (1.1) - (1.4) jsou tyhle funkce popsány.



Obr. 1.1: Princip Merkle-Damgårdovi konstrukce

[1]

$$\text{SHA3-224}(M)=\text{KECCAK}[448](M||01,224); \tag{1.1}$$

$$\text{SHA3-256}(M)=\text{KECCAK}[512](M||01,256); \tag{1.2}$$

$$\text{SHA3-384}(M)=\text{KECCAK}[768](M||01,384); \tag{1.3}$$

$$\text{SHA3-512}(M)=\text{KECCAK}[1024](M||01,512). \tag{1.4}$$

1.1.1 Permutace KECCAK-p

Permutace je popsána z [1] a je dána předpisem KECCAK-p[b, n_r], kde n_r je počet iterací funkce a $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ je šířka permutace. Cyklus permutací, definován *Rnd*, zahrnuje pět transformací. Pro SHA-3 platí $b = 1600$ a $n_r = 24$.

Stavové pole

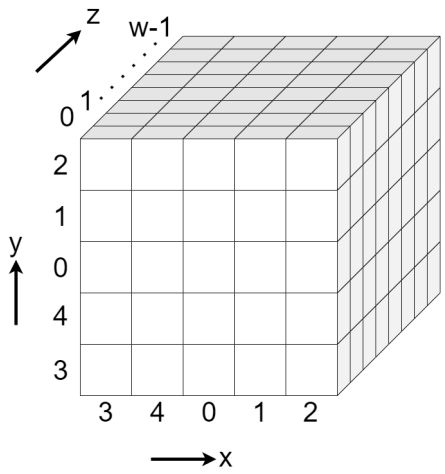
Stavové pole permutace KECCAK-p[b, n_r] se skládá z b bitů. Je definovaná jeho hloubka $l = \log_2 \frac{b}{25}$ a šířka $w = \frac{b}{25}$. Hodnoty l a w pro každé b jsou uvedeny v tab 1.1. [1]

b	25	50	100	200	400	800	1600
w	1	2	4	8	16	32	64
l	0	1	2	3	4	5	6

Tab. 1.1: Hodnoty parametrů stavového pole

Vstupní řetězec S tvořen z b bitů s indexací od 0 do b-1, konkrétněji $S = S[0] \,||\, S[1] \,||\, ... \,||\, S[b-2] \,||\, S[b-1]$, je přeskupen do trojrozměrného prostoru o šířce $x \in \langle 0, 5\rangle$, výšce $y \in \langle 0, 5\rangle$ a hloubce $z \in \langle 0, w\rangle$. Vznikne tedy pravidelný čtyřboký hranol, který nazveme **A**. Každý bit v prostoru je určen pomocí souřadnice

A=[x, y, z]. Souřadnice na osách x a y jsou v následujícím pořadí: 3, 4, 0, 1, 2. Pro z je řazení vzestupné 0 až w-1. [3]



Obr. 1.2: Stavové pole

Konverze řetězce na stavové pole

Stavové pole je vytvořeno z jednotlivých bitů v řetězci S, pomocí rovnice (1.5) z [1].

$$\mathbf{A}[x,y,z]=S[w(5y+x)+z] \tag{1.5}$$

Příklad pro b=1600 a w=64:

$$\begin{array}{ccccc}
\mathbf{A}[0,0,0]=S[0] & & \mathbf{A}[1,0,0]=S[64] & & \mathbf{A}[4,0,0]=S[256] \\
\mathbf{A}[0,0,1]=S[1] & \ldots & \mathbf{A}[1,0,0]=S[64] & \ldots & \mathbf{A}[4,0,1]=S[257] \\
\mathbf{A}[0,0,2]=S[2] & & \mathbf{A}[1,0,0]=S[64] & & \mathbf{A}[4,0,2]=S[258] \\
\vdots & & \vdots & & \vdots \\
\mathbf{A}[0,1,0]=S[320] & & \mathbf{A}[1,1,0]=S[384] & & \mathbf{A}[4,1,0]=S[576] \\
\mathbf{A}[0,1,1]=S[321] & \ldots & \mathbf{A}[1,1,1]=S[385] & \ldots & \mathbf{A}[4,1,1]=S[577] \\
\mathbf{A}[0,1,2]=S[322] & & \mathbf{A}[1,1,2]=S[386] & & \mathbf{A}[4,1,2]=S[578] \\
\vdots & & \vdots & & \vdots \\
\mathbf{A}[0,2,61]=S[701] & & \mathbf{A}[1,2,61]=S[765] & & \mathbf{A}[4,2,61]=S[957] \\
\mathbf{A}[0,2,62]=S[702] & \ldots & \mathbf{A}[1,2,62]=S[766] & \ldots & \mathbf{A}[4,2,62]=S[958] \\
\mathbf{A}[0,2,63]=S[703] & & \mathbf{A}[1,2,63]=S[767] & & \mathbf{A}[4,2,63]=S[959]
\end{array}$$

atd.

Konverze stavového pole na řetězec

Stavové pole **A** lze konvertovat zpátky na řetězec S algoritmem z [1]. Konverze probíhá následovně: nejdříve vytvoříme řetězec $\check{R}ada(i, j)$, který je definován v rovnici (1.6) a pro který platí $i \in \langle 0, 4\rangle$ a $j \in \langle 0, 4\rangle$.

$$\check{R}ada(i, j) = \mathbf{A}[i, j, 0] \,||\, \mathbf{A}[i, j, 1] \,||\, \mathbf{A}[i, j, 2] \,||\, ... \,||\, \mathbf{A}[i, j, w\text{-}2] \,||\, \mathbf{A}[i, j, w\text{-}1] \tag{1.6}$$

Příklad pro b=1600 a w=64:

$$\begin{aligned}
\check{R}ada(i, j) &= \mathbf{A}[0, 0, 0] \,||\, \mathbf{A}[0, 0, 1] \,||\, \mathbf{A}[0, 0, 2] \,||\, ... \,||\, \mathbf{A}[0, 0, 62] \,||\, \mathbf{A}[0, 0, 63] \\
\check{R}ada(i, j) &= \mathbf{A}[1, 0, 0] \,||\, \mathbf{A}[1, 0, 1] \,||\, \mathbf{A}[1, 0, 2] \,||\, ... \,||\, \mathbf{A}[1, 0, 62] \,||\, \mathbf{A}[1, 0, 63] \\
\check{R}ada(i, j) &= \mathbf{A}[2, 0, 0] \,||\, \mathbf{A}[2, 0, 1] \,||\, \mathbf{A}[2, 0, 2] \,||\, ... \,||\, \mathbf{A}[2, 0, 62] \,||\, \mathbf{A}[2, 0, 63]
\end{aligned}$$

atd.

$Plocha(j)$ popsána vztahem (1.7), kde $i \in \langle 0, 4\rangle$, je rovina rovnoběžná s osou x a z stavového pole.

$$Plocha(j) = \check{R}ada(0, j) \,||\, \check{R}ada(1, j) \,||\, \check{R}ada(2, j) \,||\, \check{R}ada(3, j) \,||\, \check{R}ada(4, j) \tag{1.7}$$

Výsledný řetězec S má pak podobu:

$$S = Plocha(0) \,||\, Plocha(1) \,||\, Plocha(2) \,||\, Plocha(3) \,||\, Plocha(4)$$

Příklad pro b=1600 a **w**=64:

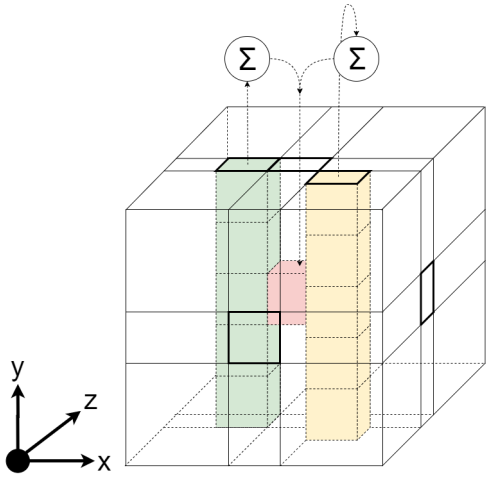
$$\begin{aligned}
S = \quad & \mathbf{A}[0,0,0] \,||\, \mathbf{A}[0,0,1] \,||\, \mathbf{A}[0,0,2] \,||\, ... \,||\, \mathbf{A}[0,0,62] \,||\, \mathbf{A}[0,0,63] \\
& ||\, \mathbf{A}[1,0,0] \,||\, \mathbf{A}[1,0,1] \,||\, \mathbf{A}[1,0,2] \,||\, ... \,||\, \mathbf{A}[1,0,62] \,||\, \mathbf{A}[1,0,63] \\
& ||\, \mathbf{A}[2,0,0] \,||\, \mathbf{A}[2,0,1] \,||\, \mathbf{A}[2,0,2] \,||\, ... \,||\, \mathbf{A}[2,0,62] \,||\, \mathbf{A}[2,0,63] \\
& \qquad\qquad\qquad\qquad\qquad\qquad \vdots \\
& ||\, \mathbf{A}[2,4,0] \,||\, \mathbf{A}[2,4,1] \,||\, \mathbf{A}[2,4,2] \,||\, ... \,||\, \mathbf{A}[2,4,62] \,||\, \mathbf{A}[2,4,63] \\
& ||\, \mathbf{A}[3,4,0] \,||\, \mathbf{A}[3,4,1] \,||\, \mathbf{A}[3,4,2] \,||\, ... \,||\, \mathbf{A}[3,4,62] \,||\, \mathbf{A}[3,4,63] \\
& ||\, \mathbf{A}[4,4,0] \,||\, \mathbf{A}[4,4,1] \,||\, \mathbf{A}[4,4,2] \,||\, ... \,||\, \mathbf{A}[4,4,62] \,||\, \mathbf{A}[4,4,63]
\end{aligned}$$

Transformace

Permutace KECCAK-p[b, n_r] se skládá z pěti transformací, popsán z [1], které značíme jako θ, ρ, π, χ a ι. Vstupními daty do každé transformace je stavové pole **A** a výstupem je pole **A'**. Transformace ι má i druhý vstup a to hodnotu dané iterace, ktrou značíme jako i_r. Parametr b je znám, a proto se velikost pole **A** vynechává ze zápisu.

Algoritmus 1: $\theta(\mathbf{A})$

1. Pro všechny dvojice (x,z), kde $x \in \langle 0,4 \rangle$ a $z \in \langle 0,w \rangle$, nechť $C[x,y] = \mathbf{A}[x,0,z] \oplus \mathbf{A}[x,1,z] \oplus \mathbf{A}[x,2,z] \oplus \mathbf{A}[x,3,z] \oplus \mathbf{A}[x,4,z]$.
2. Pro všechny dvojice (x,z), kde $x \in \langle 0,4 \rangle$ a $z \in \langle 0,w \rangle$, nechť $D[x,y] = C[(x-1) \bmod 5,z] \oplus C[(x+1) \bmod 5,(z-1) \bmod w]$
3. Pro všechny trojice (x,y,z), kde $x \in \langle 0,4 \rangle$, $y \in \langle 0,4 \rangle$ a $z \in \langle 0,w \rangle$, nechť $\mathbf{A'}[x,y,z] = \mathbf{A}[x,y,z] \oplus D[x,y]$
4. Vrať **A'**.

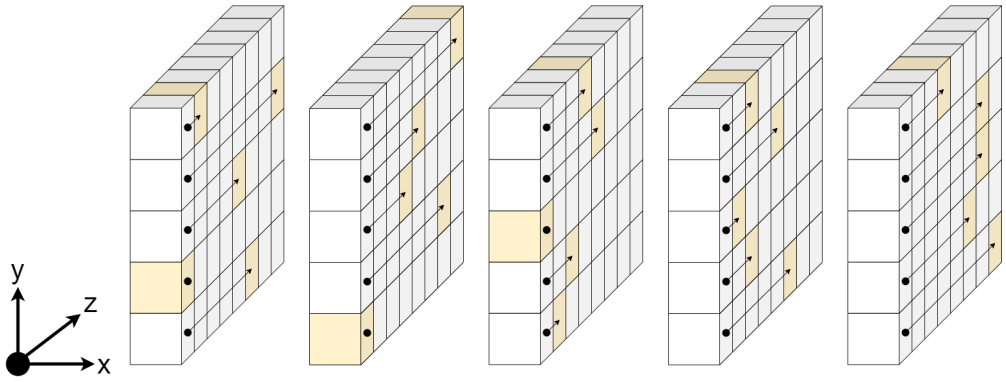


Obr. 1.3: Alogritmus θ

Algoritmus θ provede XOR součet bitů ve dvou sloupcích stavového pole **A**. Výsledek je následně uložen na příslušnou pozici v poli.
Pro bit $\mathbf{A}[x_0,y_0,z_0]$: x-ová souřadnice prvního sloupce je $(x_0 - 1) \bmod 5$ a souřadnice z je z_0 , zatímco souřadnice x sloupce druhého je $(x_0 + 1) \bmod 5$ a z je dáno výpočtem: $(z_0 - 1) \bmod w$.

Algoritmus 2: $\rho(\mathbf{A})$

1. Pro všechna $z \in \langle 0,w \rangle$, nechť $\mathbf{A'}[0,0,z] = \mathbf{A}[0,0,z]$.
2. Nechť $(x,y) = (1,0)$.
3. Pro $t \in \langle 0,23 \rangle$:
   (a) pro všechna $z \in \langle 0,w \rangle$, nechť $\mathbf{A'}[x,y,z] = \mathbf{A}[x,y,\frac{z-(t+1)(t+2)}{2} \bmod w]$
   (b) nechť $(x,y) = (y,(2x+3y) \bmod 5)$
4. Vrať **A'**.



Obr. 1.4: Alogritmus ρ pro $w = 8$

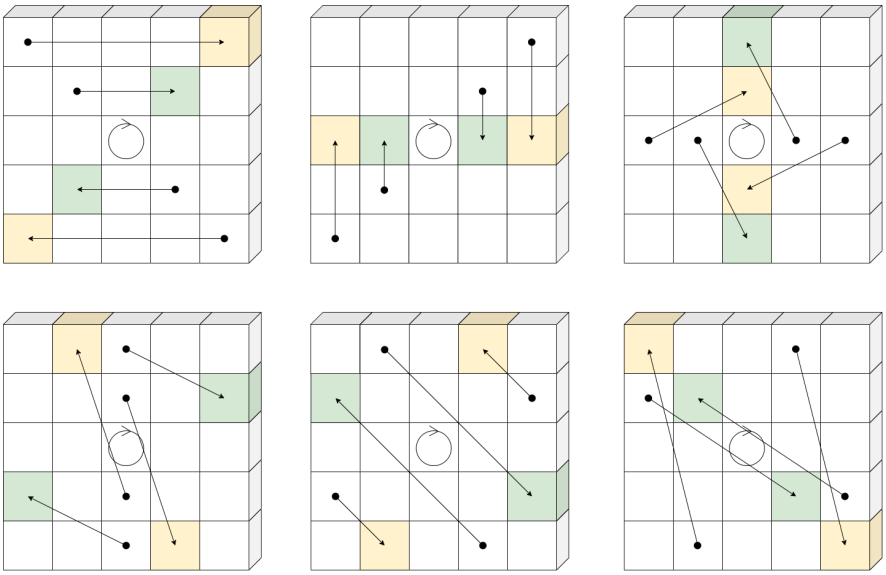
Funkce rotuje všechny bity podél řady o délku zvanou *offset*, která je fixně dána souřadnicemi x a y každé řady. Pro každý bit v řadě je z souřadnice dána součtem *offsetu* modulo délkou řady w s původní souřadnicí. V tabulce 1.2 jsou uvedeny hodnoty *offsetu* pro každou řadu. [3]

	$x = 3$	$x = 4$	$x = 0$	$x = 1$	$x = 2$
$y = 2$	153	231	3	10	171
$y = 1$	55	276	36	300	6
$y = 0$	28	91	0	1	190
$y = 4$	120	78	210	66	253
$y = 3$	21	136	105	45	15

Tab. 1.2: Hodnoty offsetu funkce ρ

Algoritmus 3: $\pi(\mathbf{A})$

1. Pro všechny trojice (x,y,z), kde $x \in \langle 0, 4\rangle$, $y \in \langle 0, 4\rangle$ a $z \in \langle 0, w\rangle$, nechť $\mathbf{A'}[x,y,z] = \mathbf{A}[(x+3y) \bmod 5, x, z]$.
2. Vrať $\mathbf{A'}$.

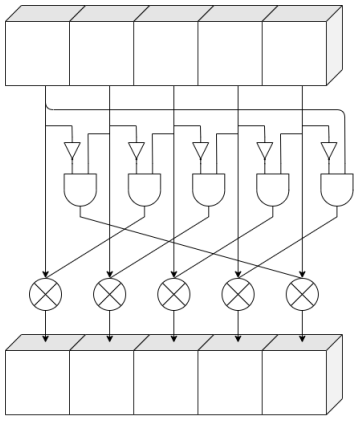


Obr. 1.5: Alogritmus π

Efektem funkce π je přeskupení bitů v rovině, která je rovnoběžná s osami x a y. Na obrázku 1.5 je zobrazen průběh přeskupení. Bit roviny se souřadnicemi $x = 0$ a $y = 0$, je středem, kolem kterého se všechny ostatní bity otáčí.

Algoritmus 4: $\chi(\mathbf{A})$

1. Pro všechny trojice (x,y,z), kde $x \in \langle 0,4\rangle$, $y \in \langle 0,4\rangle$ a $z \in \langle 0,w\rangle$, nechť $\mathbf{A'}[x,y,z] = \mathbf{A}[x,y,z] \oplus ((\mathbf{A}[(x{+}1) \bmod 5,y,z],\oplus 1) \cdot \mathbf{A}[(x{+}2) \bmod 5,y,z])$
2. Vrať $\mathbf{A'}$.



Obr. 1.6: Alogritmus χ

Cílem algoritmu χ je provést XOR každého bitu s dvěma dalšími bity pole, mezi kterými se oprevede operace AND a z nichž jeden má invertovanou hodnotu.

Algoritmus 5: $rc(t)$

1. Pokud $t \bmod 255 = 0$, vrať 1.
2. Nechť $R = 10000000$.
3. Pro $z \in \langle 1, t \; mod \; 255 \rangle$:
   (a) $R = 0 \,||\, R$;
   (b) $R[0] = R[0] \oplus R[8]$;
   (c) $R[4] = R[4] \oplus R[8]$;
   (d) $R[5] = R[5] \oplus R[8]$;
   (e) $R[6] = R[6] \oplus R[8]$;
   (f) $R = Trunc_8[R]$;
4. Vrať $R[0]$.

Funkce rc je součástí algoritmu ι. Vstupními daty je hodnota iterace t a výstupem je bit $R[0]$. Funkce $Trunc_s$(X), užita v algoritmu, jejímž vstupem je řetězec X o délce n má výstupní řetězec bitů X[0] až X[s-1]. Příklad: $Trunc_3(110111) = 110$.

Algoritmus 6: $\iota(\mathbf{A}, i_r)$

1. Pro všechny trojice (x,y,z), kde $x \in \langle 0, 4 \rangle$, $y \in \langle 0, 4 \rangle$ a $z \in \langle 0, w \rangle$, nechť **A'**[$x$,$y$,$z$] = **A**[$x$,$y$,$z$]
2. Nechť $RC = 0^w$.
3. Pro $j \in \langle 0, l \rangle$, nechť $RC[2^j - 1] = rc(j + 7i_r)$
4. Pro všechna$z \in \langle 0, w \rangle$, nechť **A'**[0,0,$z$] = **A'**[0,0,$z$] $\oplus$ $RC[z]$.
5. Vrať **A'**.

Algoritmus ι modifikuje pouze bity v *Řadě*(0,0) v souvislosti s hodnotou iterace i_r. Zápis 0^w značí vektor nulových bitů o délce w.

KECCAK-p[b, n_r]

Je dáno stavové pole **A** a hodnota iterace i_r. V rovnici (1.8) z [1] je popsána funkce *Rnd*, která se skládá z posloupnosti výše popsaných transformací. Permutace Keccak-p[b, n_r] se skládá z n_r iterací funkce *Rnd*.

$$Rnd(\mathbf{A}, i_r) = \iota(\chi(\pi(\rho(\theta)))), i_r) \tag{1.8}$$

Algoritmus 7: KECCAK-p[b, n_r](S)

1. Konvertuj řetězec S na stavové pole **A**.
2. Pro $i_r \in \langle 12 + 12l - n_r, 12 + 12l - 1 \rangle$, nechť **A**=$Rnd$(**A**,$i_r$).
3. Konvertuj stavové pole **A** na řetězec S' o délce b.
4. Vrať S'.

Srovnání s KECCAK-*f*

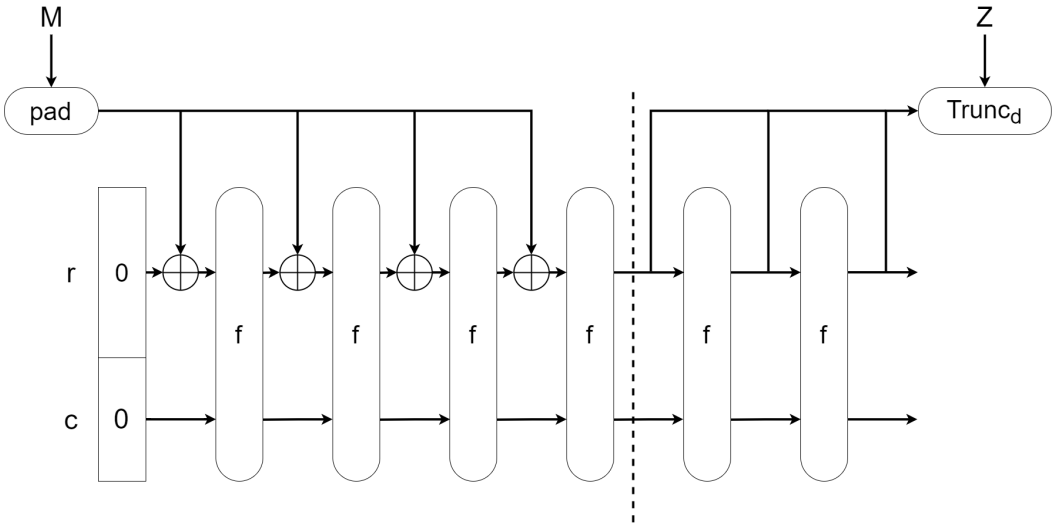
Permutace KECCAK-*f* je speciálním případem KECCAK-p, kde n_r=12+12l . Proto funkce KECCAK-p[1600,24], která popisuje SHA-3 funkce, je shodná s KECCAK-*f*[1600]. Rovnící (1.9) z [1] je zmíněný speciální případ popsán.

$$\text{KECCAK-}f\text{= KECCAK-}p[b,12+2l] \tag{1.9}$$

1.1.2 Konstrukce SPONGE

Konstrukce sponge, která je popsána v rovnici (1.10) z [1] a zobrazena na obrázku 1.7 je kostrou pro specifické funkce s binárními vstupními daty a fixní výstupní délkou. Využívá se tří komponent:

- funkce *f*, která má pevně danou délku dat b
- parametru míra, který značíme jako r
- funkce *pad*, která pomocí bitů zarovnánva řetězec na určitou délku



Obr. 1.7: Konstrukce Sponge

Parametr r je kladné celé číslo, které je vždy menší než šířka b. Kapacita c je hodnota pro kterou platí b-r a tudíž b=r+c.

$$Z\text{=SPONGE}[f,pad,r](M,d) \tag{1.10}$$

Zarovnání *pad* funguje tak že: pokud máme funkci *pad*(x,m), kde x je kladnou hodnotou a m je hodnotou nezápornou, pak její výstup je řetězec takový, že m+len(*pad*(x,m)) je kladným násobkem hodnoty x.

Algoritmus 8: Z=SPONGE[f,*pad*,r](M,d)

1. Nechť P=N $||$ pad(r,len(N)).
2. Nechť n=len(P)/r.
3. Nechť c=b-r.
4. Nechť P_0,...,P_{n-1} je sekvence řetězců délky r takových, že P= $P_0 \,||\, ... \,||\, P_{n-1}$.
5. Nechť S=0^b.
6. Pro $r \in \langle 0, n-1 \rangle$, nechť S=$f(S \oplus (P_i \,||\, 0^c))$.
7. Nechť Z je prázdný řetězec.
8. Nechť Z=$Z \,||\, Trunc_r(S)$.
9. Pokud $d \leq |Z|$, vrať $Trunc_d(Z)$; jinak pokračuj.
10. Nechť S=$f(S)$ a vrať se na bod číslo 8.

Vstupními daty jsou řetězec M a nezáporná hodnota d, která určuje kolik bitů funkce vrátí, ale neovlivňuje jejich hodnotu.

1.1.3 KECCAK-c

KECCAK je popsán z [1] funkcemi KECCAK-p[b,12+12l] a pad10*1. Volba parametrů míry r a kapacity c musí být taková, aby r+c odpovídalo jedné z hodnot b z tabulky (1.1) a tedy $b \in \{25, 50, 100, 200, 400, 800, 1600\}$.

Pro b=1600 se jedná o fukci KECCAK-c , viz. rovnice (1.11).

$$\text{KECCAK[c]=SPONGE[KECCAK-p[1600,24]},pad\text{10*1,1600-}c] \tag{1.11}$$

Pří vstupním řetězci M a výstupní délce d,

$$\text{KECCAK[c]}(N,d)\text{=SPONGE[KECCAK-p[1600,24]},pad\text{10*1,1600-}c](N,d) \tag{1.12}$$

Algoritmus 9: pad10*1(x,m)

1. Nechť j=(-m-2) mod x).
2. Vrať P=1 $||$ 0^j $||$ 1.

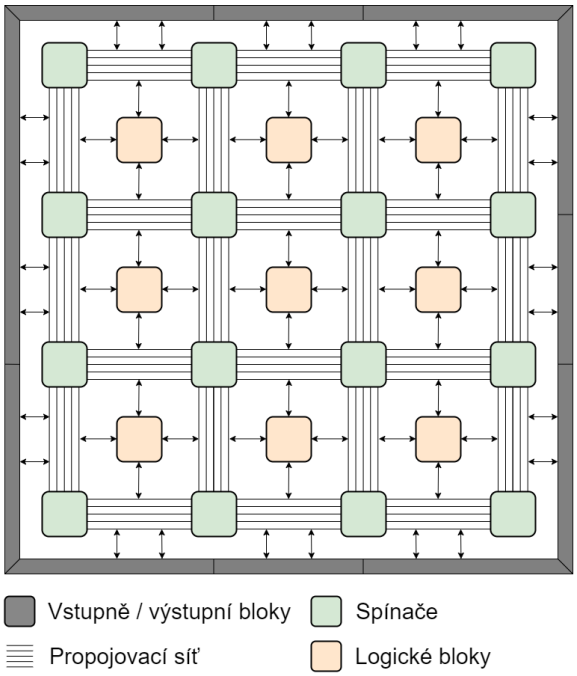
Algoritmus je popsán z [1] a jeho zápis "pad10*1" znamená, že bit 0 je buď úplně vynechán, nebo opakován, a to až do doby dokud není dosaženo požadované délky výstupného řetězce m+len(P).

2 Technologie FPGA

Field-**P**rogrammable **G**ate **A**rray neboli programovatelná hradlová pole patří mezi nejsložitější, ale i nejobecnější obvody typu PLD (**P**rogrammable **L**ogic **D**evice). Jejich hlavní výhodou je možnost být znovu nakonfigurovány. Můžeme zcela lehce zrealizovat a otestovat navržené číslicové zařízení. Při programování FPGA vytváříme vlastně fyzický hardware a tak můžeme stanovit přesný počet komponent, například čítačů, které bude čip obsahovat. Chování hardwaru popisujeme pomocí HDL (**H**ardware **D**escription **L**anguage) jazyků mezi které patří například VHDL či Verilog. FPGA obvody mají velmi širokou škálu využití a to od akcelerování algoritmů, těžení kryptoměn až po zpracovávání obrazu.

2.1 Architektura

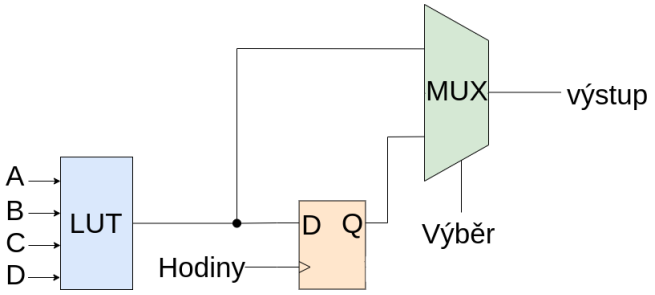
Základem architektury jsou programovatelné bloky, spínače a vstupně / výstupní bloky. Obvody jsou nejčastěji realizovány jako dvourozměrná matice, kde v kanálech mezi logickými bloky se nachází ještě propojavací síť. V posledních letech se FPGA čipy doplňují o rozšiřující funkční bloky (např. paměti, procesory, sčítačky). Většinu obvodů jsme schopni realizovat i v univerzálních logických blocích, ale při této realizaci je spotřebována větší plocha čipu a výsledek by v konečném důsledku nebyl příliš efektivní.[4] Na obrázku 2.1 si můžeme prohlédnout základní architekturu FPGA obvodu.



Obr. 2.1: Základní architektura FPGA obvodu

2.1.1 Programovatelné logické bloky

Hlavním cílem logických bloků je vytvářet logické, paměťové a aritmetické prvky v FPGA čipu. Typicky se skládají z šestivstupého obvodu, který realizuje jednoduché kombinační funkce. Tenhle jednoduchý obvod se nazývá **LUT** - **L**ook **U**p **T**able neboli vyhledávací tabulka. Pomocí LUT jsme schopni realizovat všechny logické kombinační funkce vstupních proměnných. Dalším stavebním prvkem je klopný obvod sloužící k realizaci synchronní sekvenční části (většinou klopný obvod typu D). Součástí logické buňky můžou být i jiné podpůrné struktury (například multiplexor). [4] Na obrázku 2.2 je vyobrazena architektura základní 4 vstupé logické buňky.



Obr. 2.2: Základní čtyř vstupá logická buňka

2.1.2 Programovatelné spínače

Spínače slouží k propojení vstupů a výstupů logických bloků k propojovací síti. Pro propojení sítě se tyhle spínače sdružují v maticích umístěných v křížení vertikálních a horizontálních kanálů. Jak lze mezi sebou spojovat vodiče mezi jednotlivými kanály je dáno architekturou matic. [5]

2.1.3 Vstupně / výstupní bloky

Konfigurovatelné vstupně výstupní bloky zajišťují propojení mezi vývodem obvodu a vnitřní logikou čipu. Bloky mohou být nakonfigurovány jako vstupní, výstupní nebo obousměrné. [4]

2.1.4 Propojovací síť a rozvod hodinového signálu

Všeobecně má propojení podobu mřížky jejíž hrany tvoří propojovací vodiče. Hrany můžeme rozdělit do přibližně 3 kategorií. Nejkratší vodiče propojují pouze sousední bloky. Dále máme vodiče o dvojnásobné až šestinásobné délce a nakonec velmi dlouhé vodiče, které dokáží pohltit až 20 logických buněk a nebo celý řádek či sloupec. [4]

FPGA obvody obsahují samostatnou síť pro rozvod hodinového signálu po celém čipu. Síť je vytvořena tak, aby zajistila co nejmenšní rozdílový čas na hodinových vstupech klopných obvodů. Snažíme se tím snížit riziko nežádoucího chování z důvodů opoždění hodinového signálu do jednotlivých bloků. [5]

2.2 Srovnání s technologií ASIC

Hlavním rozdílem technologie FPGA a ASIC (zákaznicky integrovanými obvody) je cena, rychlost a flexibilita. Mezi hlavní výhody FPGA můžeme zařadit velmi jednoduchou a rychlou reprogramovatelnost. Funkce čipu ASIC je dána už při výrobě mezitím co funkci FPGA můžeme naprogramovat a následně i reprogramovat a vytvořit tak zcela jiné čislicové zařízení. Design na čipu FPGA lze velmi snadno měnit a můžeme tak vyvíjet například prototyp jehož výsledek bude nakonec vyroben jako ASIC. Oproti technologii ASIC jsou FPGA ovšem pomalejší, dražší a mají větší spotřebu. [6]

3 Návrh implementace

V následující kapitole se seznámíme s výsledký již navrhnutých implementací a s blokovým návrhem vlastní implementace. V návrhu je co nejvíce blokových prvků a to právě z důvodu, abychom v budoucnu při implementaci do VHDL mohli velmi snadno otestovat funkčnost každého bloku a zvlášť v případě potřeby jej mohli jednoduše pozměnit bez nutnosti provést drastické změny v návrhu. Konkrétně implementujeme funkci **SHA3-512** a to z důvodů nejvyššího zabezpečení a nejnižší šance na kolizi.

3.1 Navrhnuté implementace

V roce 2014 G. S. Athanasiou, G. Makkas and G. Theodoridis [7] navrhly implementaci algoritmu SHA3-512 pro FPGA. Autoři implementaci odsimulovali a následně ověřili na čipech Virtex-5 (V6), Virtex-6 (V6) a Virtex-7 (V7).

Výsledky jednotlivých imlementací jsou uvedeny v tabulce 3.1, kde **F** - frekvence, **A** - počet slice[1] elementů, **P** - propustnost v gigabitech za sekundu a **P/A** - propustnost v megabitech za sekundu na jeden slice element.

FPGA	**F [MHz]**	**A [Slice]**	**P [Gbps]**	**P/A [Mbps/S]**
V5	389	1702	18,7	10,98
V6	397	1649	19,1	11,6
V7	434	1618	20,8	12,9

Tab. 3.1: Výsledky implementace [7]

Další úspěšnou implementaci ve své práci [8] provedl pan Homer Hsing. Byly navrhnuty dva modely funkce SHA3-512, jeden pro čipy s malou propustností pracujících na nízké frekvenci a druhý pro čipy s velkou propustností pracujících na frekvenci vysoké. Jako cílové platformy byly zvoleny čipy Spartan-3 (S3) pro nízkofrekvenční model a Virtex-6 (V6) pro model vysokofrekvenční. Odsimulování návrhu je popsáno v kapitole 3.2. Hlavní rozdíl mezi implementacemi je počet permutací během jednoho hodinového taktu. Pro čip Spartan-3 je spočítána jedna permutace a pro Virtex-6 jsou spočítány permutace dvě.

[1] Slice obsahuje několik LUTek a klopných obvodů. Může obsahovat i podpůrné obvody, například registry. Architektura je dána výrobcem.

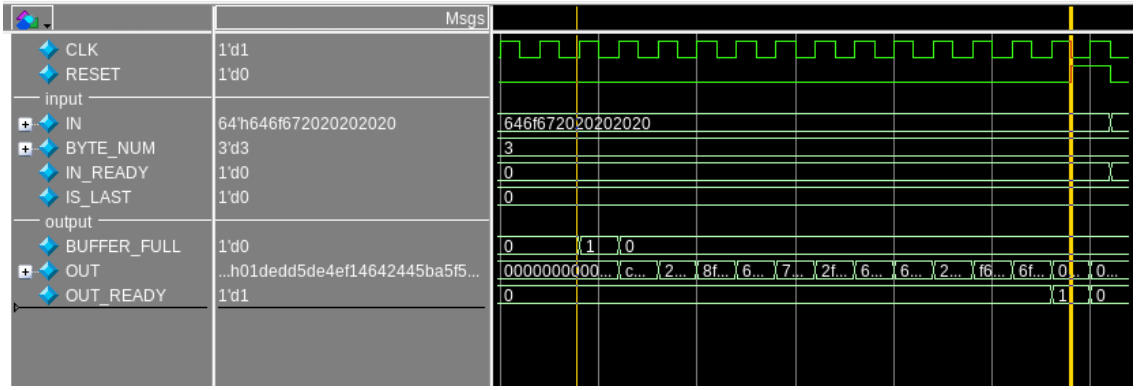
V tabulce 3.2, kde **F** - frekvence, **A** - počet slice elementů, **P** - propustnost v gigabitech za sekundu, **LUT** - počet spotřebovaných LUTek, **IOBs** - počet spotřebovaných vstupně/výstupních bloků , jsou výsledky obou implementací.

FPGA	**F [MHz]**	**A [Slice]**	**P [Gbps]**	**LUT**	**IOBs**
S-3	100	1702	2,4	4,499	552
V6	150	1649	7,2	9,895	585

Tab. 3.2: Výsledky implementace [8]

3.2 Simulace existující implementace

V rámci seznámí se s existujícími implementacemi byl otestován návrh pro čipy pracující na vysoké frekvenci z [8]. Simulace byla provedena v simulačním programu ModelSim.



Obr. 3.1: Simulace implementace

zpráva (hexa)	’54686520717569636b2062726f776e20666f78206a7 56d7073206f76657220746865206c617a7920646f67’
haš (hexa)	’01dedd5de4ef14642445ba5f5b97c15e47b9ad931326e 4b0727cd94cefc44fff23f07bf543139939b49128caf436 dc1bdee54fcb24023a08d9403f9b4bf0d450’

Tab. 3.3: Testovací zpráva simulace

Na obrázku 3.1 je simulace zobrazena. Můžeme si všimnout, že po naplnění zásobníku (signalizováno signálem *BUFFER_FULL*) se s každým hodinovým taktem

mění signál *OUT* což představuje permutace stavového pole. Signál *OUT* reprezentuje výsledný haš. Po 12 taktech a tedy 24 permutacích je *OUT* signalizován jako validní a reprezentuje tedy náš výsledný haš. V tabulce 3.4 jsou signály blíže popsány.

Název signálu	**Šířka (v bitech)**	**Popis**
CLK	1	hodinový signál
RESET	1	globální reset
IN	64	vstupní data
BYTE_NUM	3	pozice bajtu, kde končí daná zpráva
IN_READY	1	signalizace zda je komponenta připravena přijmout data
IS_LAST	1	signalizace zda se jedná o poslední blok zprávy
BUFFER_FULL	1	signalizace zda je zásobník zaplněná
OUT	512	výsledný haš
OUT_VALID	1	platnost haše

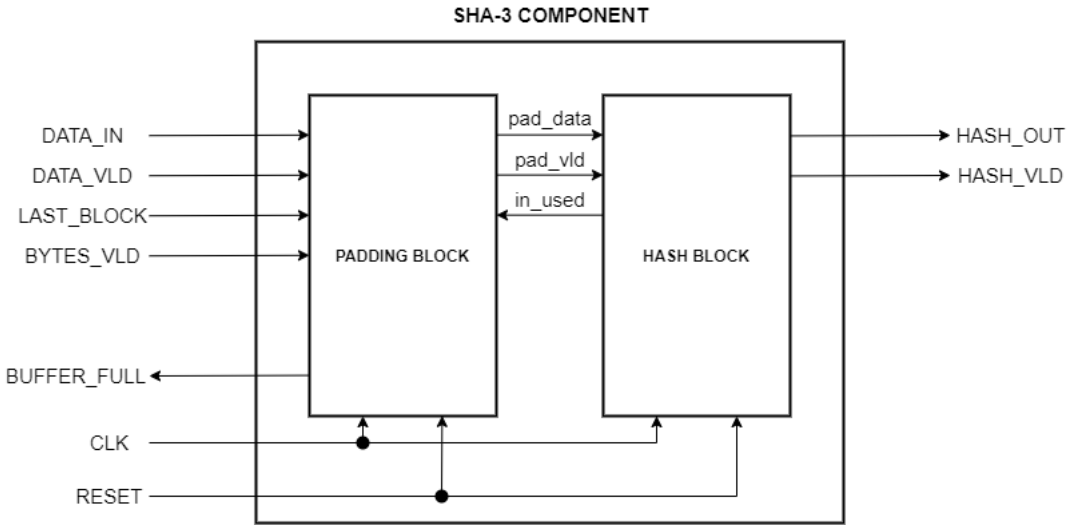
Tab. 3.4: Popis signálů simulace

3.3 Návrh komponent

Jak bylo zmíněno v kapitole 1 tak přepočet jednoho stavového pole zabere 24 iterací. Návrh počítá s provedením dvou permutací během jednoho taktu. Bylo by možné spočítat i více permutací, ale výsledek by byl velmi náročný na zdroje v FPGA. Celkový propočet stavového pole by měl tedy zabrat 12 hodinových taktů. Protože se komponenta bude připojovat na 512-bitovou sběrnici z počátku bylo tedy uvažováno se stejně širokým vstupem do komponenty. Návrh se ale nakonec ukázal jako velmi problémový a to z důvodů, že by bylo potřeba vytvořit zásobník, který by byl schopen pojmout nejméně dvě slova sběrnice společně s informacemi o počátku a konci paketu. Následná manipulace s daty v zásobníku by byla obtížná jelikož by mohlo dojít ke stavu, že ne vždy by byl zásobník zaplněn a nebo bychom nemohli přijmout celé slovo ze sběrnice. Z tohoto důvodu byla šířka vstupních dat změněna na 64-bitů.

Hodinový signál je nutný pro správné řizení komponent. Nezbytný reset slouží k vynulování vnitřních registrů komponent.

3.3.1 SHA-3 komponenta



Obr. 3.2: Struktura SHA-3 komponenty

Na obrázku 3.2 můžeme vidět celkový návrh SHA-3 komponenty. Hlavními prvky jsou 2 podkomponenty a to **padding block**, který realizuje zarovnávání vstupní zprávy a **hash block**, jenž slouží k výpočtu haše.

Vstupní data budou přicházet přímo do padding blocku, který bude sloužit zároveň jako zásobník. Pokud přijdou informace o tom, že se jedná o poslední blok zprávy, data budou zarovnána s ohledem na počet platných bajtů. Jakmile se zásobník zaplní, je předán signál o plném zásobníku, data se přestanou přijmat a zarovnaná data, reprezentující datový blok **r** se předají hash blocku společně s informací o jejich platnosti.

Hashovací blok označí data jako přijatá a provede potřebný počet permutací. Zda bude výstupní hash označen za platný se bude starat logika samotné SHA-3 komponenty. Pokud ano, bude nutné před začátkem přijmání další zprávy provést reset komponent a pokud ne, data zůstanou uložené v hash bloku, aby mohlo následně dojít k logické funkci xor s následujícím blokem zprávy.

Název signálu	Šírka (v bitech)	Popis
DATA_IN	64	vstupní data
DATA_VLD	1	platnost dat na vstupu
LAST_BLOCK	1	signalizuje, zda se jedná o poslední data ve zprávě
BYTES_VLD	4	počet platných bajtů ve vstupních datech
BUFFER_FULL	1	signalizace zda je zásobník zaplněná
pad_data	576	zarovnané data
pad_vld	1	zarovnání platné
data_used	1	signalizuje, zda data byla zpracována
HASH_OUT	512	výsledný haš
HASH_VLD	1	platnost haše
CLK	1	hodinový signál
RESET	1	globální reset

Tab. 3.5: Popis signálů SHA-3 komponenty

3.4 Požadavky na zdroje FPGA

V již existující implementaci [8], která je popsána výše, se v návrhu pro vysokofrekvenční čipy kalkulují dvě permutace v jednom taktu. Tenhle návrh spotřebovává 9 895 LUT bloků. V aktuálním návrhu se počítá s provedením právě dvou permutací v jednom taktu a očekává se tedy velmi podobná spotřeba zdrojů. Nejvíce zdrojů zabírají výpočetně náročné operace nad stavovým polem. Konkrétně transformace *theta* a počáteční logická funkce xor dvou stavových polí v hašovací komponentě.

3.5 Cílová platforma

Cílovou platformou je čip od firmy Xilinx[1] a to Virtex 7 Ultrascale+ s označením xcvu7p-flvb2104-2-i. Tabulka 3.6 obsahuje dostupné zdroje čipu. V tabulce **LB** - počet logických bloků, **Flip-Flop** - počet klopných obvodů, **LUT** - počet LUT bloků, **RAM** - velikost paměti ram, **UltraRAM**[2] - velikost paměti ultraram.

[1] www.xilinx.com

[2] Speciální rozšiřující paměť typu RAM

LB	Flip-Flop	LUT	RAM [MB]	UltraRAM [MB]
1 724 000	1 576 000	788 000	50,6	180

Tab. 3.6: Dostupné zdroje čipu Virtex 7 Ultrascale+

FPGA je umístěno na vysokorychlostní kartě NFB-200G2Q. Na obrázku 3.3 si můžeme kartu prohlédnout.

Obr. 3.3: Karta NFB-200G2QL firmy Netcope Technologies, a.s.

4 Implementace

V kapitole o implementaci si popíšeme vzniklou implementaci v jazyce Python a bude detailně objasněn princip funkcionality jednotlivých komponent. Pro lepší představu a pochopení bude vše vysvětleno na simulacích. Popíšeme architekturu simulace a simulaci komponenty společně se sběrnicí. Dále budou představeny získané výsledky.

4.1 Python implementace

Hlavním účelem implementace v jazyce Python bylo úplné seznámení se s hašovacím algoritmem. Implementace probíhala pro verzi Pythonu 3.8. Ačkoliv Python patří k interpretovaným jazykům, tak byl zvolen z důvodů dobré čitelnosti a jeho vysoké úrovně. Byly implementovány všechny SHA-3 funkce, které byly následně otestovány. Všechny funkce přijmají vstupní data v ASCII či hexadecimální podobě. Ve výpisu 4.1 je příklad funkce *theta* v jazyce Python.

Výpis 4.1: Funcke *theta* v jazyce Python

```
def theta(A):
    c_t=[[] for x in range(5)]
    for x in range(5):
        c_t[x]=A[x][0]^A[x][1]^A[x][2]^A[x][3]^A[x][4]
    d_t=[[] for x in range(5)]
    for x in range(5):
        d_t[x]=c_t[(x-1)%5]^rot(c_t[(x+1)%5],-1)
    for y in range(5):
        for x in range(5):
            A[x][y]=A[x][y]^d_t[x]
    return A
```

Testování proběhlo dvěma způsoby. V prvním případě byly použity hexadecimální zprávy doporučené společností NIST[1] získané z [9]. Zpráva společně s výsledným hašem byly vyčteny z textového souboru, na zprávu byla aplikovaná hašovací funkce a výsledky byly následně porovnány. V druhém případě byly generovány náhodné zprávy čísel a písmen. Bylo vygenerováno 10 000 náhodných vektorů pro každou funkci. Po aplikování hašovací funkce byl výsledek porovnán s výsledkem již existující funkční Python implementace, která byla získána z [10].

V tabulce 4.1 se můžeme podívat na příklad testovací zprávy.

[1] Národní institut standardů a technologie. Vydává i standardy ohledně šifrování.

zpráva (hexa)	’0d512eceb74d8a047531c1f716’
haš (hexa)	’29ae0744051e55167176317eb17850a22939d8d94ebb0a90b6d98fde’

Tab. 4.1: Příklad testovacího řetězce pro SHA3-224

4.1.1 Implementované funkce

- **sha3_224** (msg), realizující funkci SHA3-224.
  Vstupem je zpráva msg v hexadecimální nebo ASCII podobě.
  Výstupem je výsledný haš.
- **sha3_256** (msg), realizující funkci SHA3-256.
  Vstupem je zpráva msg v hexadecimální nebo ASCII podobě.
  Výstupem je výsledný haš.
- **sha3_384** (msg), realizující funkci SHA3-384.
  Vstupem je zpráva msg v hexadecimální nebo ASCII podobě.
  Výstupem je výsledný haš.
- **sha3_512** (msg), realizující funkci SHA3-512.
  Vstupem je zpráva msg v hexadecimální nebo ASCII podobě.
  Výstupem je výsledný haš.
- **theta** (A), funkce odpovídá permutaci θ.
  Vstupem je stavové pole A.
  Výstupem je stavové pole A’.
- **rho_pi** (A), funkce odpovídá permutaci ρ a π.
  Vstupem je stavové pole A.
  Výstupem je stavové pole B.
- **chi** (A, B), funkce odpovídá permutaci χ. Zmíněná funkce má na rozdíl od funkce popsané v kapitole 1 dva vstupy a to z důvodů sjednocení funkceρ a π. Výsledek však zůstává stejný.
  Vstupem je stavové pole A a B.
  Výstupem je stavové pole A’.
- **iota** (A, ir), funkce odpovídá permutaci ι.
  Vstupem je stavové pole A a hodnota iterace ir.
  Výstupem je stavové pole A’.
- **rot** (lane, i), rotuje bity ve směru osy X.
  Vstupem je řada stavového pole A a a hodnota rotace i.
  Výstupem je rotovaná řada.
- **to_str** (A), převede stavové pole na bitový řetězec.
  Vstupem je stavové pole A.

Výstupem je bitový řetězec.

- **padding** (r, l), funkce vrací zarovnávací bity.
  Vstupem je délka datového bloku r a délka bitové zprávy l.
  Výstupem jsou zarovnávací bity.
- **tobitvector_and_pad** (msg, r), funkce převede zprávu do bitové podoby a provede zarovnání.
  Vstupem je zpráva msg a velikost datového bloku r.
  Výstupem je zarovnaná zpráva v bitové podobě.
- **to3dmat** (msg_bits), převede bitový řetězec na stavové pole A.
  Vstupem je bitový řetězec msg_bits.
  Výstupem je stavové pole A.
- **str2hex** (msg_bits), převede výsledný haš na hexadecimální podobu.
  Vstupem je bitový řetězec msg_bits.
  Výstupem je haš v hexadecimálním podobě.
- **keccak_m** (A), realizující funkci KECCAK.Převod na stavové pole a následně na bitový řetězec je realizováno ve funkci **sponge_m**.
  Vstupem je stavové pole A.
  Výstupem je permutované stavové pole.
- **sponge_m** (msg, d), realizující funkci SPONGE. Součástí funkce je i převod na stavové pole a následně na bitový řetězec.
  Vstupem je zpráva msg a délka výstupu d.
  Výstupem je haš v binární podobě.

4.1.2 Použité knihovny

Před vlastnoručním otestováním implementace je nutné mít nainstalované zmíněné knihovny :

- **bitarray** umožňuje operace s bitovými řetězci.
- **numpy** rozšiřuje matematické operace.
- **string** umožňuje práci s datovým formátem string.
- **bitstring** umožňuje operace s bitovými řetězc.
- **unittest** umožňuje testování funkcí.
- **pycryptodome** součástí knihovny jsou již implementavané SHA3 referenční funkce.

4.2 Implementace v jazyce VHDL

Hašovací algoritmus byl implementován v jazyce VHDL. Kvůli požadavku na pracovní frekvenci 200 MHz jsme museli snížit počet permutací na jednu. Počet permutací byl snížen z toho důvodu, že při požadované pracovní frekvenci při implementaci nebylo dodržené časování. WNS byl v tomhle případě -1,65 ns.

Princip funčknosti si nejlépe vysvětlíme na obrázcích ze simulace. Jednotlivé takty jsou na obrázcích očislovány a průběh každého taktu je detailně popsán. Simulace byly prováděny v simulačním programu ModelSim - Intel FPGA edition 10.6d.

4.2.1 SHA-3 komponenta

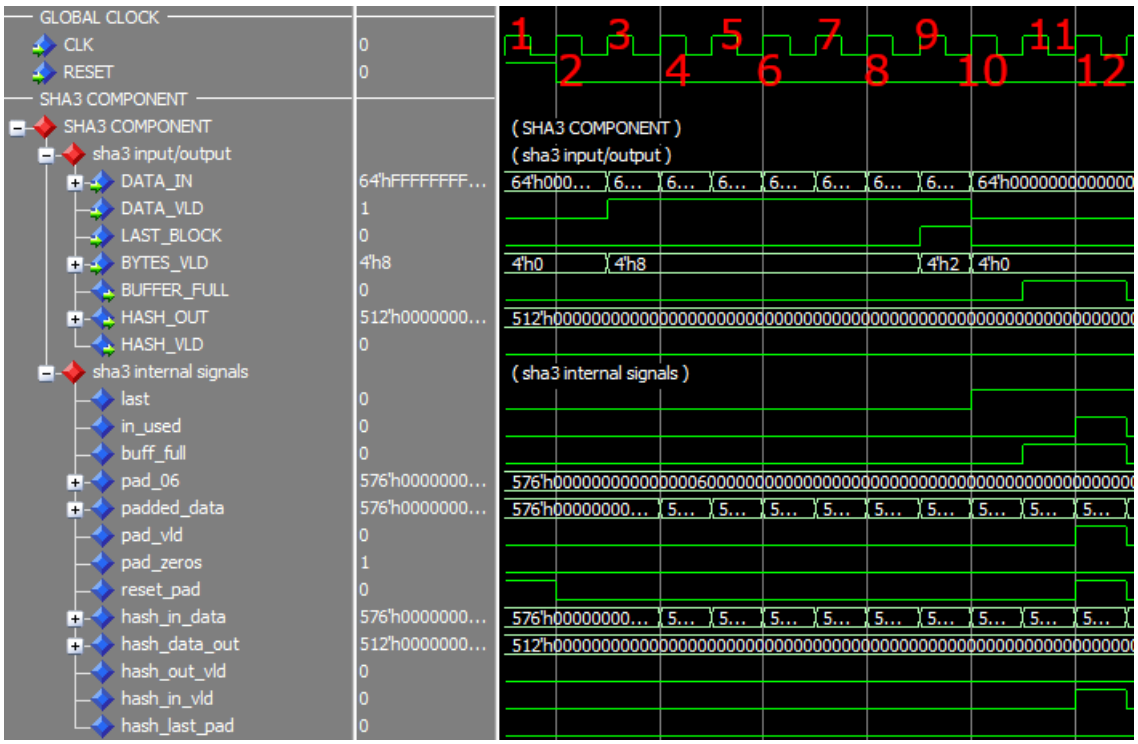
Komponenta SHA-3 (sha3.vhd v příloze) propojuje a řídí veškeré podkomponenty. Oproti návrhu nedošlo k žádným změnám.

Pro lepší zobrazení si simulaci rozdělíme do dvou částí. V první části bude přijmána zpráva a v části druhé se bude počítat výsledný haš. Jako první si popíšeme část, kde se přijmá vstupní zpráva.

Na obrázku 4.1 můžeme vidět, že v prvním hodinovém taktu je aktivní signál *RESET* a probíhá resetování komponenty. S druhou náběžnou hranou se *RESET* stává neaktivním a komponenta je připravena přijmout data. Při třetí náběžné hraně jsou na vstup *DATA_IN* přivedena data, která jsou vstupním signálem *DATA_VLD* označena jako platná. V následujících taktech jsou přijmáná vstupní data. Můžeme si také všimnout toho, že po celou dobu přijmání dat má vstupní signál *BYTES_VLD* hodnotu 0x8. Signalizujeme tím, že všechny bajty v přijatém bloku jsou platné.

S devátou náběžnou hranou přijmáme blok zprávy, který je signálem *LAST_BLOCK* označen jako posledním blokem dané zprávy. Zároveň měníme hodnotu signálu *BYTES_VLD* na 0x2, což znamená, že v přijatém bloku jsou platné pouze dva nejvyšší bajty. Poslední přijmutá data 0x0431**30108abf631e** jsou tedy zarovnána na 0x0431**060000000080**, což odpovídá předpisu funkce SHA-3. Během desáté nástupní hrany se signál *last* stává aktivním. Signálem řídíme výstup komponenty. Značí nám, že poslední blok zprávy již byl přijat. S jedenáctou náběžnou hranou je zásobník zaplněn a výstup *BUFFER_FULL* je v logické jedničce. Aktivním se stává i signál *buff_full*, který je pomocným signálem pro výstup *BUFFER_FULL*. Během dvanáctého taktu jsou vnitřním signálem *pad_vld* zarovnaná data *padded_data* označena za platná. Signál *padded_data* je vstupním signálem do hašovací komponenty. Jelikož jsou zarovnaná data platná a hašovací komponenta neprovádí žádný výpočet, jsou data hašovací komponentou přijata a započíná výpočet. Přijmutí dat signalizujeme pomocí signálu *in_used*. Protože byl výstup zarovnávací

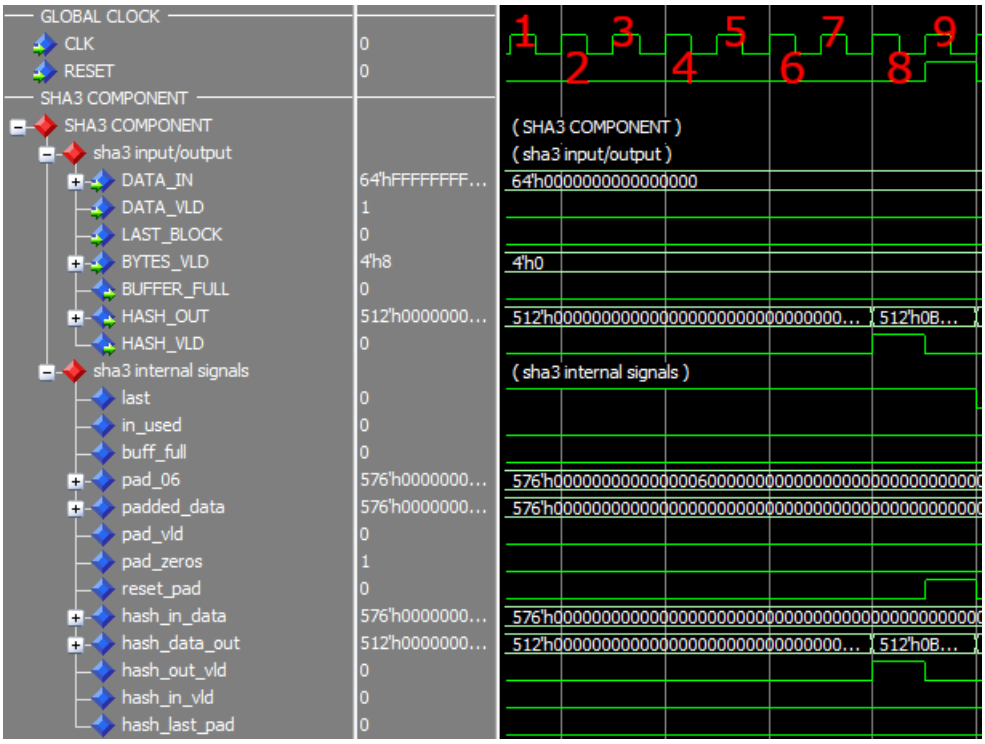
komponenty zpracován, dochází k jejímu resetování pomocí signálu *reset_pad*. Reset slouží k vynulování čítače v zarovnávací komponentě. Abychom docílili co největší propustnosti, přijmáme data kdykoliv je zásobník vyprázděn až do chvíle než je přijmutý poslední blok zprávy.



Obr. 4.1: Simulace komponenty sha3 - přijmání zprávy

Stav signálů než dojde k vypočítaní haše můžeme pozorovat na obrázku 4.2. Jelikož výpočet trvá 25 taktů, tak si ukážeme pouze posledních pár taktů před vypočítáním výsledného haše.

Od první náběžného hrany čekáme na výsledek. S osmou náběžnou hranou hašovací komponenta přiřadí výsledek na svůj výstup pomocí signálů *hash_data_out*. Signál slouží pouze jako pomocný signál pro výstup *HASH_OUT*. Jelikož není nutno provést dodatečné zarovnání a signál *last* je aktivní, můžeme výsledek výstupem *HASH_VLD* označit za platný. S devátou náběžnou hranou dochází k resetu všech komponent. Reset je nutný, protože musíme před začátkem přijmání nové zprávy vynulovat vnitřní registry komponent.



Obr. 4.2: Simulace komponenty sha3 - výpočet haše

Signály *pad_zeros*, *pad_06* a *last_pad_hash* sice nejsou využity v obrázcích ze simulace, ale popíšeme si alespoň jejich účel. Pokud přijmeme poslední blok zprávy takový, že zaplní zásobník a má všechny bajty platné je nutné zprávu dodatečně zarovnat. Jesltiže tedy na vstup přijdou platná data, která májí všechny bajty platné, jedná se o poslední blok zprávy a zásobník je zaplněn, stává se aktivním signál *pad_zeros*. Tenhle signál nám říká, že jakmile hašovací komponenta dokončí výpočet, tak výsledek není označen výstupem jako platný. Po prvním výpočtu hašovací komponenty se dalším vstupem stává signál *pad_06* ve kterém je uloženo pevně dané zarovnání. Signál *last_pad_hash* slouží jako pomocný signál pro označení vstupních dat do hašovací komponenty jako platných. Z toho plyne, že až druhý výsledek hašovací komponenty můžeme označit za platný. Signál *pad_zeros* se stává neaktivním jakmile je první výpočet hašovací komponenty označet za platný.

V tabulce 4.2 jsou popsány všechny signály komponenty.

Název signálu	**Šířka (v bitech)**	**Typ**	**Popis**
CLK	1	vstup	hodinový signál
RESET	1	vstup	reset
DATA_IN	64	vstup	vstupní data
DATA_VLD	1	vstup	platnost vstupních dat
BYTES_VLD	4	vstup	počet platných bajtů ve vstupních datech
LAST_BLOCK	1	vstup	poslední blok zprávy
BUFFER_FULL	1	výstup	zásobník zaplněn
HASH_OUT	512	výstup	výstupní haš
HASH_VLD	1	výstup	platnost haše
last	1	vnitřní	řízení výstupního signálu *HASH_VLD*
in_used	1	vnitřní	výstup zarovnávací komponenty byl zpracován
buff_full	1	vnitřní	pomocný signál pro výstup *BUFFER_FULL*
pad_06	576	vnitřní	pomocný signál pro zarovnání
padded_data	576	vnitřní	zarovnaná data
pad_vld	1	vnitřní	platnost zarovnání
pad_zeros	1	vnitřní	signalizace, že je to potřeba dodatečně zarovnat vstupní zprávu
reset_pad	1	vnitřní	pomocný signál pro resetování zarovnávací komponenty
hash_in_data	576	vnitřní	vstup do hašovací komponenty
hash_data_out	512	vnitřní	výstup hašovací komponenty
hash_out_vld	1	vnitřní	platnost výstupu hašovací komponenty
hash_in_vld	1	vnitřní	platnost vstupu hašovací komponenty
hash_last_pad	1	vnitřní	start dodatečného zarovnání

Tab. 4.2: Popis signálů komponenty sha3

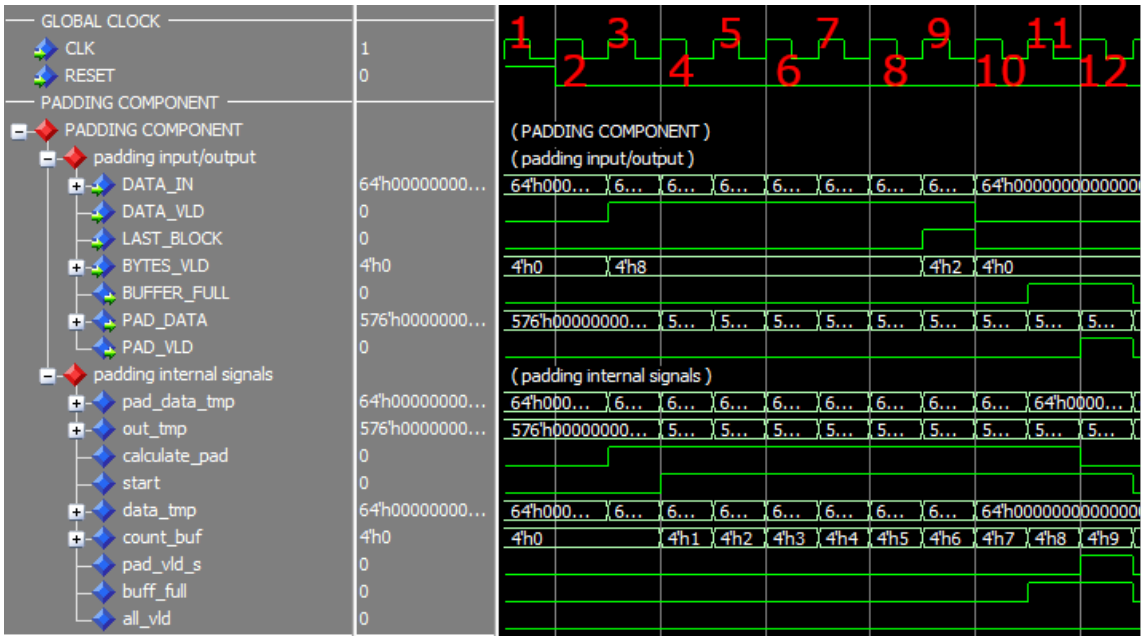
V tabulce 4.3 je zpráva, která je na obrázcích ze simulace společně s výsledným hašem.

zpráva (hexa)	302FA84FDAA82081B1192B847B81DDEA10A9F05A0F0413 8FD1DA84A39BA5E18E18BC3CEA062E6DF92FF1ACE89B 3C5F55043130108ABF631E
haš (hexa)	0B25087159F17655AB1CEFE4A20BB0B24BA7AB064867C 38980DA029FF34261987DA0B2EDDEF45800C2AA389134 F6455A0B1181E897267546C0ACDDADE04BF728

Tab. 4.3: Zpráva ze simulace

4.2.2 Zarovnávací komponenta

Komponenta (v příloze padder.vhd) slouží k zarovnání vstupní zprávy dle algoritmu 9 popsaného v podkapitole 1.1.3. Pro lepší popsání funkcionality si opět vypomůžeme obrázkem 4.3 ze simulace.



Obr. 4.3: Simulace zarovnávací komponenty

Podobně jak na obrázku 4.1, tak v prvním taktu dochází k resetování komponenty. Od druhé náběžné hrany čekáme na vstupní data. Vstupní data přivádíme na vstup *DATA_IN* se třetí náběžnou hranou. Zároveň dostáváme informaci o platnosti vstupních dat vstupem *DATA_VLD* a platný počet bajtů pomocí vstupu *BYTES_VLD*. Do logické jedničky se dostává i signál *calculate_pad*, který slouží jako indikace pro komponentu, že zpráva ještě není zarovnána a má tedy pokračovat v zarovnávání. Signál je aktivním pokud jsou data na vstupu platná nebo je aktivní

signál start a zarovnání není platné. Můžeme si také všimnou měnících se signálů *data_tmp* a *pad_data_tmp*. Signál *data_tmp* slouží jako pomocný signál pro vstup a do *pad_data_tmp* přiřazujeme zarovnaná data. Zarovnávání probíhá na základě hodnoty vstupu *BYTES_VLD*, kdy neplatný počet bajtů je nahrazen ukončovacím znakem zprávy a následně nulami. Se čtvrtou náběžnou hranou se mění hodnota čtyř bitového čítače reprezentovaného signálem *count_buf*. Pomocí čítače čítáme počet přijatých a zarovnaných bloků. Řídíme jím signály *buff_full* a *pad_vld_s*. Signál *buff_full* je pomocným signálem pro výstup *BUFFER_FULL* a signál *pad_vld_s* je pomocným signálem pro výstup *PAD_VLD*.

S devátánou náběžnou hranou dostáváme informaci, že se jedná o poslední blok dané zprávy a mění se nám taktéž počet platných bajtů. Zbytek zprávy je zarovnán dle předpisu funkce SHA-3.

Při jedenácté náběžné hraně se stává aktivním výstup *BUFFER_FULL* na základně hodnoty counteru a jeho pomocného signálu *buff_full*. S poslední dvanáctou náběžnou hranou je zarovnání označeno za platné pomocí výstupu *PAD_VLD*.

V tabulce 4.4 si můžeme prohlédnout závislost platných bajtů ve zprávě na hodnotě signálu *BYTES_VLD*.

Hodnota signálu (v bitech)	**Počet platných bajtů**
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8

Tab. 4.4: Tabulka počtu platných bajtů v závilosti na hodnotě signálu *BYTES_VLD*

Nepopsaný signál *all_vld* je pomocným signálem pro zarovnání v případě, že poslední blok zprávy měl všechny bajty platné. V rámci zarovnávací komponenty provádíme i přeskládání bajtů. Přeskládání je nutné kvůli endianitě. Ve výpisu 4.2 si můžeme prohlédnout princip přeskládání bajtů.

Výpis 4.2: Princip přeskládání bajtů

```
pad_reorder_g1 : for w in 0 to 8 generate
    par_reorder_g2 : for b in 0 to 7 generate
        PAD_DATA((w*64+b*8)+7 downto (w*64+b*8)) <=
                out_tmp((w*64+(7-b)*8)+7 downto (w*64+(7-b)*8));
    end generate;
end generate;
```

V tabulce 4.5 jsou popsány všechny signály zarovnávací komponenty.

Název signálu	**Šířka (v bitech)**	**Typ**	**Popis**
CLK	1	vstup	hodinový signál
RESET	1	vstup	reset
DATA_IN	64	vstup	vstupní data
DATA_VLD	1	vstup	platnost vstupních dat
LAST_BLOCK	1	vstup	poslední blok zprávy
BYTES_VLD	4	vstup	počet platných bajtů ve vstupních datech
PAD_DATA	576	výstup	zarovnaná data
PAD_VLD	1	výstup	platnost zarovnaných dat
pad_data_tmp	64	vnitřní	zarovnaná vstupní data
out_tmp	576	vnitřní	pomocný signál pro výstup *PAD_DATA*
calculate_pad	1	vnitřní	signalizace, že data se mají zarovnávat
start	1	vnitřní	pomocný signál pro calculate_pad
data_tmp	64	vnitřní	pomocný signál pro vstup *DATA_IN*
count_buf	4	vnitřní	hodnota čítače
pad_vld_s	1	vnitřní	pomocný signál pro výstup *PAD_VLD*
buff_full	1	vnitřní	pomocný signál pro výstup *BUFFER_FULL*
all_vld	1	vnitřní	pomocný signál pro zarovnání

Tab. 4.5: Popis signálů zarovnávací komponenty

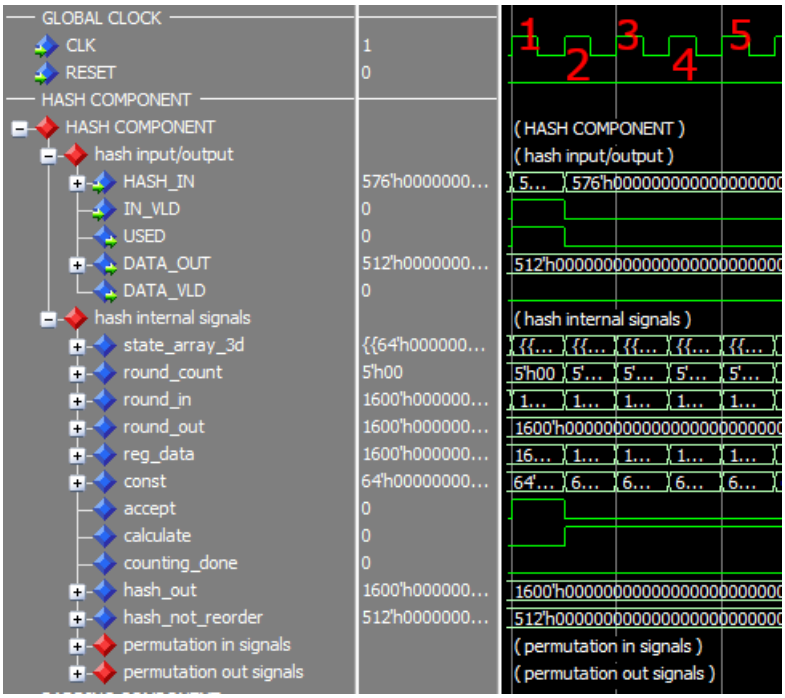
.

4.2.3 Haš komponenta

Výpočet haše je realizován v hašovací komponentně (hash.vhd). Je počítána jedna permutace během taktu. Při navyšování počtu permutací v jednom taktu neprocházelo časování při implementaci a proto byl snížet počet permutací na jednu. Hašovací komponenta také spotřebovává nejvíce zdrojů v FPGA z celkové implementace. Pro lepší zobrazení je simulace rozdělena do dvou části. V první části započíná výpočet a ve druhé částí dochází k vypočítání výsledku.

Na obrázku 4.4 si opět můžeme prohlédnout simulaci komponenty. S první náběžnou hranou přivádíme na vstup *HASH_VLD* platná vstupní data. Vnitřní signál *accept*, který nabývá hodnotu logické jedničky značí, že vstup byl přijat a započíná tak jeho zpracovávání. Na základě tohoto signálu se stává aktivním výstupní signál *USED*. Je provedena logická funkce xor vstupních dat a vnitřního registru *reg_data*, který uchovává výsledek předešlých výpočtů. Pokud začínáme počítat novou zprávu, registr je vynulován. Výsledek je následně převeden na pomyslné 3D stavové pole, které reprezentujeme signálem *state_array_3d*. Signál *const* vyčítá konstantu pro jednotlivé kolo permutace na základě hodnoty pěti bitového čítače. Hodnota čítače je uložena v signálu *round_counter*. Signály *round_in* a *round_out* jsou pomocnými signály pro vstup a výstup z permutace.

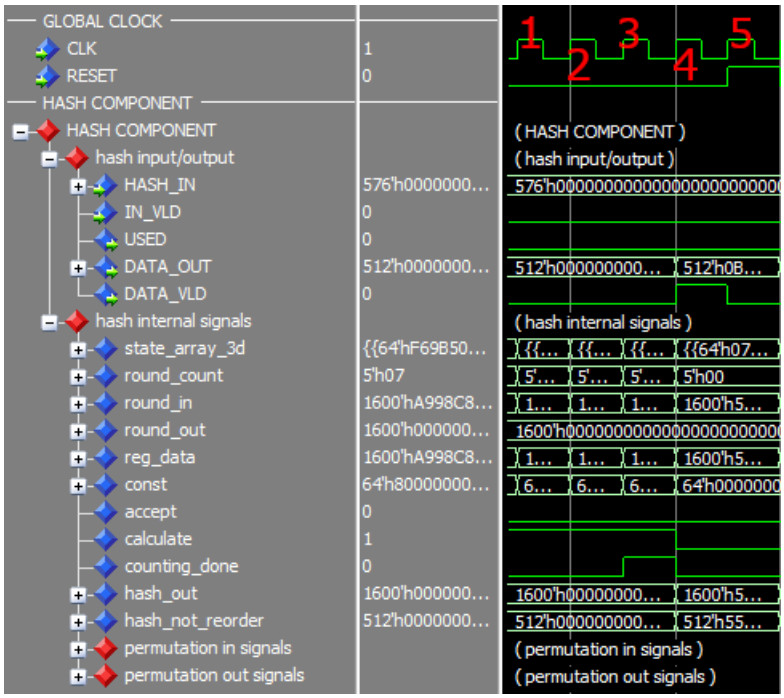
S druhou náběžnou hranou se stává aktivním signál *calculate*, kterým signalizujeme, že má probíhat výpočet.



Obr. 4.4: Simulace hašovací komponenty - začátek výpočtu

Na obrázku 4.5 si můžeme prohlédnout vypočítání haše. Se třetí náběžnou hranou se stává aktivním signál *counting_done*. Signál nám říká, že všechny permutace byly spočítány a můžeme vyresetovat čítač. Se čtvrtou náběžnou hranou přiřadíme výsledný haš na výstup a označíme ho jako platný. Před přiřazením jsou opět přeskládány bajty. Jelikož byla celá zpráva spočítána, tak v pátém taktu probíhá reset komponenty.

Skupiny signálů *permutation in signals* a *permutation out signals* nejsou popsány jelikož obsahují pouze pomocné vstupní a výstupní signály do jednotlivých funkcí permutace. Ssignály slouží převážně k detekcí chyb mezi jednotlivými propočty. Ke kontrole správnosti mezivýpočtů byla pouzitá online platoforma dostupná na [11].



Obr. 4.5: Simulace hašovací komponenty - výsledek

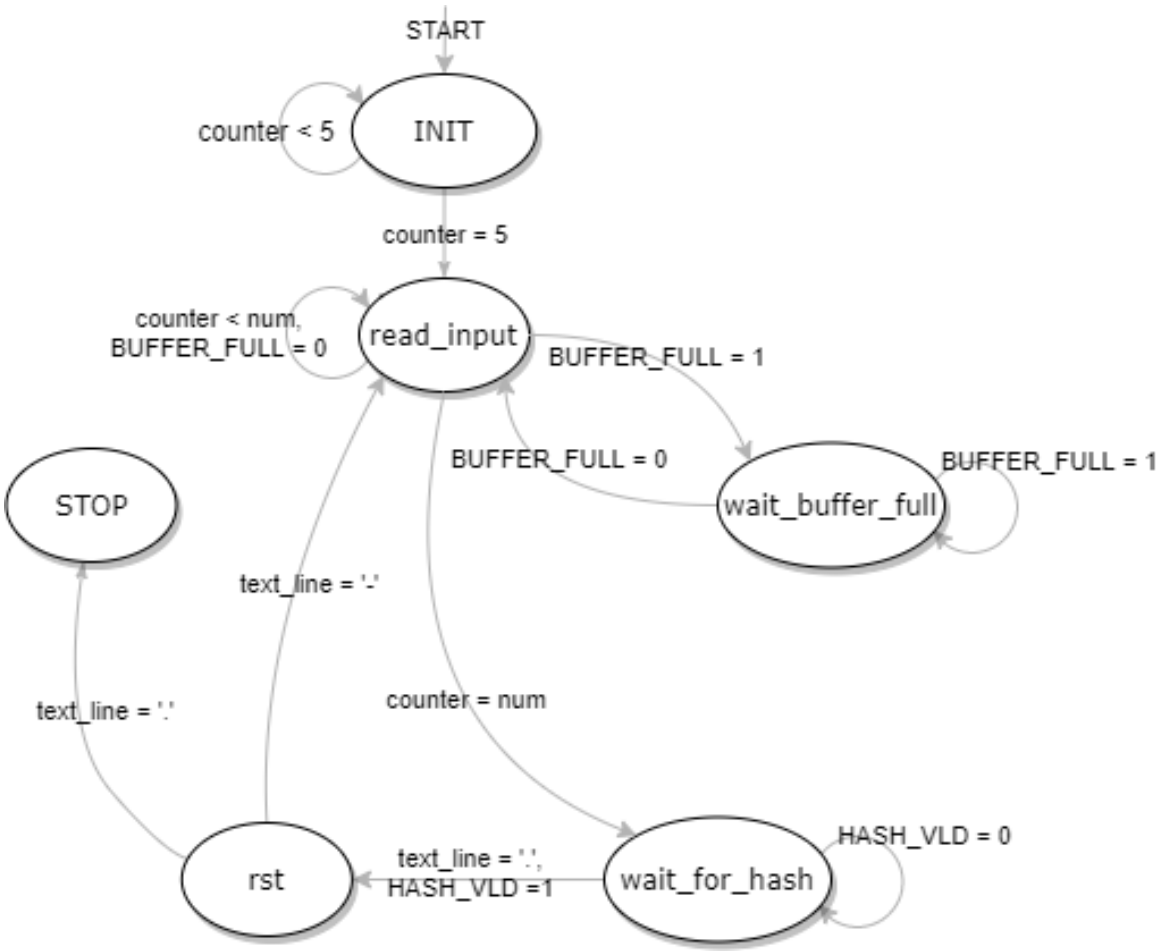
V tabulce 4.6 jsou popsány signály hašovací komponenty.

Název signálu	Šířka (v bitech)	Typ	Popis
CLK	1	vstup	hodinový signál
RESET	1	vstup	reset
HASH_IN	576	vstup	vstupní data
IN_VLD	1	vstup	platnost vstupu
USED	1	výstup	využití vstupu
DATA_OUT	512	výstup	výsledný haš
DATA_VLD	1	výstup	platnost haše
state_array_3d	1600	vnitřní	stavové pole
round_count	5	vnitřní	čítač permutací
round_in	1600	vnitřní	vstup do permutace
round_out	1600	vnitřní	výstup permutace
reg_data	1600	vnitřní	vnitřní registr výsledku permutace
const	64	vnitřní	konstanta kola
accept	1	vnitřní	přijmutí vstupu
calculate	1	vnitřní	výpočet hašování
counting_done	1	vnitřní	výpočet dokončen
hash_out	1600	vnitřní	pomocný signál s výsledkem
hahs_not_reorder	512	vnitřní	nepřeskládaný výsledek

Tab. 4.6: Popis signálů hašovací komponenty

4.2.4 Architektura simulace

Komponenta je součástí reálné aplikace a je tedy připojena k dalším externím komponentám. Bylo nutné vytvořit simulaci, která chováním odpovídá externí komponentě ke které jsme připojeni. Simulace stejně tak jako externí komponenta je napsána formou stavového automatu jehož diagram si můžeme prohlédnout na obrázku 4.6.



Obr. 4.6: Stavový diagram simulace

Počátečním stavem je stav *INIT* ve kterém probíhá reset všech komponenet. Pro pomocné čítaní taktů využíváme proměnnou *counter*. Po napočítaní pěti taktů přecházíme do stavu *read_input*. V tomhle stavu načteme informace o zprávě a to konkrétně počet bloků, které budeme vysílat a počet platných bajtů v posledním bloku zprávy. Následně v každém taktu načítáme data samotné zprávy, které pak přiřazujeme na vstup. Pokud je zásobník zaplněn přecházíme do stavu *wait_buffer_full* a čekáme dokud není vyprázdněn, abychom mohli vysílat další data. Jakmile odvysíláme všechny bloky zprávy, přecházíme do stavu *wait_for_hash*. K počítání odvysílaných bloků opět používáme pomocnou proměnnou *counter*. Jakmile napočítáme počet zpráv, které mají být odvysílaný, označíme poslední vyslaný blok jako posledním blokem zprávy a pošleme informaci o počtu platných bajtů v posledním bloku. Počet bloků dané zprávy, kterou budeme vysílat je uložen v pomocné promněnné *num*. Ve stavu *wait_for_hash* čekáme dokud haš není označen za platný.

Jakmile je haš označen za platný, načteme správný výsledek ze souboru s výsledky a porovnáme ho s výstupním signálem *HASH_OUT*. Jestli jsou výsledky

stejné je do konzole vypsána hláška "Hash is valid!", pokud je výsledek nesprávný, je výpsáno "Hash is not valid!". Stav výstupu *HASH_OUT* je uložen do souboru. Následně přejdeme do stavu *rst*. Ve stavu *rst* dojde k resetování komponenty. Poté jsou přečtena další data. Pokud přečteme znak '**-**' značíme tím, že soubor obsahuje další zprávy ke zpracování a následuje tedy stav *read_input*. Jestliže je načten znak '**.**', už žádné další zprávy ke zpracování nejsou a následuje stav *STOP* ve kterém simulace končí. Konec simulace je ohlášen zprávou "Simulation completed".

Data o zprávě a následnou zprávu načítáme ze souboru *in.txt*, výsledky jsou načítány ze souboru *out.txt* a výsledky hašů získané simulací jsou ukládány do souboru *results.txt*. Soubory jsou v příloze v adresáři *./SHA3-VHDL/test_msgs*.

Aby při načítání dat ze souboru *in.txt* nedošlo k chybě, musíme dodržovat danou formu zápisu informací. Příklad takového zápisu je uveden ve výpisu 4.3. První řádek je počet bloků zprávy, které budeme vysílat. Dále počet platných bajtů v posledním bloku za kterým následuje celá zpráva. Na posledním řádku je ukončovací znak.

Výpis 4.3: Příklad zápisu dat pro zpracování

```
6
0011
5468652071756963
6b2062726f776e20
666f78206a756d70
73206f7665722074
6865206c617a7920
646f672020202020
-
```

4.3 Implementace do FPGA čipu

V následující podkapitole si popíšeme simulaci hašovací komponenty v reálné aplikaci a představíme výsledky, které jsme získali.

4.3.1 Simulace s reálnou aplikací

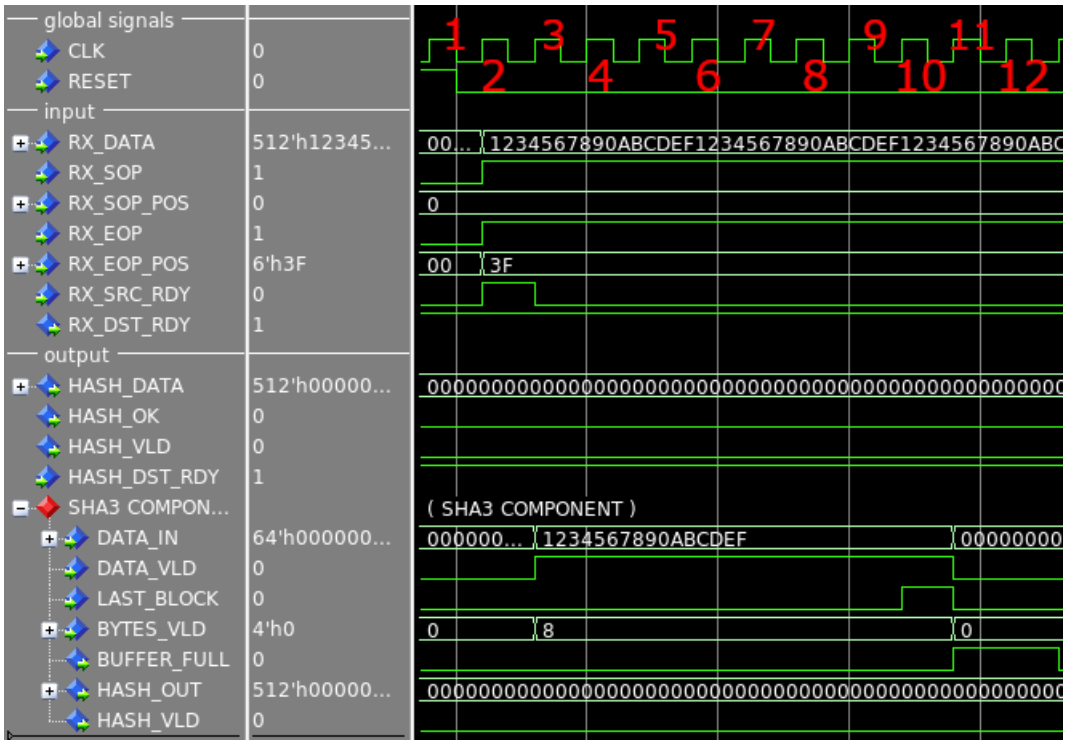
Komponenta se připojuje k 512-bitové sběrnice, která slouží k přenosu paketů po kartě. V našem případě jsou využívány externí komponenty. Na obrázcích 4.7 a 4.8 si můžeme prohlédnout společnou simualci.

Se sestupnou hranou prvního taktu na obrázku 4.7 si signál *RESET* stává neaktivním. S náběžnou hranou druhého taktu se stává aktivním vstup *RX_SRC_RDY*,

který nám říká, že data na vstupu *RX_DATA* jsou platná a můžeme je začít zpracovávat. Aktivní signál *RX_SOP* nám říká, že přijatá data obsahují nový paket, který začíná na pozici, kterou vyčteme ze vstupu *RX_SOP_POS*. Naopak signál *RX_EOP* signalizuje, že v přijaté zprávě paket taktéž končí. K přenosu pozice posledního bajtu paketu slouží signál *RX_EOP_POS*.

Se třetí náběžnou hranou začínáme data posílat na vstup hašovací komponenty. Posíláme i informaci o platnosti bajtů.

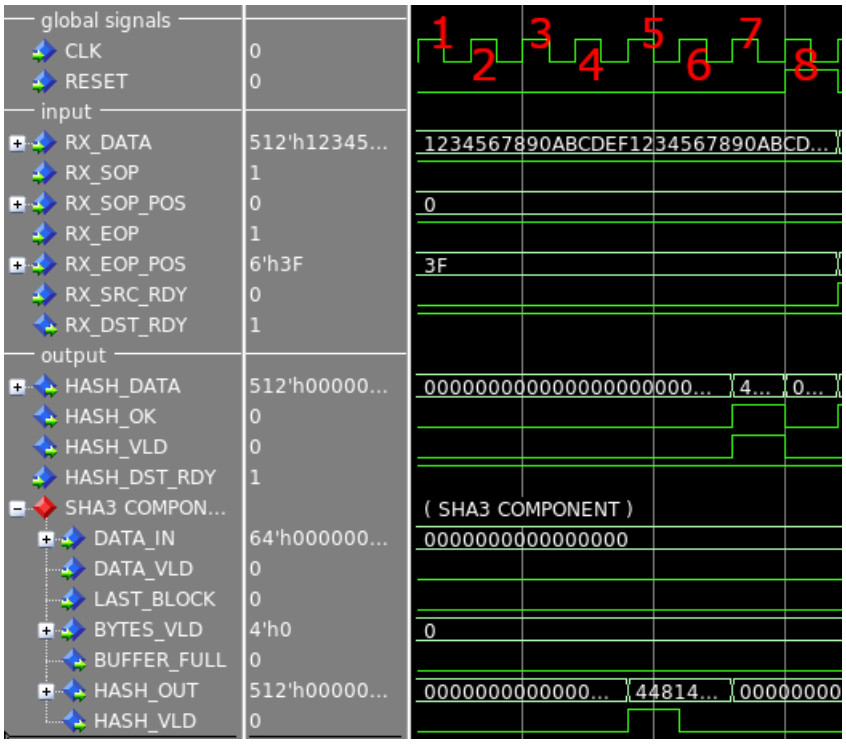
V desátém taktu odešleme informaci o posledním bloku zprávy. S náběžnou hranou jedenáctého taktu je zásobník zaplněn a čeká se na hašovací komponentu až začne data zpracovávat.



Obr. 4.7: Simualce se sběrnicí - přijmutí zprávy

Na obrázku 4.8 si můžeme prohlédnout následné zpracování výsledku. S náběžnou hranou pátého taktu je haš platný a je uložen do vnitřního registru wrapperu[1]. Následně je hašovací komponenta vyresetována a jelikož je signál *HASH_DST_RDY* aktivní, další komponenta je připravena přijmout data. Přiřadíme na výstup *HASH_DATA* výsledný haš. Výstupy *HASH_OK* a *HASH_VLD* signalizujeme platnost výsledného haše.

[1]Hyerarchický provek pro řazení komponent.



Obr. 4.8: Simualce se sběrnicí - výpočet haše

V tabulce 4.7 jsou popsány dosud nepopsané signály.

Název signálu	Šířka v bitech	Popis
CLK	1	hodinový signál
RESET	1	globální reset
RX_DATA	512	vstupní data
RX_SOP	1	informace o tom, zda data obsahují začátek paketu
RX_SOP_POS	6	pozice začátku paketu
RX_EOP	1	informace o tom, zda data obsahují konec paketu
RX_EOP_POS	6	pozice konce paketu
RX_SRC_RDY	*1*	vstupní data jsou platná
RX_DST_RDY	*1*	následující destinace je připravena přijmout data
HASH_DATA	*512*	výsledný haš
HASH_OK	*1*	platnost haše
HASH_VLD	*1*	platnost haše
HASH_DST_RDY	*1*	následující destinace je připravena přijmout data

Tab. 4.7: Popis signálů propojení sběrnice a hašovací komponenty

4.3.2 Ověření funkcionality v reálné aplikaci

Na obrázku 4.9 si můžeme prohlédnout zjednodušené schéma aplikace. Účel bloku zpracování paketů je například zahodit nevyhovující pakety nebo vyčíst data z těch paketů, které jsou pro nás zajímavé. Následně jsou takhle zpracovaný pakety přeposlány do SHA-3 komponenty, kde jsou zahašovány a předány dál.

Obr. 4.9: Zjednodušené schéma aplikace

Na obrázku 4.10 si můžeme prohlédnout příklad odeslaných dat do karty.

```
ff ff ff ff ff ff 00 13  3b 9c 7d 67 08 00 45 00
00 48 68 4d 00 00 80 11  4f 39 c0 a8 00 cf c0 a8
00 ff e1 15 e1 15 00 34  ec 3b 53 70 6f 74 55 64
70 30 9a 13 d2 8f 43 12  b4 fa 00 01 00 04 48 95
c2 03 a7 2b 8c 96 3b a7  83 22 e4 8a 94 f8 1c e5
7f 1f b3 69 1a ba
```

Obr. 4.10: Příklad odeslaných dat

Na obrázku 4.11 si můžeme prohlédnout přijatá zahašovaná data z obrázku 4.10.

```
3f e6 23 1e 6d 52 b6 09  75 78 2b 5d ed a6 cd a6
cf 42 aa 95 c9 2a d9 1c  a7 84 3e ca b8 26 bd 94
94 89 14 3d a2 7f 7c 60  e9 22 5a 85 06 f5 0b 41
29 75 b3 23 2d b1 ae e4  1a 5f f5 47 9c 96 65 1f
```

Obr. 4.11: Příklad přijatých dat

Data byla odesílána a vyčítána pomocí řídícího softwaru karty.

4.3.3 Dosažené výsledky

V následující podkapitole si popíšeme výsledky, kterých jsme dosáhli při implementaci. Veškeré výsledky využitých zdrojů a časování byly získány pomocí softwaru **Vivado 2019.1**.

V tabulce 4.8 jsou uvedeny jednotlivé využité zdroje FPGA. Můžeme si všimnou, že nejvíce využiváný blok je šestivstupá LUT naopak nejméně využívaná je dvouvstupá LUT.

Blok	**Použitých**
LUT6	1499
LUT5	669
LUT4	668
LUT3	353
LUT2	135
FLIP FLOP Registrů	2715
LUT celkem	3324

Tab. 4.8: Využité zdroje FPGA

V tabulce 4.9 je uvedeno kolik která komponenta spotřebovává zdrojů. Jak bylo předpokládáno, tak nejvíce zdrojů spotřebovává právě hašovací komponenta, která realizuje výpočet.

Komponenta	Počet využitých LUT	Počet využitých registrů
SHA3	3324	2715
PADDING	146	584
HASH	3178	2120

Tab. 4.9: Využité zdroje jednotlivých komponent

V tabulce 4.10 jsou výsledky časování komponenty pro 200 MHz získané při implementaci. V tabulce **WNS** - worst negative slack, **WHS** - worst hold slack, **WPWS** - worst pulse width slack.

WNS [ns]	WHS [ns]	WPWS [ns]
1,304	0,006	2,225

Tab. 4.10: Časovací údaje komponenty

Z námi známé hodnoty **WNS** jsme schopni spočítat maximální frekvence komponenty a to pomocí iterování vzorce 4.1 z [12].

$$F_{max} = \frac{1}{Period - slack} \tag{4.1}$$

Výpočet je iterován dokud nedosáhneme záporné hodnoty WNS. Předchozí vypočítaná frekvence je pak frekvencí maximální. Postup výpočtu maximální frekvence si můžeme prohlédnout v tabulce 4.11. V rovnici 4.2 si můžeme prohlédnout postup výpočtu pro nultou iteraci.

Iterace	WNS [ns]	Fmax [MHz]
0	1,304	270
1	0,619	324
2	0,366	365
3	0,191	398
4	0,063	410
5	-1,296	nesplněno

Tab. 4.11: Postup výpočtu F_{max}

Z tabulky je zřejmé, že maximální frekvence komponenty je přibližně 410 MHz.

$$F_{max} = \frac{1}{Period - slack} = \frac{1}{5 \times 10^{-9} - 1,304 \times 10^{-9}} \approx 270MHz \tag{4.2}$$

V reálne aplikaci jsme naměřili propustnost 4,51 Gbps. Při maximální frekvenci bychom mohli dosáhnout teoreticky propustnosti okolo 9 Gbps. Měření probíhá tak,

že dvě karty jsou propojeny vůči sobě. Zapojení se využívá z toho důvodu, že jedna karta nám slouží jako generátor dat a na druhé kartě je nahrán měřený design. Propustnost pak vyhodnocuje software karty.

Propustnost se může mírně lišit v závilosti na počtu dlouhých a krátkých zpráv. Pokud totiž vysíláme spíše krátké zprávy, tak komponenta musí častěji čekat na vypočítání haše a nemůže přijmat data do zásobníku, čímž je propustnost mírně ovlivňována.

Oproti námi testované již navrhnuté implementaci jsme docílili větší pracovní frekvence, maximální frekvence a nižší spotřeby LUT. Nevýhodou ovšem může být počítání pouze jedné permutace za takt a tedy nižní propustnost a také nutnost posílat s každými platnými daty informaci o platnosti bajtů. Mezi další výhody můžeme zařadit schopnost provést dodatečné zarovnání v případě, že poslední blok zprávy měl všechny bajty platné a zaplnil zásobník.

Závěr

Hlavním cílem práce bylo seznámit se s algoritmem SHA-3 a provést jeho implementaci do technologie FPGA.

V rámci seznámení se s funkcí SHA-3 jsme provedli implementaci v jazyce Python. Implementace byla úspešně otestována pomocí generování náhodných vektorů a pomocí oficiálních testovacích vektorů.

Pro lepší představu jak postupovat při návrhu do FPGA bylo prostudováno několik již navrhnutých implementací. Funkčnost jedné z implementací byla ověřena v simulačním programu ModelSim. Prostudování přispělo k lepší představě o požadavcích na zdroje a lepší schopnosti odhadnout kolik iterací jsme schopni stihnout v jednom hodinovém taktu.

Následně byl proveden jednoduchý blokový návrh. V návrhu se počítá se dvěmi permutacemi v jednom taktu. Více permutací by s největší pravděpodobností nesplnilo požadavky na časování a komponenta by spotřebovávala mnoho zdrojů FPGA.

Algoritmus byl následně přepsán do jazyka VHDL, který se jevil jako nejvhodnější kandidát. Provedli jsme simualce jednotlivých komponent na kterých byla ověřena jejich funkcionalita.

Během implementace bylo zjistěno, že při výpočtu dvou permutací nejsou splněny časovací podmínky při pracovní frekvenci 200 MHz. Časování nebylo splněno přibližně o -1,65 ns. Z toho důvodu jsme snížili počet permutací v taktu na jednu. V tomhle případě byla implementace úspěšná a časování bylo splněno.

Následně byla komponenta připojena k reálné aplikaci a byla ověřena společná funkcionalita.

Při implementaci jsme zjistili, že komponenta spotřebovává celkem 3324 LUT a 2715 registrů. Její maximální frekvence byla stanovena na přibližně 410 MHz. Její propustnost je přibližně 4,51 Gbps. Mezi výhody implementace můžeme zařadit vysokou maximální frekvenci, nízkou spotřebu zdojů v FPGA a schopnost provést dodatečné zarovnání zprávy.

Literatura

[1] DWORKIN, Morris J. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* [online]. 2015 [cit. 2020-01-01]. Dostupné z: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf

[2] SMART, Nigel. *Cryptography: An Introduction* [online]. 3rd Edition. [cit. 2020- 01-01]. Dostupné z: https://www.cs.umd.edu/ waa/414-F11/IntroToCrypto.pdf

[3] JÉGROVÁ, E. *Implementace kryptografických primitiv* [online]. Brno: Vysoké učení technické v Brně. Fakulta elektrotechniky a komunikačních technologií. 2015.

[4] KOLÁŘ, Milan. *Architektura a návrh FPGA obvodů* [online]. Liberec, 2010 [cit. 2020-05-01]. Habilitační práce. Technická univerzita v Liberci. Dostupné z: http://dspace.tul.cz/bitstream/handle/15240/39016/U_692_M.pdf

[5] ZDRÁLEK, Jaroslav. *Programovatelné logické prvky* [online]. Ostrava, 2007 [cit. 2020-05-01]. Studijní opora. Technická univerzita Ostrava. Dostupné z: http://www.elearn.vsb.cz/archivcd/FEI/PLP/zdralek_PLP.pdf

[6] ASIC-System on Chip-VLSI Design: What is the difference between FPGA and ASIC?. ASIC-System on Chip-VLSI Design [online] [cit. 2020-05-01]. Dostupné z: http://asic-soc.blogspot.com/2007/11/what-is-difference-between-fpga-and_06.html

[7] ATHANASIOUS, George S., MAKKAS, George-Prais, THEODORIDIS Georgios. *High throughput pipelined FPGA implementation of the new SHA-3 cryptographic hash algorithm* [online]. Athens, 2014 [cit. 2020-05-01]. Dostupné z : https://ieeexplore.ieee.org/document/6877931

[8] HSING, Homer. *SHA3 Core specification* [online]. 2013 [cit. 2020-05-01]. Dostupné z: https://opencores.org/projects/sha3

[9] KECCAK Team. .SHA-3 Hash Function Test Vectors for Hashing Byte-Oriented Messages [online]. [cit. 2020-03-01]. Dostupné z: https://csrc.nist.gov/Projects/cryptographic-algorithm-validation-program/Secure-Hashing

[10] PyCryptodome. [online]. [cit. 2020-03-01]. Dostupné z: https://www.pycryptodome.org/en/latest/

[11] LEVENT Ozturk. *ONLINE SHA-3 Keccak CALCULATOR - CODE GENERATOR* [online]. [cit. 2020-02-06]. Dostupné z: https://leventozturk.com/engineering/sha3/

[12] Xilinx. *Ar 57304: Vivado Timing - Where Can I Find the Fmax in the Timing Report?* [online]. [cit. 2020-04-06]. Dostupné z : https://www.xilinx.com/support/answers/57304.html

Seznam symbolů, veličin a zkratek

SHA-3 - **S**escure **H**ash **A**lghorithm 3
FPGA - **F**ield **P**rogrammable **G**ate **A**array (Programovatelná hradlová pole)
PLD - **P**rogrammable **L**ogic **D**evice
HDL - **H**ardware **D**escription **L**anguage (Jazyk k popisu hardware)
LUT - **L**ook **U**p **T**able (Vyhledávácí tabulku)
MUX - **M**ultiplexer
ASIC - **A**ppliacation-**s**pecific **i**ntegrated **c**ircuit (Zákaznicky integrovaný obvod)
Gbps - **G**iga**b**ites **p**er **s**econd (Gigabitů za sekundu)
Mbps - **M**ega**b**ites **p**er **s**econd (Megabitů za sekundu)
IOBs - **I**nput **O**utput **B**locks (Vstupně výstupní bloky)
LB - **L**ogic **B**locks (Logické bloky)
RAM - **R**andom-**A**ccess **M**memory (Pamět s přímým přístupem)
WNS - **W**orst **N**egative **S**lack
WHS - **W**orst **H**old **S**lack
WPWS - **W**orst **P**ulse **W**idth **S**lack

Seznam příloh

A Příloha obsahového CD