ELEKTROTEHNIČKI FAKULTET UNIVERZITETA U BEOGRADU

HARDVERSKA IMPLEMENTACIJA *JH* ALGORITMA ZA HEŠIRANJE

–Master rad–

Kandidat:

Milica Mijatović 2013/3306

Mentor:

doc. dr Zoran Čiča

Beograd, Maj 2015.

SADRŽAJ

1. UVOD

Sigurnost telekomunikacionih mreža je veoma važna zbog činjenice da se preko njih razmenjuju ogromne količine informacija i podataka. Informacije se prenose nesigurnim komunikacionim putevima koji se ne mogu fizički zaštititi, te je stoga vrlo važna primena zaštitnih komunikacionih mehanizama. Jedan od takvih mehanizama je **kriptografija**.

Kriptografija je nauka koja se bavi metodama očuvanja tajnosti informacija. Šifra i digitalni potpis su kriptografske tehnike koje se koriste da bi se implementirali bezbedonosni servisi. Šifrovanje je procedura koja transformiše originalnu informaciju u šifrovane podatke, dok je dešifrovanje obrnut proces, kada se šifrovani podaci transformišu u originalnu poruku. Oba procesa podrazumevaju korišćenje tzv. *ključa šifrovanja*. Kriptografija može biti simetrična i asimetrična, međutim, nijedan od pomenutih algoritama ne štiti integritet poruke koja je šifrovana. Ovo je vrlo važno iz razloga da je ključ „provaljen“ i da napadač šalje lažne poruke, ali i mogućnosti da je došlo do greške prilikom šifrovanja, tako da primljena poruka nije identična originalnom dokumentu. Iz tog razloga kreirane su funkcije za sažimanje, odnosno **heš algoritmi**. Najpoznatiji i najkorišćeniji heš algoritmi su: SHA, MD5, MDC-2, RIPEMD-160 itd. Heš algoritmi se svrstavaju u kriptografske algoritme *bez ključa* [1].

Kriptografski JH heš algoritam je bio jedan od pet finalista poslednje runde na takmičenju za novi SHA-3 standard (novembar, 2012). Zvanično postoje četiri verzije JH algoritma čija se razlika ogleda u dužini izlazne heš poruke. To su: JH-224, JH-256, JH-384 i JH-512, gde svaki broj reprezentuje dužinu heš izlaza.

U ovom radu biće prikazane specifikacije i performanse JH algoritma za heširanje. Za realizaciju implementacije koristi se VHDL programski jezik, a razvoj i verifikacija dizajna vrši se u ISE okruženju za FPGA čipove kompanije Xilinx. Svi projekti će biti priloženi u elektronskoj formi na priloženom CD-u.

Ostatak rada je organizovan na sledeći način: Drugo poglavlje daje osnovne informacije o heš algoritmima, njihovoj kriptografskoj primeni i napadima. Treće poglavlje detaljno opisuje osnove JH algoritma. Četvrto poglavlje opisuje hardversku implementaciju JH algoritma. Prvo su opisani interfejsi „crne kutije“, zatim tipovi promenljivih i stanja u kojima se može naći dizajn. Nakon toga sledi detaljan opis korišćenih procedura i opis rada top-level entiteta koji obavlja JH heširanje. Peto poglavlje se bavi analizom performansi realizovanog JH algoritma u zavisnosti od izabrane verzije algoritma. Drugi deo petog poglavlja se bavi verifikacijom dizajna. Poslednje, šesto poglavlje rezmira utiske autora ovog rada o realizovanom algoritmu, kao i predloge za njegovu optimizaciju.

2. KRIPTOGRAFSKI HEŠ ALGORITMI

U ovom poglavlju biće opisane osnovne osobine heš algoritama, kao i problemi sa kojima se ovi algoritmi susreću.

2.1. Definicija i osobine heš algoritama

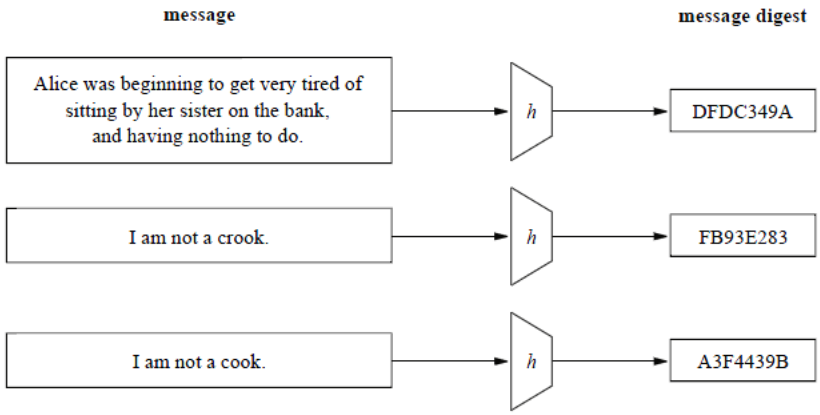
Jednosmerna heš funkcija (engl. *one-way hash function*), često zvana i sažimanje poruke (engl. *message digest*), ključna je u savremenoj kriptografiji. Jednosmerne heš funkcije su osnova za definisanje mnogih sigurnosnih protokola. One se već dugo koriste u računarskoj nauci. Heš funkcija je matematička ili neka druga funkcija koja uzima ulazni znakovni niz promenljive dužine, pod nazivom original (engl. *pre-image*), i konvertuje ga u (obično kraći) izlazni znakovni niz fiksne dužine, pod nazivom heš vrednost (engl. *hash value*). Jednosmerna heš funkcija radi u jednom smeru: izračunavanje heš vrednosti na osnovu ulazne vrednosti je jednostavno, ali je teško da se odredi ulazna vrednost koja daje određenu heš vrednost. Dobra heš funkcija takođe je bez kolizije (engl. *collision-free*), odnosno teško je da se generišu dve ulazne vrednosti koje generišu istu heš vrednost. Heš funkcija je javna, odnosno nema tajnosti postupka. Sigurnost heš funkcije leži u njenoj jednosmernosti. Nije vidljiv način na koji izlazna vrednost zavisi od ulazne. Promena jednog bita u originalnoj vrednosti dovodi do promene prosečno jedne polovine bitova heš vrednosti. Ako je zadata heš vrednost, neizvodljivo je izračunavanje ulazne vrednosti heš funkcije [2].

Heš funkcija h uzima ulaz proizvoljne dužine i računa izlaz fiksne dužine m, koji se zove heš vrednost, kao što je prikazano u jednačini 2.1:

$$h: \{0, 1\}^* \rightarrow \{0, 1\}^m \tag{2.1}$$

Kao što vidimo, ulaz je binarni string proizvoljne dužine, dok je izlaz takođe binarni string, ali fiksne dužine m. Kako se od heš funkcije očekuje da „smanji" ulaz, podrazumeva se da je veličina ulaza veća ili jednaka m.

Princip rada heš funkcija prikazan je na slici 2.1.1:



Slika 2.1.1 Princip rada heš funkcija

Pored toga što su determinističke i efikasno se računaju, od kriptografskih heš funkcija se uglavnom očekuje da se ponašaju kao slučajna funkcija - da daju „*random looking*" izlaz, odnosno nešto što izgleda kao slučajan niz simbola.

U praksi, to znači da je nemoguće predvideti bilo koji bit izlaza za dati ulaz. Time se postiže da i slični stringovi – oni koji se malo razlikuju, imaju potpuno različite heš vrednosti. Ovo se još zove *efekat lavine* (engl. *avalanche effect*) koji znači da male promene ulaza izazivaju značajne, nepredvidive promene na izlazu.

2.2. Napadi na kriptografske heš funkcije

U kriptografiji, napadač je često predstavljen kao neprijateljski algoritam. Kako je cilj kriptografije da obezbedi sigurnost komunikacije u realnim uslovima, polazna pretpostavka svake kriptografske aplikacije je postojanje napadača koji pokušava da dođe do poverljivih informacija. Kako su heš funkcije veoma važan segment sigurne komunikacije, izvesno je da će se određeni napadi usmeriti upravo na njih.

Shodno opisanim osobinama heš funkcija, razlikujemo tri moguća napada na njih:

- **Ulazni napad**. Kod ulaznog napada, za datu izlaznu vrednost d heš funkcije H, napadaču je cilj da nađe ulaznu poruku m koja odgovara izlaznoj heš vrednosti d, tj. $H(m) = d$.
- **Drugi ulazni napad.** Kod drugog ulaznog napada, za datu poruku m i heš funkciju H, napadač pokušava da nađe ulaznu poruku m_1, $m_1 \neq m$, tako da je $H(m) = H(m_1)$.
- **Napad kolizije.** Kod napada kolizije, za datu heš funkciju H, napadač pokušava da nađe dve poruke m i m_1 takve da $m \neq m_1$ i $H(m) = H(m_1)$.

Osnovni princip nalaže da je napadaču poznat kompletan dizajn heš funkcije. Time podrazumevamo da će napadač iskoristiti sve uočljive matematičke slabosti u dizajnu. Međutim, napad se često izvodi koristeći i neke druge, manje istaknute slabosti heš funkcije. Na primer, uspešan napad se može izvesti ukoliko se uoči da heš funkcija ima *slab efekat lavine*, odnosno da male promene ulaza izazivaju male promene na izlazu.

2.3. Primena kriptografskih heš funkcija

Heš funkcije su pronašle primenu u mnogim oblastima. Koriste se za kreiranje heš tabela, koje omogućuju brzo pretraživanje podataka bez obzira na veličinu tabele. Takođe se koriste i za pronalaženje dupliranih zapisa: svaki podatak se propusti kroz heš funkciju i pronađu se podaci koji imaju istu vrednost heša. Imaju široku upotrebu u kriptografiji. Takođe se mogu koristiti i za detekciju grešaka. Najznačajnija primena jeste tzv. digitalni potpis.

Digitalni potpis elektronskih podataka može da se posmatra u analogiji sa potpisom ili pečatom štampanih dokumenata. On omogućava da primalac poruke bude siguran da nije došlo do izmene originalnog sadržaja poruke, kao i da bude siguran u identitet pošiljaoca poruke. Drugim rečima, garantuje integritet poruke i autentifikaciju pošiljaoca iste. Pored toga, digitalni potpis ne može da se porekne, tako da onaj ko je poslao poruku kasnije ne može da tvrdi da je nije poslao ili da je neko drugi poslao umesto njega. Digitalni potpis poruke se formira korišćenjem tehnike asimetričnih ključeva. Pošiljalac kreira heš (šifrovani sažetak poruke) tako što na originalnu poruku primenjuje heš algoritam. Heš poruke predstavlja „digitalni otisak prsta" poruke. Ako bi došlo do najmanje promene u originalnoj poruci, promenio bi se i rezultujući heš poruke. Pošiljalac zatim vrši enkripciju heša svojim privatnim ključem. Ovako enkriptovan heš predstavlja digitalni potpis poruke. Pošiljalac dodaje na kraj originalne poruke i njen digitalni potpis i tako digitalno potpisanu poruku šalje. Na prijemu, primalac pomoću javnog ključa pošiljaoca vrši dekripciju digitalnog potpisa poruke kako bi dobio heš poslate poruke i poredi ga sa vrednošću heša koji on dobija primenjujući isti heš algoritam na samu primljenu poruku. Ako se dobijene heš vrednosti ne

razlikuju, primalac može da bude siguran da poruka nije izmenjena. Ako se vrednosti razlikuju, znači da je došlo do neovlašćenog menjanja sadržaja poruke i/ili da je neko drugi poslao poruku od onog za koga se izdaje [3].

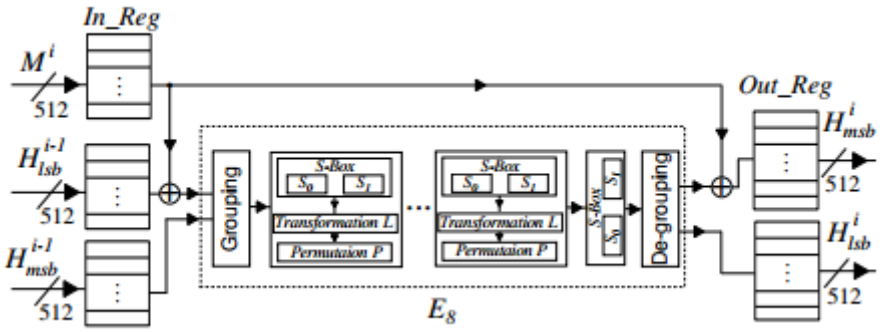
3. JH HEŠ ALGORITAM

Kriptografski JH heš algoritam je jedan od pet finalista poslednje runde na takmičenju za novi SHA-3 standard (oktobar, 2008). Autor ovog algoritma je Hongjun Wu, univerzitetski profesor matematike i računarskih nauka. Zvanično postoje četiri verzije JH algoritma čija se razlika ogleda u dužini izlazne heš poruke. To su: JH-224, JH-256, JH-384 i JH-512, gde svaki broj reprezentuje dužinu heš izlaza. U nastavku ovog poglavlja biće opisani koraci algoritma.

JH heš funkcija je iterativna heš funkcija koja vrši obrade ulaznih blokova poruke dužine 512 bita i na izlazu generiše veličinu koja može imati dužinu u bitima: 224, 256, 384 i 512, u zavisnosti koja je verzija algoritma korišćena.

Postoje dve tehnike koje se koriste za dizajn JH heš algoritma. Prva je kreiranje funkcije kompresije. Druga tehnika se bazira na AES metodologiji u smislu podele ulazne poruke na određeni broj komponenti (blokova), kako bi se na jednostavniji način izračunala heš vrednost, te na taj način poboljšale hardverske i softverske performanse.

Uopšteni blok dijagram koji predstavlja hardversku implementaciju verzije JH-256 algoritma je prikazan na slici 3.1:



Slika 3.1. Hardverska arhitektura funkcije kompresije JH-256 verije algoritma

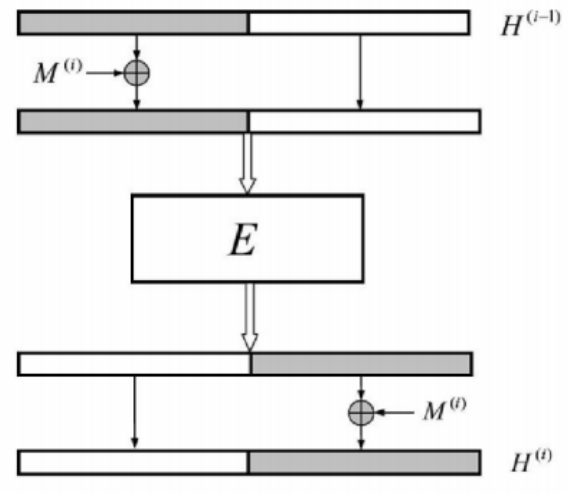
Pre objašnjenja gorepomenutih koraka sledi objašnjenje parametara:

d	dimenzija bloka poruke u bitima. d-dimenzionalni blok sadrži 2^d 4-bitnih elemenata.
$H^{(i)}$	i-ta heš vrednost, veličine h bita. $H^{(0)}$ je inicijalna heš vrednost; $H^{(N)}$ je finalna heš vrednost i ona se koristi za „odsecanje“ poslednjih bita (svaka verzija odseca različiti broj bita) da bi se dobio rezultujući heš poruke.
l	dužina poruke, M, u bitima.
$H^{(i),j}$	j-ti bit i-te heš vrednosti, $H^{(i)}$.
h	broj bita heš vrednosti $H^{(i)}$.
m	broj bita u jednom bloku poruke $M^{(i)}$, m=512.
M	poruka koja treba da se hešira.

$M^{(i)}$ *i*-ti blok poruke, veličine *m* bita.

N broj blokova u poruci nakon „padding“ procedure.

JH funkcija kompresije predstavlja bijektivnu funkciju. Predstavljena je na slici 3.2:



Slika 3.2. Struktura JH funkcije kompresije

2m-bitna heš vrednost $H^{(i-1)}$ i *m*-bitni ulazni blok $M^{(i)}$ su kompresovani tako da daju izlaz $H^{(i)}$. Kao što se vidi na slici 3.2, vrši se operacija XOR ulaznog bloka poruke sa levom polovinom heš vrednosti. Zatim se vrše određene matematičke operacije u bloku označene pod „E“. Na kraju se vrši XOR operacija ulaznog bloka poruke sa desnom polovinom heš vrednosti. Heš vrednost se računa na osnovu jednačine 3.1:

$$H^{(i)} = f\,(H^{(i-1)}, M^{(i)}) \tag{3.1}$$

gde je $H^{(i-1)}$ heš vrednost prethodnog bloka, a $M^{(i)}$ trenutni ulazni blok. Funkcija kompresije $f(H^{(i-1)}, M^{(i)})$ se može izračunati na osnovu jednačine 3.2:

$$f\,(H^{(i-1)}, M^{(i)}) = E\,(H^{(i-1)} \oplus M^{(i)} \| \, 0^{512}\,) \oplus 0^{512} \| \, M^{(i)} \tag{3.2}$$

gde simbol „‖“ označava operaciju konkatanacije, odnosno nadovezivanje bita trenutnog ulaznog bloka poruke $M^{(i)}$ sa bitima „0“, dok simbol „⊕“ predstavlja XOR operaciju. 0^{512} predstavlja string od 512 „0“ bita. Funkcija *E* vrši nekoliko operacija: grupisanje bita, nakon čega sledi 6*(*d*-1) rundi tzv. „*round*“ funkcije (koja obuhvata sledeće tri faze: odabir S-kutije (engl. *s-box*), linearna transformacija i permutacija), kao i degrupisanje bita.

Grupisanje bita predstavlja operaciju koja po određenom matematičkom algoritmu udružuje (grupiše) bite heš vrednosti $H^{(i-1)}$, u cilju dobijanja četvorobitnih reči koje se nadalje koriste u algoritmu. Degrupisanje je obrnut proces grupisanja, odnosno, biti iz četvorobitnih reči se raščlanjuju (degrupišu) u smislu ponovnog dobijanja osmobitnih reči, odnosno bajtova, koji sačinjavaju heš vrednost koja se nadalje koristi u algoritmu.

Nakon grupisanja bita u četvorobitne reči, vrši se odabir S-kutije. S-kutije koje se koriste u JH algoritmu su S_0 i S_1 i prikazane su u tabeli 3.1. One predstavljaju 4x4-bitne S-kutije:

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$S_0(x)$	9	0	4	11	13	12	3	15	1	10	2	6	7	5	8	14
$S_1(x)$	3	12	6	13	5	7	1	9	15	2	0	4	11	10	14	8

Tabela 3.1. S-box tabela

Odabir S-kutije predstavlja deo algoritma koji se vrši na osnovu vrednosti tzv. **konstante runde**. Konstanta runde predstavlja vektor od 256 bita, gde svaki bit predstavlja izbor S-kutije (S_0 ili S_1) za jednu četvorobitnu reč. Konstanta runde se u svakoj rundi ažurira i određuje po definisanom matematičkom algoritmu, te se stoga njena vrednost ispituje u svakoj rundi. Svaka četvorobitna reč se propušta kroz odgovarajuću (odabranu) S-kutiju - S_0 ili S_1 . Nakon odabira S-kutije sledi operacija linearne transformacije.

Linearna transformacija primenjuje tzv. MDS (*Maximum Distance Separable*) kod nad gorepomenutim četvorobitnim rečima i definisan je u okviru Galoisovog polja, GF (2^4). MDS (n, d, k) kod se definiše kao minimalno rastojanje između dve reči koje je maksimalno moguće. Rastojanje se računa po jednačini 3.3:

$$k=n-d+1 \tag{3.3}$$

gde *k* predstavlja rastojanje između reči, *n* dužinu reči u bitima, a *d* dimenziju reči (već pomenutu na početku ovog poglavlja).

JH algoritam primenjuje (4, 2, 3) MDS kod, po algoritmu prikazanom u jednačini 3.4:

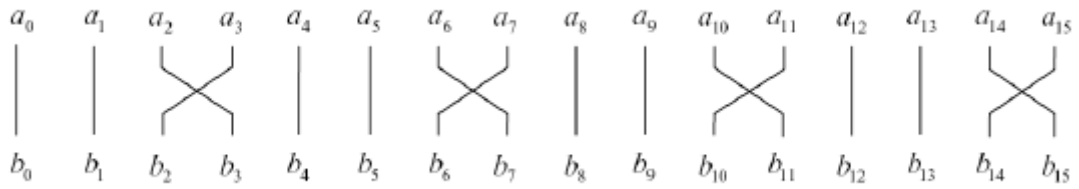
$$(C,D) = L(A,B) = (5 \bullet A + 2 \bullet B,\ 2 \bullet A + B) \tag{3.4}$$

gde *A, B, C* i *D* predstavljaju četvorobitne reči, a simbol „•“ predstavlja operaciju množenja. Operacija množenja u Galoisovom polju, GF (2^4), podrazumeva algoritam kao u tabeli 3.2:

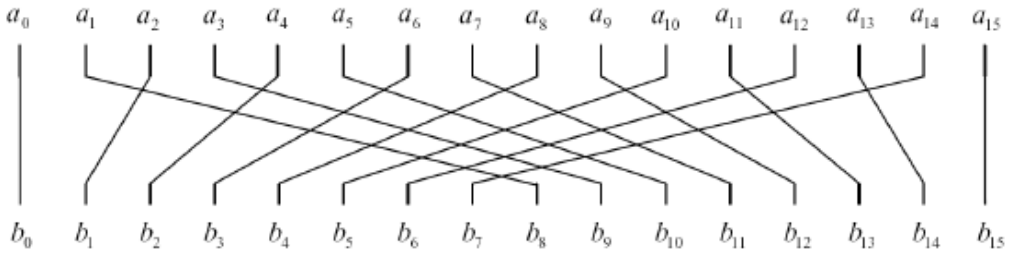
•	0	1	A	B
0	0	0	0	0
1	0	1	A	B
A	0	A	B	1
B	0	B	1	A

Tabela 3.2. Operacija množenja u Galoisovom polju

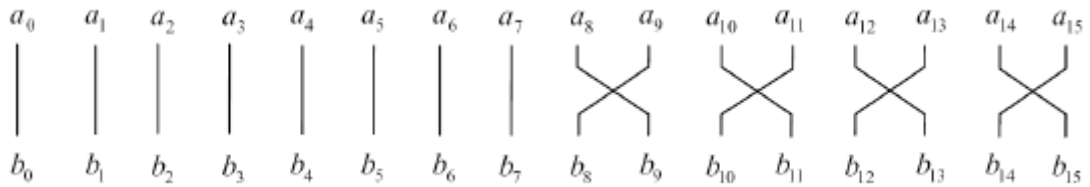
Nakon linearne transformacije sledi operacija permutacije nad četvorobitnim rečima. Permutacija podrazumeva rotiranje reči tako da se one nalaze pored drugih blokova u svakoj narednoj rundi. Algoritam na kom se zasniva permutacija predstavlja kompoziciju tri funkcije permutacije (π, P i α), kao na slikama 3.3, 3.4, 3.5 i 3.6 (za primer je uzeto d=4):



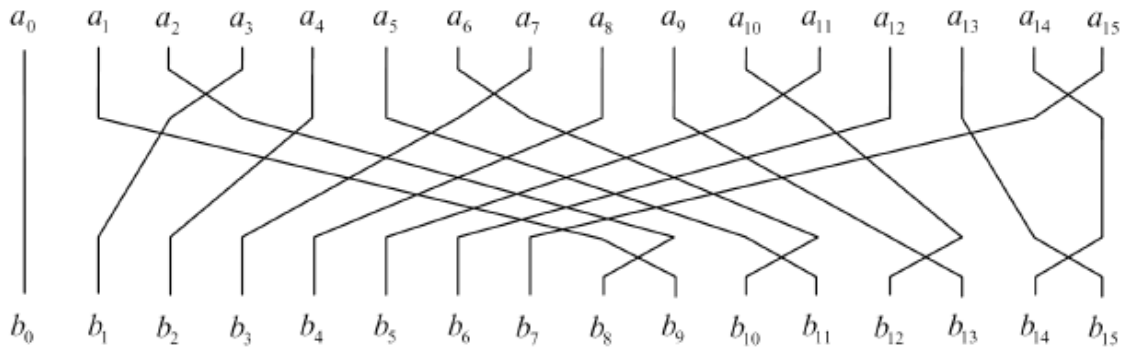
Slika 3.3. Operacija permutacije – funkcija permutacije π



Slika 3.4. Operacija permutacije – funkcija permutacije P



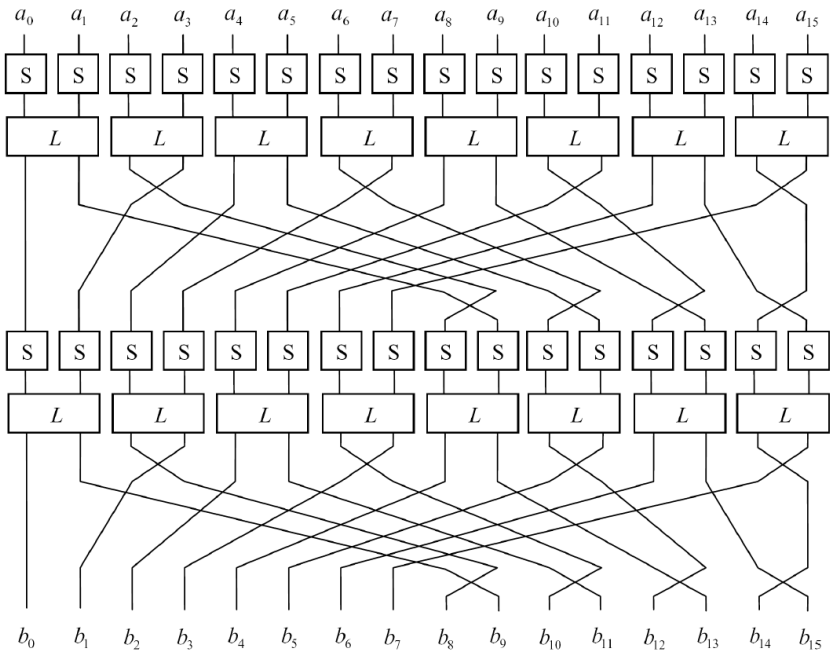
Slika 3.5. Operacija permutacije – funkcija permutacije α



Slika 3.6. Operacija permutacije – kompozicija tri funkcije permutacije

gde *a* i *b* predstavljaju bite četvorobitnih reči.

Dakle, princip „*round*“ funkcije koja obuhvata gorepomenute operacije je prikazan na slici 3.7 (za primer je uzeto *d*=4):



Slika 3.7. Dve runde „*round*“ funkcije za d=4

Ovakav način korišćenja funkcije kompresije je jednostavan i efikasan. Zahvaljujući permutaciji bita nisu potrebne dodatne promenljive koje bi se koristile, tako da je mnogo jednostavnija analiza sigurnosti funkcije kompresije za različite napade.

AES dizajn se koristi u smislu podele ulazne poruke na određeni broj komponenti, odnosno blokova. Po uopštenom AES dizajnu, ulazni biti su podeljeni na $\sum_{i=0}^{n-1} \alpha_i$ ($\alpha_i \geq 2$) elemenata, pri čemu ovi elementi sačinjavaju *d*-dimenzionalni niz. Nakon *r*-te runde, gorepomenuti MDS kod se primenjuje na (*r* mod *d*) dimenziju niza.

Postoji pet koraka koji sačinjavaju JH algoritam:

1. „*Padding*“ ulazne poruke;
2. Podela „*padding*“-ovane poruke na blokove;
3. Računanje inicijalne heš vrednosti;
4. Računanje finalne heš vrednosti;
5. Generisanje heš vrednosti, odnosno sažimanje poruke.

Napomena. U radu nadalje je korišćena vrednost za **d=8**, te odatle prefiks „8“ u nazivima funkcija.

3.1. „*Padding*“ poruke

Nad porukom *M* se vrši „padding“, na takav način da poruka postane umnožak od 512 bita. Pretpostavimo da je dužina poruke *l* bita. Bit “1” se dodaje na kraj poruke praćen sa 384 – 1 + (-*l*

mod 512) bita “0” (za l mod 512 = 0, 1, 2, …, 510, 511, broj bita “0” koji se dodaju je 383, 894, 893,…, 385, 384, respektivno). Zatim se na ovu dužinu dodaje 128 bita koji predstavljaju binarnu reprezentaciju l broja. Redosled bita unutar bajtova je predstavljen u *big-endian* formi, dok je redosled bajtova predstavljen u *little-endian* formi..

3.2. Podela poruke na blokove

Nakon „*padding*“ procedure, poruka se deli na N 512-bitnih blokova, $M^{(1)},M^{(2)},...,M^{(N)}$. 512-bitni blok poruke se predstavlja kao četiri 128-bitne reči. Prvih 128 bita bloka i je označeno kao $M_0^{(i)}$, sledećih128 bita je označeno kao $M_1^{(i)}$, i tako dalje do $M_3^{(i)}$. Redosled bita unutar bajtova je predstavljen u *big-endian* formi, dok je redosled bajtova predstavljen u *little-endian* formi.

3.3. Računanje inicijalne heš vrednosti $H^{(0)}$

Inicijalna heš vrednost zavisi od verzije algoritma koji se koristi (JH-224, JH-256, JH-384 ili JH-512). Prva dva bajta $H^{(-1)}$ imaju unapred definisanu vrednost (u zavisnosti od verzije), dok se ostali postavljaju na 0. Takođe se $M^{(0)}$postavlja na 0. Inicijalna vrednost se računa po jednačini 3.5:

$$H^{(0)} = F_8 (H^{(-1)}, M^{(0)}) \tag{3.5}$$

Prva dva bajta (16 bita) imaju vrednost 0x00E0, 0x0100, 0x0180, 0x0200 za JH-224, JH-256, JH-384 i JH-512, respektivno. Neka je $H^{(-1),j} = 0$ za $16 \leq j \leq 1023$. Postavimo $M^{(0)}$na 0. Inicijalna 1024-bitna heš vrednost $H^{(0)}$ se računa po jednačini 3.6:

$$H^{(0)} = F_8 (H^{(-1)}, M^{(0)}) \tag{3.6}$$

gde F_8 predstavlja funkciju kompresije definisanu u jednačini 3.2, za d=8.

3.4. Računanje finalne heš vrednosti $H^{(N)}$

Funkcija kompresije F_8 se primenjuje na svih N blokova poruke M. Finalna 1024-bitna heš vrednost $H^{(N)}$ se računa po jednačini 3.7:

$$H^{(i)} = F_8 (H^{(i-1)}, M^{(i)}), \ za\ 1 \leq i \leq N \tag{3.7}$$

3.5. Generisanje sažetka poruke

Sažetak poruke se dobija odsecanjem bita finalne heš vrednosti, u zavisnosti od verzije JH heš algoritma koji je primenjen [4]:

- **JH-224**: Prvih 224 bita $H^{(N)}$ vrednosti se odsecaju, tj. $H^{(N),800} \| H^{(N),801} \| ... \| H^{(N),1023}$.
- **JH-256**: Prvih 256 bita $H^{(N)}$ vrednosti se odsecaju, tj. $H^{(N),768} \| H^{(N),769} \| ... \| H^{(N),1023}$.

- **JH-384**: Prvih 384 bita $H^{(N)}$ vrednosti se odsecaju, tj. $H^{(N),640} \| H^{(N),641} \| ... \| H^{(N),1023}$.
- **JH-512**: Prvih 512 bita $H^{(N)}$ vrednosti se odsecaju, tj. $H^{(N),512} \| H^{(N),513} \| ... \| H^{(N),1023}$.

gde simbol „‖“ označava operaciju konkatanacije bita.

4. IMPLEMENTACIJA *JH* ALGORITMA

U ovom poglavlju biće opisan programski kod napisan za potrebe implementacije algoritma. Kod je napisan u VHDL (*VHSIC hardware description language*) programskom jeziku. Detaljan opis sledi za verziju čija je heš vrednost dužine 224 bita (JH-224), a potom će biti navedene razlike u delovima koda za ostale verzije JH algoritma (256, 384 i 512). Informacija o razlikama u koracima za različite dužine heš izlaza dobijena je iz originalnog dokumenta [4].

Dizajn opisan u ovom poglavlju ne vrši „*padding*“ proceduru, kao ni podelu poruke na blokove. Ove funkcije su izvršene prilikom verifikacije dizajna, na način koji će biti objašnjen u narednom pogavlju.

Dizajn se najpre može predstaviti crnom kutijom sa interfejsima.

4.1. Interfejsi

Dizajn sadrži ulazne i izlazne interfejse. Ulazni interfejsi su interfejsi sa imenom *clk*, *reset*, *first_block*, *last_block* i *message_block*. Izlazni interfejsi su interfejsi sa imenom: *recieve_ready*, *get_hash*, *init_done* i *hashvalue*.

Signal *clk* je korišćen kao signal takta, pri čemu je aktivna uzlazna ivica takta.

Signal *reset* vrši reset dizajna u početno stanje.

Signal *first_block* signalizira da je na ulazu prisutan prvi blok poruke čiji se heš računa.

Signalom *last_block* signalizira se da je na ulazu prisutan poslednji blok poruke. Ovaj signal predstavlja i „okidač“ za prelazak u stanje finalizacije (*finalization*), na takav način da aktivira signal *flag*.

Ulazni signal *message_block* predstavlja jedan blok poruke koju je potrebno obraditi.

Izlazni signal *recieve_ready* označava da se može postaviti novi blok poruke. Ovaj signal je neophodan jer obrada blokova traje nekoliko taktova.

Izlazni signal *get_hash* označava da je dizajn postavio heš vrednost na odgovarajući izlazni port (*hashvalue*) i da korisnik može da preuzme generisanu heš vrednost.

Izlazni signal *init_done* označava da je inicijalna heš vrednost izračunata i da se može početi sa obradom prvog bloka poruke.

Izlazni signal *hashvalue* sadrži rezultujući heš date poruke.

4.2. Unutrašnjost crne kutije

U ovom potpoglavlju biće opisani tipovi promenljivih definisani za potrebe ovog rada i konačni automat korišćen u dizajnu.

4.2.1. Tipovi promenljivih

Pored predefinisanih tipova podataka (*std_logic, std_logic_vector* i *integer*), korišćeni su i korisnički definisani tipovi podataka, koji su definisani u zasebnom VHDL paketu. Sledi lista korisnički definisanih tipova i kratko objašnjenje za kakve promenljive se koriste:

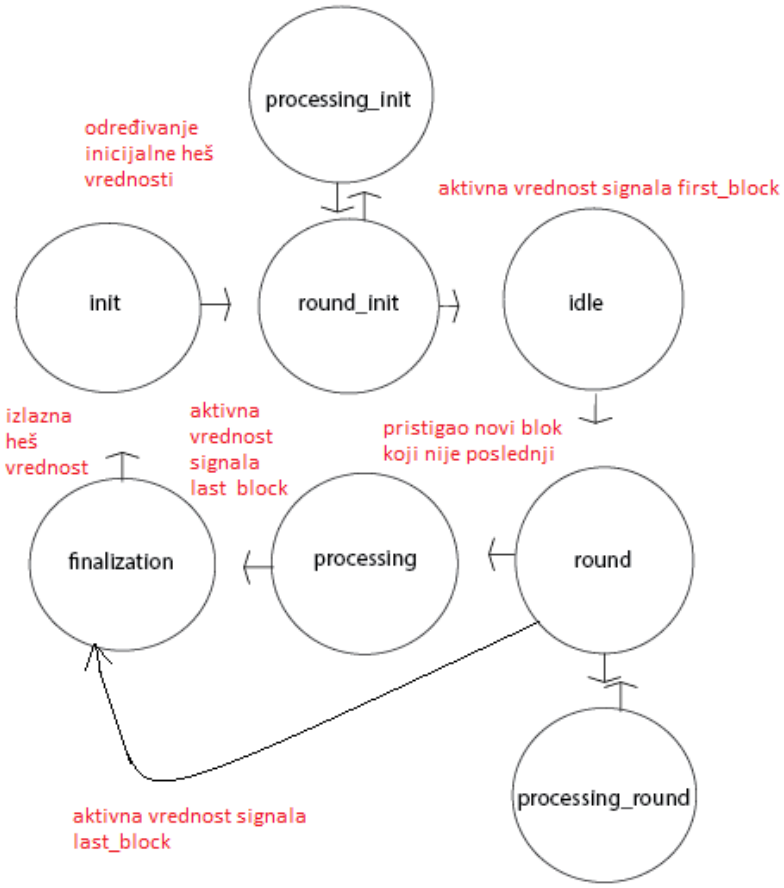
- *message_type*: Tip promenljive koji predstavlja veličinu poruke u bitima. Autor algoritma je definisao veličinu poruke od 512 bita. Ovaj tip je definisan kao niz od 64 bajta od po 8 bita.
- *hash_type*: Tip promenljive koji predstavlja veličinu heš funkcije. Autor algoritma je definisao veličinu poruke od 1024 bita. Ovaj tip je definisan kao niz od 128 bajtova od po 8 bita.
- *round_type*: Tip promenljive koji predstavlja veličinu promenljive nakon procedure grupisanja bita od po 4 bita, nad kojom se nadalje vrši tzv. „*round*“ funkcija. Autor algoritma je definisao veličinu poruke od 1024 bita. Ovaj tip je definisan kao niz od 256 bajtova koje sačinjava po 4 bita.
- *round_zero_type*: Tip promenljive koji predstavlja veličinu roundkonstante. Autor algoritma je definisao veličinu poruke od 256 bita. Ovaj tip je definisan kao niz od 64 bajta od po 4 bita.
- *s_box_type*: Tip promenljive koji predstavlja s-box tabelu. S-box tabela predstavlja preslikavanje 16 različitih četvorobitnih reči, pa je ovaj tip definisan kao niz od 16 četvorobitnih reči.
- *final_automat*: Tip promenljive koji predstavlja stanje konačnog automata. Ovo je enumerisani tip koji sadrži logičke nazive stanja automata. Ovaj tip će biti detaljnije opisan u sledećem odeljku.

4.2.2. Konačni automat

Na osnovu trenutne vrednosti promenljive koja predstavlja stanje konačnog automata, primenjuje se odgovarajući kod, tj. odgovarajuća faza u obradi poruke.

Obzirom da postoje 42 runde izvršavanja, a kombinaciona logika procedura za određivanje inicijalne, kao i finalne heš vrednosti je velika (kompajleru je potrebno više vremena da analizira dizajn i da ga sintetiše što je moguće optimalnije), dizajn se može podeliti na dva dela: prvi, gde se generiše inicijalna heš vrednost, i drugi, gde se generiše finalna heš vrednost. Runde se izvršavaju u stanju koji sadrži u sebi naziv „*round*“, što je detaljno objašnjeno u tekstu ispod.

Stanja koja su definisana su: *init, round_init, processing_init,idle, round, processing_round, processing, finalization*. Navedena stanja su prikazana na slici 4.2.2.1:



Slika 4.2.2.1. Stanje konačnog automata

Stanje *init* je stanje u kome se nalazi dizajn pre prijema bilo kakve poruke. U ovom stanju se vrši inicijalizacija stanja. U okviru ovog stanja se izvršava grupisanje bita u četvorobitne reči, , a ujedno se i definiše konstanta runde, čiju je vrednost definisao autor algoritma.

U stanju *round_init* se izvršavaju 42 runde *round* funkcije, koju čine operacije: odabir S-kutije, linearna transformacija i permutacija. Takođe i konstanta runde dobija novu vrednost koja se nadalje koristi u procesu *round* funkcije. Izvršen broj rundi se beleži u signalu *brojac* koji je tipa *integer*. Ovo je takođe stanje pre prijema bilo kakve poruke.

Stanje *processing_init* predstavlja stanje koje vraća dizajn u stanje *round_init*dok se ne izvrši svih 42 rundi. Ovo je takođe stanje pre prijema bilo kakve poruke.

Ova tri stanja se koriste za inicijalizaciju interne strukture koja se koristi pri heširanju, čiji je rezultat $H^{(0)}$.

U stanju *idle* se algoritam nalazi u trenutku kada čeka prijem poruke.

Stanje *round* ima istu funkciju kao stanje *round_init*, samo što se ovo stanje sada vrši nad porukom, odnosno stvarnim podacima.

Stanje *processing_round* ima istu funkciju kao stanje *processing_init*, samo što se ovo stanje sada vrši nad porukom, odnosno stvarnim podacima.

Stanje *finalization* označava izvršavanje finalnog dela heširanja, tj. generisanje sažetka poruke (engl. *message digest*), odnosno odsecanje određenog broja bita (u zavisnosti od verzije) poslednje izračunatog heša, tj. $H^{(N)}$ (izračunate heš funkcije nad poslednjim blokom).

Poslednje tri navedena stanja se koriste za heširanje poruke.

4.3. Opis algoritma

Najpre će biti objašnjene procedure koje predstavljaju korake algoritma, a potom će biti opisan i kod koji prati celokupan tok obrade poruke. Procedure, kao i pojedine konstante korišćene pri implementaciji ovog algoritma definisane su u paketu.

Spisak svih procedura koje se koriste u dizajnu su: *hash_initial, f8, e8_grouping, roundconstant_expanded_func, r8, s_box0_new, s_box1_new, update_roundconstant, e8_degrouping, f8_reverse, hash_calculate, hash_calculate_reverse, hash_final_func.*

Spisak promenljivih koje se koriste u kodu:

H	reprezentuje heš promenljivu.
roundconstant	reprezentuje konstantu runde.
Q	reprezentuje promenljivu nakon grupisanja bita iz H promenljive.
hash_final	reprezentuje finalnu heš vrednost.

4.3.1. Opis procedura

Procedura *hash_initial* dodeljuje promenljivoj *H* vrednost, tako da prva dva bajta imaju vrednost 0*x*00*E*0. Ova vrednost se dodeljuje iz razloga što se opisani algoritam odnosi na JH-224 verziju. Ostalim bajtovima se dodeljuje vrednost 0. Takođe, interna promenljiva *bufer* dobija vrednost 0. Nakon ovih inicijalizacija poziva se procedura *f8*.

Kod kojim se gorepomenute operacije vrše je sledeći:

```
hashbitlen_var:=hashbitlen;
FOR i IN 63 DOWNTO 0 LOOP
    bufer(i)(7 DOWNTO 0):= X"00";
END LOOP;
FOR i IN 127 DOWNTO 0 LOOP
    H(i)(7 DOWNTO 0):= X"00";
END LOOP;
H(1) := hashbitlen_var(7 DOWNTO 0);
H(0) := hashbitlen_var(15 DOWNTO 8);
f8(bufer, H,Q);
```

hashbitlen_var je interna promenljiva kojoj se dodeljuje vrednost konstante definisana unutar paketa (*hashbitlen*), i to je upravo vrednost 0*x*00*E*0. Svakom bajtu promenljive *bufer* se dodeljuje vrednost 0, kao i bajtovima promenljive *H*, izuzev prva dva. Procedura *f8* vrši operaciju XOR interne promenljive *bufer* sa prvom (levom) polovinom heš promenljive *H*. Zatim se poziva procedura za grupisanje bita, *e8_grouping*:

```
FOR i IN 63 DOWNTO 0 LOOP
    FOR j IN 7 DOWNTO 0 LOOP
        H(i)(j):=H(i)(j) XOR bufer(i)(j);
    END LOOP;
END LOOP;
e8_grouping(H,Q);
```

Grupisanje bita promenljive *H* u promenljivu *Q* koju sačinjavaju četvorobitne reči je predstavljeno na sledeći način kao u jednačini 4.1:

for i=0 to 2^{d-1}-1

{

$$q_{0,2i} = H^{i} \| H^{i+2^{d}} \| H^{i+2*2^{d}} \| H^{i+3*2^{d}} \quad (4.1)$$

$$q_{0,2i+1} = H^{i+2^{d-1}} \| H^{i+2^{d-1}+2^{d}} \| H^{i+2^{d-1}+2*2^{d}} \| H^{i+2^{d-1}+3*2^{d}}$$

}

gde simbol „‖“ označava operaciju konkatanacije bita.

Odnosno, kod koji prikazuje grupisanje:

```
FOR i IN 0 TO 15 LOOP
    FOR j IN 0 TO 7 LOOP
        q0 := A(i)(7-j);
        q1 := A(i+32)(7-j);
        q2 := A(i+64)(7-j);
        q3:=  A(i+96)(7-j);
        q0_new := A(i+16)(7-j);
        q1_new := A(i+32+16)(7-j);
        q2_new := A(i+64+16)(7-j);
        q3_new:=  A(i+96+16)(7-j);
        Q(2*i*8+2*j)(3 DOWNTO 0) := q0 & q1 & q2 & q3;
        Q(2*i*8+2*j+1)(3 DOWNTO 0) := q0_new & q1_new & q2_new & q3_new;
    END LOOP;
 END LOOP;
```

Promenljiva *A* je interna promenljiva unutar procedure *e8_grouping* koja dobija vrednost promenljive *H*. $q_{0,}$ $q_{1,}$ $q_{2,}$ $q_{3,}$ $q_{0\_new,}$ $q_{1\_new,}$ q_{2_new} i q_{3_new} predstavljaju 1-bitne promenljive tipa *STD_LOGIC*, pri čemu se prvoj dodeljuje prvih 255 bita promenljive H, zatim drugoj biti promenljive H koje su na pozicijama od 256-og do 511-og bita, zatim biti promenljive H koje su na pozicijama od 512-og do 767-og bita, i biti promenljive H koje su na pozicijama od 768-og do 1023-eg bita. Ove promenljive reprezentuju prvi deo jednačine 4.1. Poslednje četiri promenljive, $q_{0\_new,}$ $q_{1\_new,}$ q_{2_new} i $q_{3\_new,}$ reprezentuju drugi deo jednačine 4.1. Njima se dodeljuju biti po istom algoritmu kao za $q_{0,}$ $q_{1,}$ $q_{2,}$ q_{3}, samo što im se dodeljuju biti koji se nalaze na pozicijama pomerenim za 128 bita, respektivno. Operacijom konkatanacije u kodu „**&**“ ovi biti se povezuju i čine strukturu promenljive *Q*.

Degrupisanje bita se vrši na sličan, odnosno obrnut način, kao u jednačini 4.2:

for i=0 to 2^{d-1}-1

{

$$Q^{i} \| Q^{i+2^{d}} \| Q^{i+2*2^{d}} \| Q^{i+3*2^{d}} = q_{6(d-1),2i} \quad (4.2)$$

$$Q^{i+2^{d-1}} \| Q^{i+2^{d-1}+2^{d}} \| Q^{i+2^{d-1}+2*2^{d}} \| Q^{i+2^{d-1}+3*2^{d}} = q_{6(d-1),2i+1}$$

}

gde simbol „‖“ označava operaciju konkatanacije bita. Odnosno kod koji prikazuje gorepomenutu operaciju:

```
FOR i IN 0 TO 15 LOOP
    FOR j IN 0 TO 7 LOOP
        A(i)(7-j):=Q(2*i*8+2*j)(3);
        A(i+32)(7-j):=Q(2*i*8+2*j)(2);
```

```
        A(i+64)(7-j):=Q(2*i*8+2*j)(1);
        A(i+96)(7-j):=Q(2*i*8+2*j)(0);
        A(i+16)(7-j):=Q(2*i*8+2*j+1)(3);
        A(i+32+16)(7-j):=Q(2*i*8+2*j+1)(2);
        A(i+64+16)(7-j):=Q(2*i*8+2*j+1)(1);
        A(i+96+16)(7-j):=Q(2*i*8+2*j+1)(0);
    END LOOP;
END LOOP;
```

Dodela vrednosti konstante, unapred definisane od strane autora algoritma, promenljivoj *roundconstant*, vrši se unutar procedure *roundconstant_expanded_func*:

```
roundconstant_zero_var:=roundconstant_zero;
FOR i IN 0 TO 63 LOOP
    roundconstant(i)(3 DOWNTO 0):=roundconstant_zero_var(255 DOWNTO 252);
    roundconstant_zero_var:=SHL(roundconstant_zero_var,"0100");
END LOOP;
```

Interna promenljiva kojoj se dodeljuje vrednost konstante, *roundconstant_zero_var*, definisana je unutar paketa (*roundconstant_zero*), i to je vrednost:

0x6a09e667f3bcc908b2fb1366ea957d3e3adec17512775099da2f590b0667322a

SHL operacija pomera bite za četiri mesta ulevo, odnosno, dodeljuje se vrednost narednoj četvorobitnoj reči.

Nakon grupisanja i dodele vrednosti promenljivoj *roundconstant*, vrše se 42 runde „*round*“ funkcije, koja obuhvata tri operacije: izbor i izvršavanje S-kutije, linearna transformacija i permutacija. Odabir S-kutije se vrši na osnovu vrednosti bita promenljive *roundconstant*. Dakle, potrebno je predstaviti promenljivu *roundconstant* kao 256 1-bitnih reči (*roundconstant_expanded*), a zatim izršiti odabir s-kutije, odnosno:

```
FOR i IN 0 TO 63 LOOP
    FOR k IN 0 TO 3 LOOP
        roundconstant_expanded(4*i+k):=roundconstant(i)(3-k);
        IF (roundconstant_expanded(4*i+k)='0') THEN
            s_box0_new(Q(4*i+k)(3 DOWNTO 0));
        ELSE
            s_box1_new(Q(4*i+k)(3 DOWNTO 0));
        END IF;
    END LOOP;
END LOOP;
```

s_box0_new uzima vrednosti iz prvog reda tabele, a *s_box1_new* iz drugog reda iste tabele, odnosno,

```
index:=conv_integer(element);
element:=s_box0(index);
```

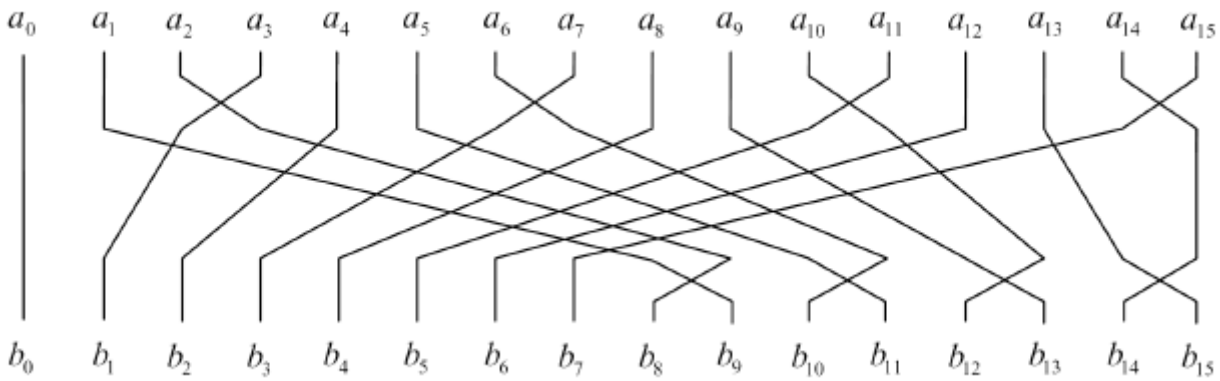
Linearna transformacija je definisana kao u jednačini 4.3:

$$\begin{aligned} D^0 &= B^0 \oplus A^1; & D^1 &= B^1 \oplus A^2; \\ D^2 &= B^2 \oplus A^3 \oplus A^0; & D^3 &= B^3 \oplus A^0; \\ C^0 &= A^0 \oplus D^1; & C^1 &= A^1 \oplus D^2; \\ C^2 &= A^2 \oplus D^3 \oplus D^0; & C^3 &= A^3 \oplus D^0; \end{aligned} \tag{4.3}$$

gde su A i B reprezentacije promenljive *Q*, a C i D reprezentacije interne *w* promenljive, koja se kasnije putem operacije permutacije dodeljuje ponovo promenljivoj *Q*, odnosno kod koji opisuje linearnu transformaciju:

```
FOR i IN 0 TO 127 LOOP
    w(2*i+1)(0):=Q(2*i+1)(0) XOR Q(2*i)(3);
    w(2*i+1)(1):=Q(2*i+1)(1) XOR Q(2*i)(0) XOR Q(2*i)(3);
    w(2*i+1)(2):=Q(2*i+1)(2) XOR Q(2*i)(1);
    w(2*i+1)(3):=Q(2*i+1)(3) XOR Q(2*i)(2);
    w(2*i)(0):=Q(2*i)(0) XOR w(2*i+1)(3);
    w(2*i)(1):=Q(2*i)(1) XOR w(2*i+1)(0) XOR w(2*i+1)(3);
    w(2*i)(2):=Q(2*i)(2) XOR w(2*i+1)(1);
    w(2*i)(3):=Q(2*i)(3) XOR w(2*i+1)(2);
END LOOP;
```

Operacija permutacije je prikazana na slici 4.3.1:



Slika 4.3.1. Operacija permutacije za d=4

Kod koji opisuje operaciju permutacije je:

```
FOR i IN 0 TO 63 LOOP
    Q(2*i):=w(4*i);
    Q(2*i+1):=w(4*i+3);
    Q(2*i+128):=w(4*i+2);
    Q(2*i+129):=w(4*i+1);
END LOOP;
```

gde *Q* reprezentuje vrednost *b* sa slike 4.3.1, a *w* vrednost *a* koja je prikazana na slici 4.3.1.

42 runde „obuhvataju“ funkciju „*round*“, ali ujedno i ažuriranje round konstante. To se vrši pomoću funkcije *update_roundconstant*. Ona obuhvata operacije: propuštanje kroz S-kutiju, linearna transformacija i permutacija:

```
FOR i IN 0 TO 63 LOOP
    s_box0_new(roundconstant(i));
END LOOP;
FOR i IN 0 TO 31 LOOP
    w(2*i+1)(0):=roundconstant(2*i+1)(0) XOR roundconstant(2*i)(3);
    w(2*i+1)(1):=roundconstant(2*i+1)(1) XOR roundconstant(2*i)(0)  XOR
roundconstant(2*i)(3);
    w(2*i+1)(2):=roundconstant(2*i+1)(2) XOR roundconstant(2*i)(1);
    w(2*i+1)(3):=roundconstant(2*i+1)(3) XOR roundconstant(2*i)(2);
    w(2*i)(0):=roundconstant(2*i)(0) XOR w(2*i+1)(3);
    w(2*i)(1):=roundconstant(2*i)(1) XOR w(2*i+1)(0) XOR w(2*i+1)(3);
    w(2*i)(2):=roundconstant(2*i)(2) XOR w(2*i+1)(1);
```

```
    w(2*i)(3):=roundconstant(2*i)(3) XOR w(2*i+1)(2);
END LOOP;
FOR i IN 0 TO 15 LOOP
    roundconstant(2*i):=w(4*i);
    roundconstant(2*i+1):=w(4*i+3);
    roundconstant(2*i+32):=w(4*i+2);
    roundconstant(2*i+33):=w(4*i+1);
END LOOP;
```

XOR-ovanje druge polovine heš promenljive sa porukom vrši se u procedurama *f8_reverse* i *hash_calculate_reverse*:

```
FOR i IN 63 DOWNTO 0 LOOP
    FOR j IN 7 DOWNTO 0 LOOP
        H(i+64)(j):=A(i+64)(j) XOR bufer(i)(j);
    END LOOP;
END LOOP;
```

Dodela bita iz bloka poruke internoj promenljivoj *bufer* unutar procedura *hash_calculate* i *hash_calculate_reverse* vrši se na sledeći način:

```
bufer(0)(7 DOWNTO 0):=message_block(7 DOWNTO 0);
bufer(1)(7 DOWNTO 0):=message_block(15 DOWNTO 8);
bufer(2)(7 DOWNTO 0):=message_block(23 DOWNTO 16);
bufer(3)(7 DOWNTO 0):=message_block(31 DOWNTO 24);
bufer(4)(7 DOWNTO 0):=message_block(39 DOWNTO 32);
bufer(5)(7 DOWNTO 0):=message_block(47 DOWNTO 40);
bufer(6)(7 DOWNTO 0):=message_block(55 DOWNTO 48);
bufer(7)(7 DOWNTO 0):=message_block(63 DOWNTO 56);
bufer(8)(7 DOWNTO 0):=message_block(71 DOWNTO 64);
bufer(9)(7 DOWNTO 0):=message_block(79 DOWNTO 72);
bufer(10)(7 DOWNTO 0):=message_block(87 DOWNTO 80);
bufer(11)(7 DOWNTO 0):=message_block(95 DOWNTO 88);
bufer(12)(7 DOWNTO 0):=message_block(103 DOWNTO 96);
bufer(13)(7 DOWNTO 0):=message_block(111 DOWNTO 104);
bufer(14)(7 DOWNTO 0):=message_block(119 DOWNTO 112);
bufer(15)(7 DOWNTO 0):=message_block(127 DOWNTO 120);
bufer(16)(7 DOWNTO 0):=message_block(135 DOWNTO 128);
bufer(17)(7 DOWNTO 0):=message_block(143 DOWNTO 136);
bufer(18)(7 DOWNTO 0):=message_block(151 DOWNTO 144);
bufer(19)(7 DOWNTO 0):=message_block(159 DOWNTO 152);
bufer(20)(7 DOWNTO 0):=message_block(167 DOWNTO 160);
bufer(21)(7 DOWNTO 0):=message_block(175 DOWNTO 168);
bufer(22)(7 DOWNTO 0):=message_block(183 DOWNTO 176);
bufer(23)(7 DOWNTO 0):=message_block(191 DOWNTO 184);
bufer(24)(7 DOWNTO 0):=message_block(199 DOWNTO 192);
bufer(25)(7 DOWNTO 0):=message_block(207 DOWNTO 200);
bufer(26)(7 DOWNTO 0):=message_block(215 DOWNTO 208);
bufer(27)(7 DOWNTO 0):=message_block(223 DOWNTO 216);
bufer(28)(7 DOWNTO 0):=message_block(231 DOWNTO 224);
bufer(29)(7 DOWNTO 0):=message_block(239 DOWNTO 232);
bufer(30)(7 DOWNTO 0):=message_block(247 DOWNTO 240);
bufer(31)(7 DOWNTO 0):=message_block(255 DOWNTO 248);
bufer(32)(7 DOWNTO 0):=message_block(263 DOWNTO 256);
bufer(33)(7 DOWNTO 0):=message_block(271 DOWNTO 264);
bufer(34)(7 DOWNTO 0):=message_block(279 DOWNTO 272);
bufer(35)(7 DOWNTO 0):=message_block(287 DOWNTO 280);
bufer(36)(7 DOWNTO 0):=message_block(295 DOWNTO 288);
bufer(37)(7 DOWNTO 0):=message_block(303 DOWNTO 296);
bufer(38)(7 DOWNTO 0):=message_block(311 DOWNTO 304);
bufer(39)(7 DOWNTO 0):=message_block(319 DOWNTO 312);
bufer(40)(7 DOWNTO 0):=message_block(327 DOWNTO 320);
```

```
bufer(41)(7 DOWNTO 0):=message_block(335 DOWNTO 328);
bufer(42)(7 DOWNTO 0):=message_block(343 DOWNTO 336);
bufer(43)(7 DOWNTO 0):=message_block(351 DOWNTO 344);
bufer(44)(7 DOWNTO 0):=message_block(359 DOWNTO 352);
bufer(45)(7 DOWNTO 0):=message_block(367 DOWNTO 360);
bufer(46)(7 DOWNTO 0):=message_block(375 DOWNTO 368);
bufer(47)(7 DOWNTO 0):=message_block(383 DOWNTO 376);
bufer(48)(7 DOWNTO 0):=message_block(391 DOWNTO 384);
bufer(49)(7 DOWNTO 0):=message_block(399 DOWNTO 392);
bufer(50)(7 DOWNTO 0):=message_block(407 DOWNTO 400);
bufer(51)(7 DOWNTO 0):=message_block(415 DOWNTO 408);
bufer(52)(7 DOWNTO 0):=message_block(423 DOWNTO 416);
bufer(53)(7 DOWNTO 0):=message_block(431 DOWNTO 424);
bufer(54)(7 DOWNTO 0):=message_block(439 DOWNTO 432);
bufer(55)(7 DOWNTO 0):=message_block(447 DOWNTO 440);
bufer(56)(7 DOWNTO 0):=message_block(455 DOWNTO 448);
bufer(57)(7 DOWNTO 0):=message_block(463 DOWNTO 456);
bufer(58)(7 DOWNTO 0):=message_block(471 DOWNTO 464);
bufer(59)(7 DOWNTO 0):=message_block(479 DOWNTO 472);
bufer(60)(7 DOWNTO 0):=message_block(487 DOWNTO 480);
bufer(61)(7 DOWNTO 0):=message_block(495 DOWNTO 488);
bufer(62)(7 DOWNTO 0):=message_block(503 DOWNTO 496);
bufer(63)(7 DOWNTO 0):=message_block(511 DOWNTO 504);
```

Odsecanje poslednjih bita iz heš vrednosti vrši se pomoću funkcije *hash_final_func*:

```
hash_final(223 DOWNTO 216):=H(100)(7 DOWNTO 0);
hash_final(215 DOWNTO 208):=H(101)(7 DOWNTO 0);
                  ...
hash_final(7 DOWNTO 0):=H(127)(7 DOWNTO 0);
```

4.3.2. Opis rada top-level entiteta

Nakon što su objašnjene procedure potrebne za implementaciju algoritma, biće opisan kod u arhitekturi top-level entiteta kojim se realizuje celokupna obrada poruke. Pre nego što se počne sa opisom koda, radi uvida u celokupan kod biće prikazana arhitektura top-level entiteta:

```
ARCHITECTURE JH_algorithm OF JH_224 IS

     SIGNAL state_automat:final_automat;
     SIGNAL brojac: INTEGER;
     SIGNAL message_block_internal: STD_LOGIC_VECTOR(511 DOWNTO 0);
     SIGNAL flag: STD_LOGIC;

BEGIN

    PROCESS(reset,clk)

        VARIABLE H: hash_type;
        VARIABLE roundconstant:round_zero_type;
        VARIABLE Q: round_type;
        VARIABLE hash_final: STD_LOGIC_VECTOR(223 DOWNTO 0);

     BEGIN

        IF (reset='1') THEN
            state_automat<= init;
            flag<='0';
            get_hash<='0';
            init_done<='0';
            receive_ready<='0';
```

```
     hashvalue<=(OTHERS=>'0');
ELSIF (clk'EVENT AND clk='1') THEN
       CASE (state_automat) IS
             WHEN init =>
                  init_done<='0';
                  get_hash<='0';
                  hash_initial(H,Q);
                  roundconstant_expanded_func(roundconstant);
                  state_automat<=round_init;
                  brojac<=0;
                  receive_ready<='0';
             WHEN round_init =>
                  r8(roundconstant,Q);
                  update_roundconstant(roundconstant);
                  brojac<=brojac+1;
                  IF (brojac>40) THEN
                     f8_reverse(Q,H);
                     state_automat<= idle;
                     init_done<='1';
                  ELSE
                     state_automat<= processing_init;
                     init_done<='0';
                  END IF;
             WHEN processing_init =>
                  state_automat<= round_init;
             WHEN idle =>
                  get_hash<='0';
                  receive_ready<='1';
                  IF (first_block='1') THEN
                      brojac<=0;
                      message_block_internal<=message_block;
                      roundconstant_expanded_func(roundconstant);
                      hash_calculate(message_block, H, Q);
                      state_automat<= round;
                      receive_ready<='0';
                      IF (last_block='1') THEN
                          flag<='1';
                      ELSE
                          flag<='0';
                      END IF;
                      END IF;
             WHEN round =>
                  r8(roundconstant,Q);
                  update_roundconstant(roundconstant);
                  receive_ready<='0';
                  brojac<=brojac+1;
                  IF (brojac>40) THEN
                      IF (flag='1') THEN
                          hash_calculate_reverse(message_block, Q, H);
                          state_automat<= finalization;
                      ELSE
                          state_automat<= processing;
                      END IF;
                  ELSE
                      state_automat<= processing_round;
                 END IF;
             WHEN processing_round =>
                  state_automat<= round;
             WHEN processing =>
                  message_block_internal<=message_block;
                  hash_calculate_reverse(message_block_internal, Q, H);
```

```
                        roundconstant_expanded_func(roundconstant);
                        receive_ready<='1';
                        brojac<=0;
                        hash_calculate(message_block, H, Q);
                        state_automat<= round;
                        IF (last_block='1') THEN
                           flag<='1';
                        ELSE
                           flag<='0';
                        END IF;
                    WHEN finalization =>
                        get_hash<='1';
                        message_block_internal<=message_block;
                        hash_final_func(H,hash_final);
                        hashvalue<=hash_final;
                        state_automat<= init;
                        receive_ready<='0';
            END CASE;
        END IF;
    END PROCESS;

END JH_algorithm;
```

Kako je u pitanju sekvencijalni kod, on se nalazi unutar procesa. U listi osetljivosti nalazi se signal takta *clk*, što znači da se kod izvršava svaki put kada dođe do promene vrednosti signala takta. Dizajn ima mogućnost asinhronog reseta, što znači da se obrada vrši ukoliko signal za reset nije aktivan, u suprotnom se interni signali i izlazni portovi postavljaju na inicijalne vrednosti.

Signali definisani unutar arhitekture su: *state_automat*, *brojac* i *message_block_internal*.

Signal *state_automat* je tipa *final_automat* i njegova trenutna vrednost predstavlja fazu obrade u kojoj se dizajn nalazi (*init*, *round_init*, *processing_init*,*idle*, *round*, *processing_round*, *processing*, *finalization*).

Signal *brojac* je tipa *integer* i kontroliše broj iteracija (rundi) u fazama *round_init* i *round*, da bi se znao trenutak kada dizajn može preći u određeno naredno stanje.

Signal *message_block_internal* je interni signal u kome se čuva blok poruke sa ulaza, koji se koristi priliikom operacije XOR poruke sa drugom (desnom) polovinom heš promenljive, iz razloga što se u međuvremenu na ulazu pojavljuje novi blok poruke, a potreban je određeni broj taktova da se jedan blok poruke obradi. Ovaj signal se primenjuje prilikom korišćenja procedure *hash_calculate_reverse* kada je na ulazu novi blok poruke.

Signal flag je tipa STD_LOGIC i označava da se obrađuje poslednji blok poruke (kada je postavljen na aktivnu vrednost), tako da se na taj način prelazi u finalno stanje.

U deklarativnom delu procesa definisane su već pomenute varijable na početku potpoglavlja 4.3,*H*, *roundconstant*, *Q* i *hash_final*, tipa *hash_type*, *round_zero_type*, *round_type* i *STD_LOGIC_VECTOR*, respektivno, iz razloga što su za većinu procedura argumenti varijable.

Naredbe koje se izvršavaju u slučaju aktivnog signala za reset su sledeće:

```
state_automat<=init;
flag<='0';
get_hash<='0';
init_done<='0';
receive_ready<='0';
hashvalue<=(OTHERS=>'0');
```

Nakon reseta, dizajn se prebacuje u stanje *init*, tj. stanje u kojem još uvek ne stiže poruka, odnosno gde se računa inicijalna heš vrednost. Signal *hashvalue* se postavlja na nultu vrednost, kao i signali *flag*, *get_hash*, *init_done* i *recieve_ready*.

Ukoliko je signal za reset neaktivan, u zavisnosti od vrednosti signala *state_automat*, izvršava se odgovarajući kod. U tu svrhu je iskorišćena CASE struktura. Kod koji se izvršava je onaj koji se nalazi u okviru WHEN dela koji sadrži vrednost koja se poklapa sa trenutnom vrednosti signala *state_automat*.

Ukoliko se dizajn nalazi u *init* stanju, izvršava se sledeći kod:

```
init_done<='0';
get_hash<='0';
hash_initial(H,Q);
roundconstant_expanded_func(roundconstant);
state_automat<=round_init;
brojac<=0;
receive_ready<='0';
```

U ovoj fazi se heš promenljivoj *H* dodeljuje vrednost unapred definisane konstante u paketu, vrši se grupisanje bita i kao izlaz procedure *hash_initial* dobijamo promenljivu *Q*. Takođe se i ulazno-izlaznom argumentu procedure *roundconstant_expanded_func*, *roundconstant*, dodeljuje vrednost unapred definisane konstante u paketu. Signali *init_done*, *get_hash* i *receive_ready* su setovani na nultu vrednost, obzirom da incijalna heš vrednost još uvek nije izračunata, kao i da finalna heš vrednost nije u toku izračunavanja, i naposletku, još nije pristigao prvi blok poruke kada bi započela obrada, jer je još u toku izračunavanje inicijalne heš vrednosti. Takođe, obzirom da još nije pozvana *round* funkcija,*brojač* takođe ima nultu vrednost.

Ukoliko se dizajn nalazi u *round_init* stanju, izvršava se sledeći kod:

```
r8(roundconstant,Q);
update_roundconstant(roundconstant);
brojac<=brojac+1;
IF (brojac>40) THEN
   f8_reverse(Q,H);
   state_automat<=idle;
   init_done<='1';
ELSE
   state_automat<=processing_init;
   init_done<='0';
```

Ulazni argumenti *r8* procedure su *roundconstant* i *Q*. Kao izlaz ove procedure dobija se promenljiva Q. Takođe se i promenljiva *roundconstant* ažurira putem procedure *update_roundconstant*. Varijabla *brojac* se ažurira. Ukoliko još uvek nisu 42 runde obrađene, faza prelazi u *processing_init*, iz koje se ponovo vraća u *round_init* stanje:

```
state_automat<= round_init;
```

Ukoliko su 42 runde izvršene, izvršava se procedura *f8_reverse*, čiji je ulazni argument promenljiva *Q*, a izlazni promenljiva *H*. U okviru ove procedure se izvršava degrupisanje bita, kao i XOR operacija sa drugom polovinom heš promenljive. Na ovaj način je izračunata inicijalna heš vrednost, te je naredna faza *idle*, kada je dizajn spreman za primanje prvog bloka poruke. *init_done* je aktivan jer je inicijalizacija završena.

Naredna faza je faza *idle*:

```
get_hash<='0';
receive_ready<='1';
IF (first_block='1') THEN
    brojac<=0;
    message_block_internal<=message_block;
```

```
    roundconstant_expanded_func(roundconstant);
    hash_calculate(message_block, H, Q);
    state_automat<= round;
    receive_ready<='0';
    IF (last_block='1') THEN
       flag<='1';
    ELSE
       flag<='0';
    END IF;
END IF;
```

Obzirom da se finalna heš vrednost još uvek ne računa, signal *get_hash* i dalje ima nultu vrednost. Prvi blok je spreman da bude primljen i obrađen, pa je signal *receive_ready* aktivan. Ukoliko je primljen prvi blok, on se smešta u internu promenljivu *message_block_internal*, zatim se vrši resetovanje *roundconstant* promenljive i (dodeljuje joj se vrednost konstante definisane u paketu) u okviru procedure *roundconstant_expanded_func*. Zatim se poziva procedura *hash_calculate*, koja ima istu ulogu kao procedura *hash_initial*, samo što se sada operacije vrše nad stvarnim blokom poruke, datim kao ulazni argument. Kao izlaz ove procedure dobija se promenljiva *Q*. Dizajn još uvek nije spreman za prijem novog bloka poruke, te je signal *receive_ready* još uvek neaktivan. Ukoliko je prvi blok koji je primljen ujedno i poslednji, signal *flag* postaje aktivan. Naredna faza u koju se prelazi je *round* faza:

```
r8(roundconstant,Q);
update_roundconstant(roundconstant);
receive_ready<='0';
brojac<=brojac+1;
IF (brojac>40) THEN
   IF (flag='1') THEN
      hash_calculate_reverse(message_block, Q, H);
      state_automat<= finalization;
   ELSE
      automat<= processing;
   END IF;
ELSE
   state_automat<= processing_round;
END IF;
```

Ulazni argumenti *r8* procedure su *roundconstant* i *Q*. Kao izlaz ove procedure dobija se promenljiva Q. Takođe se i promenljiva *roundconstant* ažurira putem procedure *update_roundconstant*. Dizajn još uvek nije spreman za prijem novog bloka poruke, te je signal *receive_ready* još uvek neaktivan. Varijabla brojač se ažurira. Ukoliko još uvek nisu 42 runde obrađene, faza prelazi u *processing_round*, iz koje se ponovo vraća u *round_init* stanje:

```
state_automat<=round;
```

Ukoliko su 42 runde izvršene, ispituje se aktivnost signala *flag*, odnosno da li se obrada vrši nad poslednjim blokom poruke. Ukoliko je signal *flag* aktivan, izvršava se procedura *hash_calculate_reverse*, čiji je ulazni argument promenljiva *Q* kao i ulazni blok *message_block*, a izlazni promenljiva *H*. U okviru ove procedure se izvršava degrupisanje bita, kao i XOR operacija sa drugom polovinom heš promenljive. Na ovaj način je izračunata heš vrednost, te je naredna faza *finalization*, kada je dizajn spreman za generisanje finalne heš vrednosti. Ukoliko signal *flag* nije aktivan, dizajn prelazi u stanje *processing*:

```
message_block_internal<=message_block;
hash_calculate_reverse(message_block_internal, Q, H);
roundconstant_expanded_func(roundconstant);
receive_ready<='1';
brojac<=0;
roundconstant_expanded_func(roundconstant);
```

```
hash_calculate(message_block, H, Q);
state_automat<= round;
IF (last_block='1') THEN
    flag<='1';
ELSE
    flag<='0';
END IF;
```

Poslednji primljeni blok poruke se čuva u internoj promenljivoj *message_block_internal*. Zatim se procedurom *hash_calculate_reverse* čiji su ulazni argumenti *message_block_internal* i *Q*, a izlazni *H*, vrši izračunavanje finalne heš vrednosti, *H*. Takođe se vrši resetovanje *roundconstant* promenljive (i dodeljuje joj se vrednost konstante definisane u paketu) u okviru procedure *roundconstant_expanded_func*. Sada je dizajn spreman da primi novi blok poruke, te je signal *receive_ready* aktivan.. Brojač *brojac* se takođe resetuje na nultu vrednost, jer je sada potrebno obraditi 42 runde nad novim blokom poruke. Zatim se putem procedure *hash_calculate* ponovo dobija vrednost promenljive *Q*, te se dizajn ponovo vraća u fazu *round*. Takođe se ponovo ispituje da li je u pitanju poslednji blok, i, ukoliko jeste, signal *flag* se aktivira.

Poslednja faza izvršavanja je *finalization*:

```
get_hash<='1';
message_block_internal<=message_block;
hash_final_func(H,hash_final);
hashvalue<=hash_final;
state_automat<= init;
receive_ready<='0';
```

Sada je dizajn spreman da generiše finalnu heš vrednost, te signal *get_hash* postaje aktivan. Poslednji blok poruke se čuva u internom signalu, ukoliko naredna poruka stigne na ulaz. Procedura *hash_final_func* odseca poslednje bite (onoliko koliko zahteva verzija algoritma). Kao ulazni argument je poslednja izračunata heš funkcija *H*, a kao izlazni je *hash_final*. Promenljiva *hash_final* se smešta u signal *hashvalue*, koji predstavlja izlaz. Obzirom da je obrada završena, nijedan blok u okviru ove poruke više ne postoji da bi bio primljen, te je signal *receive_ready* neaktivan. Dizajn se vraća u *init* stanje i time je spreman za prijem naredne poruke, kada će se signal *receive_ready* ponovo aktivirati u *idle* stanju.

4.4. Razlike u kodu za ostale dužine heš vrednosti

Kako postoje određene razlike u koracima algoritma između JH-224, JH-256, JH-384 i JH-512, postoje i razlike u VHDL kodu.

Princip određivanja inicijalne heš vrednosti je identičan. Razlika je u vrednosti prva dva bajta heš promenljive, *H*, odnosno vrednosti su: 0*x*00*E*0, 0*x*0100, 0*x*0180, 0*x*0200 za JH-224, JH-256, JH-384 i JH-512, respektivno. Ove konstante su definisane u paketu:

- JH-224: `CONSTANT hashbitlen:STD_LOGIC_VECTOR(15 DOWNTO 0):= X"00E0";`
- JH-256: `CONSTANT hashbitlen:STD_LOGIC_VECTOR(15 DOWNTO 0):= X"0100";`
- JH-384: `CONSTANT hashbitlen:STD_LOGIC_VECTOR(15 DOWNTO 0):= X"0180";`
- JH-512: `CONSTANT hashbitlen:STD_LOGIC_VECTOR(15 DOWNTO 0):= X"0200";`

Takođe, razlika je i u dužini izlaza finalne heš vrednosti:

- JH-224:
  ```
  hashvalue:OUT STD_LOGIC_VECTOR(223 DOWNTO 0)
  hash_final:STD_LOGIC_VECTOR(223 DOWNTO 0);
  ```
- JH-256:

```
hashvalue:OUT STD_LOGIC_VECTOR(255 DOWNTO 0)
hash_final:STD_LOGIC_VECTOR(255 DOWNTO 0);
```

- JH-384:

```
hashvalue:OUT STD_LOGIC_VECTOR(383 DOWNTO 0)
hash_final:STD_LOGIC_VECTOR(383 DOWNTO 0);
```

- JH-512:

```
hashvalue:OUT STD_LOGIC_VECTOR(511 DOWNTO 0)
hash_final:STD_LOGIC_VECTOR(511 DOWNTO 0);
```

Usled ovih razlika ažurira se i se i procedura *hash_final_func*:

- JH-224:

```
hash_final(223 DOWNTO 216):=H(100)(7 DOWNTO 0);
hash_final(215 DOWNTO 208):=H(101)(7 DOWNTO 0);
                    ...
hash_final(7 DOWNTO 0):=H(127)(7 DOWNTO 0);
```

- JH-256:

```
hash_final(255 DOWNTO 248):=H(96)(7 DOWNTO 0);
hash_final(247 DOWNTO 240):=H(97)(7 DOWNTO 0);
                    ...
hash_final(7 DOWNTO 0):=H(127)(7 DOWNTO 0);
```

- JH-384:

```
hash_final(383 DOWNTO 376):=H(80)(7 DOWNTO 0);
hash_final(375 DOWNTO 368):=H(81)(7 DOWNTO 0);
                    ...
hash_final(7 DOWNTO 0):=H(127)(7 DOWNTO 0);
```

- JH-512:

```
hash_final(511 DOWNTO 504):=H(64)(7 DOWNTO 0);
hash_final(503 DOWNTO 496):=H(65)(7 DOWNTO 0);
                    ...
hash_final(7 DOWNTO 0):=H(127)(7 DOWNTO 0);
```

Iz priloženog se može zaključiti da nije potrebno značajno menjati dizajn kako bi se ostvarila podrška za drugačiju dužinu rezultujućeg heša.

5. OPIS PERFORMANSI I VERIFIKACIJA DIZAJNA

5.1. Opis performansi

Izvršavanjem procesa analize i sinteze u slučaju sve četiri dužine rezultujućeg heša, dobijene su informacije o performansama svake od implementacija. Proces analize i sinteze izvršen je u ISE razvojnom okruženju za FPGA čipove proizvođača Xilinx. U pitanju je dobra procena vrednosti i one su date u tabeli 5.1.1. Obzirom da dizajn podrazumeva veći broj pinova, za dužine 224 i 256 izabran je uređaj XC6VLX550T familije Virtex6, dok je za dužine 384 i 512 izabran uređaj XC6VLX760 isto familije Virtex6.

	Vrednost			
Naziv	*JH-224*	*JH-256*	*JH-384*	*JH-512*
Broj slajs registara	3079/687360 (0%)	3111/687360 (0%)	3239/948480 (0%)	3367/948480 (0%)
Broj slajs LUT-ova	5249/343680 (1%)	5250/343680 (1%)	5248/474240 (1%)	5250/474240 (1%)
Broj potpuno iskorišćenih parova LUT-FF	2345/5983 (39%)	2345/6016 (38%)	2345/6142 (38%)	2345/6272 (37%)
Broj globalnih taktova (BUFG/BUFGCTRL)	1/32 (3%)	1/32 (3%)	1/32 (3%)	1/32 (3%)
Broj pinova	743/840 (88%)	775/840 (92%)	903/1200 (75%)	1031/1200 (86%)
Maksimalna frekvencija	357.949 MHz	357.949 MHz	357.949 MHz	357.949 MHz

Tabela 5.1.1 Procenjene performanse za različite dužine izlaza

Prema podacima iz Tabele 5.1.1. može se primetiti da se broj pinova na čipu menja sa povećanjem veličine (odnosno broja bita) izlaza. Na primer, verzija JH-224 ima 744 pinova, dok JH-512 ima 1032 pina, što je posledica većeg rezultujućeg heša. Dodavanjem dodatnih blokova koji bi vršili konverziju paralela u seriju na izlazu, i konverziju serije u paralelu na ulazu, mogao bi se smanjiti broj potrebnih pinova. Takođe, broj registara se povećava sa povećanjem dužine heša jer se registri koriste za čuvanje stanja.

Za sve četiri implementacije je korišćen jedan globalni takt clk.

Maksimalna frekvencija je ista za bilo koju dužinu izlaza. Na ovaj način je pokazano da se troše skromi resursi, što je dobar svedok da je implementacija JH dizajna veoma efikasna.

5.2. Verifikacija dizajna

U ovom potpoglavlju će biti opisana verifikacija procedura kao i celokupnog dizajna JH algoritma. Testiraće se poklapanje vrednosti dobijenih nakon pokretanja funkcionalne simulacije u okviru ISim simulatora sa vrednostima u referentnom fajlu. Referentni fajlovi za sve dužine izlaza su preuzeti sa zvaničnog sajta instituta NIST i kompletni fajlovi će biti priložen na CD-u pod imenom *ShortMsgKAT_224.txt, ShortMsgKAT_256.txt, ShortMsgKAT_384.txt* i *ShortMsgKAT_512.txt*.

Verifikacija dizajna biće prikazana za **JH-224** verziju algoritma. Padding procedura je obavljena ručno.

Verifikacija dizajna je prikazana za četiri slučaja:

- Kada poruku čini jedan blok.
- Kada poruku čine dva bloka.
- Kada poruku čine tri bloka.
- Kada se vrši heširanje uzasopne dve poruke.

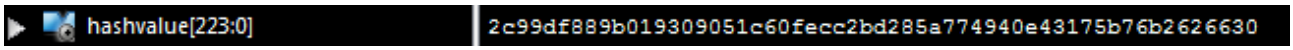
5.2.1. Verifikacija dizajna poruke sa jednim blokom

Poruka čiji se heš traži je dužine 0 bita, tako da će postojati samo jedan blok. U priloženom tekstualnom fajlu je uzet slučaj:

Len = 0 / Msg = *00*

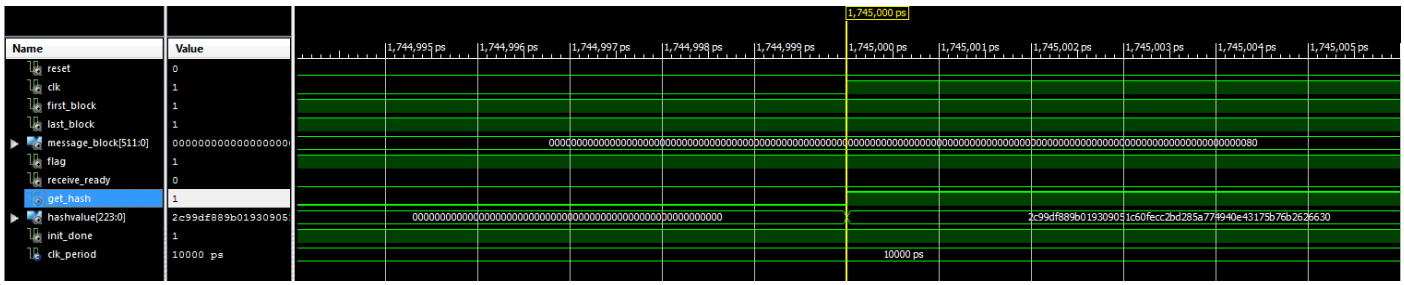
MD = *2C99DF889B019309051C60FECC2BD285A774940E43175B76B2626630*

Pokretanjem simulacije dobija se vrednost koja je priložena u tekstualnom fajlu, prikazana na slici 5.2.1.1:



Slika 5.2.1.1. Promenljiva koja predstavlja heš vrednost

Vreme čekanja da se inicijalizacija obradi je 900ns. Signal *get_hash* postaje aktivan u 1.745 ns, odnosno u tom trenutku dobijamo heširanu vrednost, kao na slici 5.2.1.2:



Slika 5.2.1.2. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna

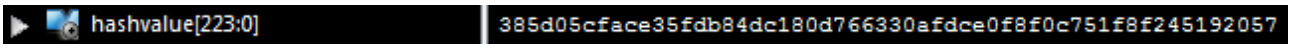
5.2.2. Verifikacija dizajna poruke od dva bloka

Poruka čiji se heš traži je dužine 24 bita, tako da će postojati dva bloka. U priloženom tekstualnom fajlu je uzet slučaj:

Len = 24 / Msg = *1F877C*

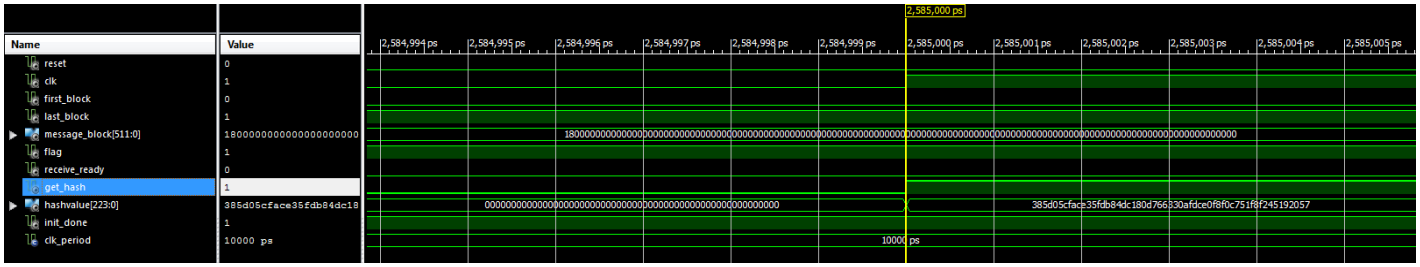
MD = *385D05CFACE35FDB84DC180D766330AFDCE0F8F0C751F8F245192057*

Pokretanjem simulacije dobija se vrednost koja je priložena u tekstualnom fajlu, prikazana na slici 5.2.2.1:



Slika 5.2.2.1. Promenljiva koja predstavlja heš vrednost

Vreme čekanja da se inicijalizacija obradi je 900ns. Signal *get_hash* postaje aktivan u 2.585 ns, odnosno u tom trenutku dobijamo heširanu vrednost, kao na slici5.2.2.2:



Slika 5.2.2.2. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna

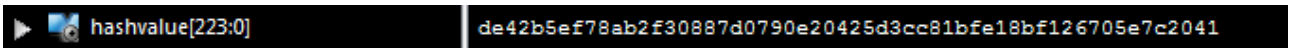
5.2.3. Verifikacija dizajna poruke od tri bloka

Poruka čiji se heš traži je dužine 576 bita, tako da će postojati tri bloka. U priloženom tekstualnom fajlu je uzet slučaj:

Len = 576 / Msg = *1EED9CBA179A009EC2EC5508773DD305477CA117E6D569E66B5F64C6BC64801CE25A8424CE4A26D575B8A6FB10EAD3FD1992EDDDEEC2EBE7150DC98F63ADC3237EF57B91397AA8A7*

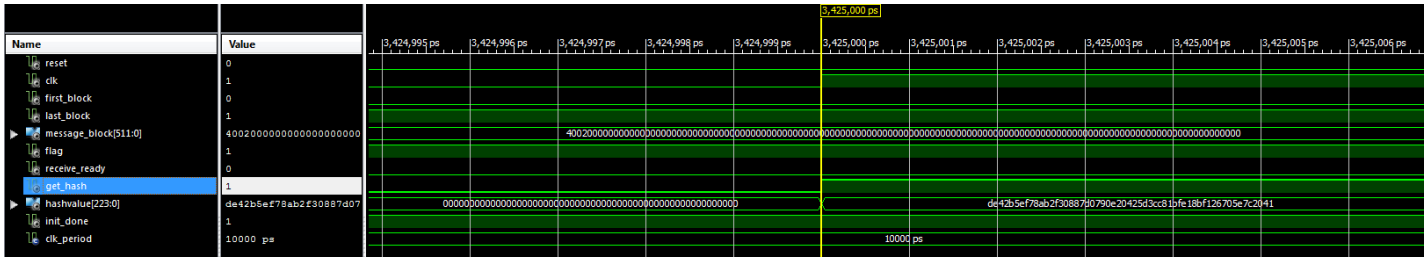
MD = *DE42B5EF78AB2F30887D0790E20425D3CC81BFE18BF126705E7C2041*

Pokretanjem simulacije dobija se vrednost koja je priložena u tekstualnom fajlu, prikazana na slici 5.2.3.1:



Slika 5.2.3.1. Promenljiva koja predstavlja heš vrednost

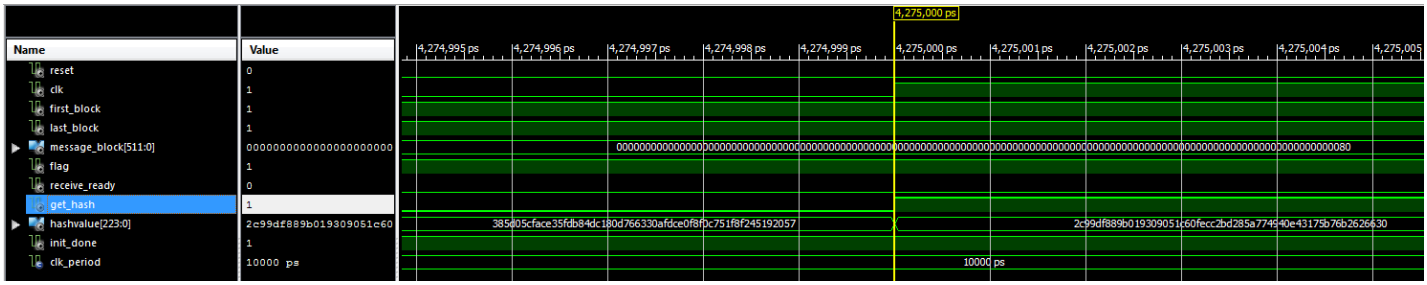
Vreme čekanja da se inicijalizacija obradi je 900ns. Signal *get_hash* postaje aktivan u 3.425 ns, odnosno u tom trenutku dobijamo heširanu vrednost, kao na slici5.2.3.2:



Slika 5.2.3.2. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna

5.2.4. Verifikacija dizajna dve pristigle poruke

Za testiranje dve poruke, koje se heširaju sukcesivno jedna za drugom, uzeti su gorepomenuti slučajevi dužina od 24 i 0 bita. Na izlazu se prvo treba pojaviti heširana vrednost za poruku dužine 24 bita, a zatim za poruku dužine 0 bita. Vreme čekanja da se inicijalizacija obradi je 900ns. Vreme čekanja između dve poruke je postavljeno na 1000ns. Signal *get_hash* postaje aktivan u 2.585 ns, odnosno u tom trenutku dobijamo heširanu vrednost za dužinu poruke od 24 bita. Zatim ponovo postaje aktivan u trenutku 4.275ns kada se dobija heš vrednost za poruku dužine 0 bita, kao na slici5.2.4.1:



Slika 5.2.4.1. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna

Uspešno je testirano ponašanje dizajna i u slučaju drugih ulaznih podataka (poruka), kao i ponašanje dizajna za različite dužine heš izlaza. Zbog analogije u principu testiranja, ali i da bi se izbegle redudantnosti u tekstu teze, prikazana je verifikacija za samo jednu dužinu heš izlaza.

6. ZAKLJUČAK

Heš funkcije predstavljaju moćan alat koji je u širokoj upotrebi u modernom računarstvu, i imaju značajnu ulogu u mnogim kriptografskim sistemima, gde se izdvajaju najpoznatiji slučajevi primene kod digitalnog potpisa, provere autentičnosti dokumenata i pseudo-slučajnih generatora brojeva.

Ova činjenica postaje jasna ako se ima u vidu u kojim se sve sferama koristi kriptografija: vojnim aktivnostima, bankarskim transakcijama, u zaštiti i prenosu računarskih podataka, telekomunikacijama, državnim aktivnostima itd. Razvojem savremenih telekomunikacija, elektronike i računarske tehnologije potrebe za dobrim kriptografskim sistemima su višestruko uvećane. Jer„slom" kriptografskog sistema koji se koristi u nekoj od nabrojanih aktivnosti može doneti ogromnu štetu (ekonomsku, političku, vojnu). Stoga se svake godine objavljuje veliki broj publikacija i kriptografskih patenata, ali i standarda i hardverskih implementacija.

JH algoritam može efikasno da se implementira na različitim platformama, počevši od hardvera do 128/256-bitnih instrukcija za procesore. Razlog za to je što se, zahvaljujući dizajnu koji se bazira na AES metodologiji, mogu koristiti veoma jednostavni elementi. S-kutija se može implementirati sa 20 različitih binarnih operacija, linearna transformacija sa 10, a čak 256 s-kutija se može procesirati u paraleli. Hardverska implementacija je jednostavna zahvaljujući jednostavnosti s-kutija i linearnoj transformaciji. JH koristi 1024 bita da uskladišti podatke koji se obrađuju procedurom *E8*(grupisanje, *R8*, degrupisanje), 512 bita da uskladišti blok poruke, i 256 bita da uskladišti round konstantu.

Jedan od problema sa kojima se autor ovog rada susreo su poteškoće pri pronalaženju odgovarajućeg čipa koji bi bio u stanju da podrži dizajn. Problem je broj pinova koji je potreban. Da bi se ovaj problem prevazišao potrebno je dodati ulazne i izlazne blokove koji bi vršili konverziju serija u paralelu i obrnuto. Resursi nisu kritični što se moglo videti iz tabele 5.1.1., obzirom da se veoma mali procenat resursa čipa troši.

Razlozi za korišćenje JH algoritma su u tome što je jednostavan za implementaciju, siguran je, ima jednostavan dizajn, znatno je otporan na kolizije, itd.

LITERATURA

[1] Wikipedia: Kriptografija. Preuzeto sa: http://sh.wikipedia.org/wiki/Kriptografija

[2] B. Pajčin, P. Ivaniš, „Softverska realizacija sistema za digitalno potpisivanje sa heš funkcijama i RSA algoritmom“, Mart 2011.

[3] M. Kovinić, „Uvod u kriptografiju i infrastrukturu javnih ključeva“, 2010.

[4] Wu Hongjun, „The Hash Function JH“, Jan. 2011.