ELEKTROTEHNIČKI FAKULTET UNIVERZITETA U BEOGRADU

HARDVERSKA IMPLEMENTACIJA *CUBEHASH* ALGORITMA ZA HEŠIRANJE

– Master rad –

Kandidat:

Maja Bijelić 2012/3224

Mentor:

doc. dr Zoran Čiča

Beograd, Septembar 2014.

SADRŽAJ

1. UVOD

U današnje vreme, kada je komunikacija digitalnim putem sve više prisutna, sigurnost komunikacije, kao i učesnika u njoj, je od ključne važnosti. Vlade, vojska, firme, bolnice, kao i mnoge druge institucije poseduju veliki broj poverljivih podataka o svojim zaposlenima, klijentima, proizvodima, istraživanjima i finansijama. Većina tih informacija se, danas, skladišti i obrađuje na računarima i šalje putem mreže drugim uređajima u mreži. Kada bi neke od tih informacija završile u rukama hakera ili konkurentske firme, posao i korisnici bi pretrpeli znatne gubitke. Zaštita poverljivih informacija je zato postala prioritet pa čak i obaveza svake institucije. Stoga su razvijeni različiti mehanizmi zaštite od zlonamernih napada [4]. Neki od njih su zasnovani na heš algoritmima koji su svoju primenu među mnogim oblastima našli i u kriptografiji.

Ovaj rad se bavi implementacijom *CubeHash* algoritma za heširanje. Ovaj algoritam je učestvovao na konkursu za izbor novog standarda *SHA-3* (*Secure Hash Algorithm-3*). Programski jezik korišćen u implementaciji algoritma je *VHDL* (*VHSIC Hardware Description Language*). Korišćeno je razvojno okruženje *ISE* proizvođača *Xilinx*. Glavni projekat, kao i test projekti namenjeni verifikaciji dizajna priloženi su na pratećem CD-u.

Pored hardverske implementacije datog algoritma, rezultat rada su i verifikacija dizajna, kao i analiza performansi za različite vrednosti parametara.

Ostatak rada je organizovan na sledeći način: Drugo poglavlje daje osnovne informacije o heš algoritmima i njihovoj kriptografskoj primeni, a takođe opisuje osnove CubeHash algoritma. Treće poglavlje detaljno opisuje hardversku implementaciju *CubeHash* 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 *CubeHash* heširanje. Četvrto poglavlje se bavi analizom performansi realizovanog *CubeHash* algoritma u zavisnosti od izabranih parametara. Drugi deo četvrtog poglavlja se bavi verifikacijom dizajna poređenjem dobijenih vrednosti nakon simulacije i referentnih vrednosti koje je objavio autor *CubeHash* algoritma. Poslednje, peto, poglavlje sadrži zaključna razmetranja autora ove teze o implementaciji *CubeHash* algoritma, kao i predloge za buduću optimizaciju realizovane implementacije.

2. HEŠ ALGORITMI

2.1. ISTORIJA, DEFINICIJA I OSOBINE

Pojam heš algoritma je u literaturi izjednačen sa heš funkcijom. Prvi dizajn kriptografske heš funkcije datira sa kraja 70-ih godina 20. veka (*Diffie* i *Hellman*). Dvadeset godina nakon toga već je postojao veliki broj različitih heš funkcija. Vremenom je otkriveno da je veliki broj tih heš funkcija imao sigurnosne propuste. MD5 (*Message Digest 5*) i SHA-1 (*Secure Hash Algorithm 1*) heš funkcije su našle primenu u velikom broju aplikacija i time su zaradile ime „švajcarski nož" kriptografije [2]. 2004. godine diferencijalnom kriptoanalizom omogućeno je pronalaženje velikog broja kolizija kod MD5. S obzirom da je zasnovana na istim principima kao i MD5, pouzdanost SHA-1 je, takođe, bila dovedena u pitanje. SHA-2 algoritam je razvijen kako bi zamenio SHA-1. Nakon toga započeo je rad na definisanju novog (SHA-3) standarda. NIST (*National Institute of Standards and Technology*) je 2007. godine objavio da će organizovati takmičenje za izbor novog SHA-3 heš algoritma. Od 64 prijavljena kandidata četrnaestoro ih je prošlo u drugu rundu takmičenja među kojima je bio i *CubeHash* algoritam realizovan u okviru ove teze.

Zahtevi koje svaka heš funkcija mora da ispuni su:

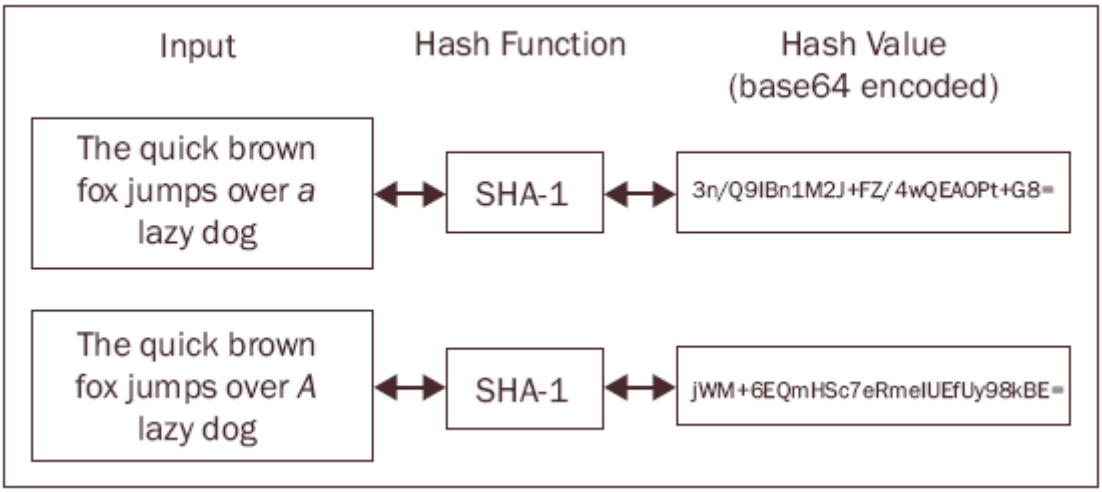
- jednostavno izračunavanje heša za bilo koju poruku
- izlaz konstantne dužine za svaki ulaz proizvoljne dužine
- skoro je nemoguće rekonstruisati ulaz na osnovu poznatog heša
- skoro je nemoguće pronaći dva ulaza koji daju isti heš

Sigurnost današnjih komunikacija se dobrim delom zasniva na kriptografiji gde se u velikoj meri primenjuju heš funkcije. One su našle primenu u aplikacijama kao što su digitalni potpis i pseudoslučajni generatori brojeva. Da bi se heš funkcija (kriptografska heš funkcija) koristila u kriptografiji, neophodno je da zadovolji rigorozne sigurnosne kriterijume. Stvaranje takvih funkcija je izuzetno težak zadatak jer su od mnoštva ponuđenih konstrukcija samo retke (sigurnost MD5 i SHA-1 algoritama je narušena, a SHA-2 heš algoritam još uvek nije probijen) preživele test vremena i pokazale zadovoljavajuće osobine. Zbog toga postoji stalna potreba za „otvorenim konkursom" za stvaranje heš funkcija sa boljim performansama od postojećih [6].

Heš funkcija mapira ulaz proizvoljne dužine u kratki izlaz fiksne dužine *m*. Taj izlaz se naziva *heš vrednost*, *heš kod* ili *heš*.

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

Kao što vidimo, ulaz je binarni string proizvoljne dužine, dok je izlaz takođe binarni string, ali fiksne dužine *m* [6].



Slika 2.1.1. Princip rada heš funkcija [10]

Na Slici 2.1.1. se može videti kako promena u samo jednom karakteru drastično menja vrednost heša. Ova pojava se naziva *efekat lavine* (*avalanche effect*). Sa Slike 2.1.1. se na primeru može videti da male promene ulaza izazivaju značajne, nepredvidive promene na izlazu.

2.2. PRIMENA HEŠ FUNKCIJA

Heš funkcije su pronašle primenu u mnogim oblastima. Koriste se za:

- Heš tabele: omogućuju brzo pretraživanje podataka bez obzira na veličinu tabele. Heš funkcija daje na izlazu indeks traženog podatka u tabeli.
- Pronalaženje dupliranih zapisa: svaki podatak se propusti kroz heš funkciju i pronađu se podaci koji imaju istu vrednost heša.
- Upotreba u kriptografiji.

Kriptografske heš funkcije se primenjuju u savremenim telekomunikacijama gde je zaštita podataka od zlonamernih napadača od ključne važnosti. Cilj je na neki način osigurati komunikaciju učesnika ako se zna da napadač može imati pristup podacima koji se razmenjuju tokom komunikacije. Neke od primena heš funkcija u kriptografiji su:

- Provera integriteta poruke: poređenjem heša primljene poruke sa izračunatim hešom primljene poruke može se utvrditi da li je poruka izmenjena tokom prenosa.
- Potvrda lozinke: skladištenje korisničkih lozinki u svom izvornom obliku nije preporučljivo jer bi napadač koji otvori fajl imao na raspolaganju sve korisničke šifre. Zbog toga se čuvaju heševi korisničkih lozinki i korisnik se autentifikuje tako što se heš njegove šifre poredi sa snimljenim hešom.
- Digitalni potpis: koristi se kao potvrda identiteta pošiljaoca poruke. Pošiljalac hešira poruku i šifruje heš svojim tajnim ključem. Primalac izvuče heš iz primljene poruke tako što ga dešifruje javnim ključem. Zatim hešira poruku i uporedi dobijeni heš sa onim dobijenim pomoću javnog ključa.
- Izvođenje novih ključeva za šifrovanje iz jednog sigurnog ključa.

2.3. CUBEHASH ALGORITAM

CubeHash algoritam je bio jedan od kandidata za izbor SHA-3 algoritma na konkursu koji je organizovao NIST. Autor ovog algoritma je *Daniel J. Bernstein,* univerzitetski profesor matematike i računarskih nauka. Prva verzija algoritma je predložena 2008. godine da bi konačan oblik algoritam dobio 2010. godine. Jedina razlika između različitih verzija ovog algoritma je u vrednostima parametara. U ovom radu će se govoriti o hardverskoj implementaciji *CubeHash* algoritma. U nastavku ovog potpoglavlja biće opisan njegov princip rada.

CubeHash r/b-h proizvodi heš dužine *h* bita nezavisno od dužine poruke na ulazu. Heš zavisi od dva podesiva parametra *b* i *r*, čiji izbor omogućuje kompromis između sigurnosti i performansi. Parametri *b* i *r* će biti objašnjeni dalje u ovom poglavlju. Dužina poruke na ulazu može biti od 0 do 2^{128}-1 bita.

Postoji i varijanta *CubeHash* algoritma koja se označava sa *CubeHash i+r/b+f-h*. Sledi objašnjenje parametara:

i - broj rundi inicijalizacije

r - broj rundi

b - dužina jednog bloka u bajtovima

f - broj rundi finalizacije

h - dužina heša u bitima

Algoritam se sastoji od nekoliko bitnih koraka [7]:

- inicijalizacija početnog stanja od 1024 bita
- poruka se dopunjuje do celobrojnog umnoška blokova dužine *b* bajtova
- vrši se logička *xor* operacija između bloka i prvih *b* bajtova stanja i dobijeno stanje se propušta kroz *r* rundi transformacije
- stanje se finalizuje
- prvih *h* bita stanja čine traženi heš

Inicijalizacija funkcioniše na sledeći način: Stanje od 1024 bita se posmatra kao niz od 32 reči od kojih je svaka dužine 32 bita. Reči se obeležavaju kao x_{00000}, x_{00001}, x_{00010}, ..., x_{11110}, x_{11111} i svaka je predstavljena kao 32-bitni ceo broj u *little-endian* formatu. Prve tri reči stanja (x_{00000}, x_{00001} i x_{00010}) se postavljaju na vrednosti *h*/8, *b* i *r*, respektivno. Ostale reči u stanju dobijaju vrednost „sve nule“. Stanje se, zatim, propušta kroz 10*r* identičnih rundi (u *CubeHash i+r/b+f-h* implementaciji se propušta kroz *i* rundi). Kao rezultat ovih operacija dobija se vektor inicijalizacije.

Poruka se dopunjuje tako što se izvornoj poruci doda bit '1' i posle njega onoliko '0' bita koliko je potrebno da dopunjena poruka dobije dužinu celobrojnog umnoška *b* okteta.

Finalizacija: Izvršava se logička *xor* operacija između broja 1 (ne bita '1') i zadnje reči stanja x_{11111}. Stanje se, zatim, propušta kroz 10*r* rundi (u *CubeHash i+r/b+f-h* implementaciji se propušta kroz *f* rundi).

Svaka runda sadrži sledećih deset koraka [7]:

1) Dodati x_{0jklm} na x_{1jklm} po modulu 2^{32}, $\forall(j,k,l,m)$,
2) Rotirati x_{0jklm} ulevo za 7 bita, $\forall(j,k,l,m)$,
3) Zameniti x_{00klm} sa x_{01klm} i obratno, $\forall(k,l,m)$,
4) Rezultat x*or* operacije između x_{1jklm} i x_{0jklm} dodeliti x_{0jklm}, $\forall(j,k,l,m)$,
5) Zameniti x_{1jk0m} sa x_{1jk1m} i obratno, $\forall(j,k,m)$,
6) Dodati x_{0jklm} na x_{1jklm} po modulu 2^{32}, $\forall(j,k,l,m)$,
7) Rotirati x_{0jklm} ulevo za 11 bita, $\forall(j,k,l,m)$,
8) Zameniti x_{0j0lm} sa x_{0j1lm} i obratno, $\forall(j,l,m)$,
9) Rezultat x*or* operacije između x_{1jklm} i x_{0jklm} dodeliti x_{0jklm}, $\forall(j,k,l,m)$,
10) Zameniti x_{1jkl0} sa x_{1jkl1} i obratno, $\forall(j,k,l)$.

3. IMPLEMENTACIJA *CUBEHASH* ALGORITMA

U ovom poglavlju biće opisan programski kod korišćen za hardversku implementaciju *CubeHash* algoritma. Pretpostavljeno je da je izvršen *padding* (dopuna poruke do celobrojnog umnoška *b* bajtova). Od prvog predloga ovog algoritma pa do sada, struktura algoritma se nije menjala. Jedine promene su bile u vrednostima parametara (*i*, *f*, *b*, *r* i *h*). Stoga je dizajn parametrizovan i korisniku je ostavljeno da odabere vrednosti parametara koje odgovaraju njegovim potrebama. U narednim potpoglavljima biće opisani interfejsi dizajna, stanja u kojima se dizajn može naći, tipovi promenljivih, korišćene procedure, kao i unutrašnjost crne kutije.

3.1. INTERFEJSI

U kodu se može videti da dizajn sadrži ulazne i izlazne interfejse. Ulazni interfejsi su *clk*, *reset*, *prvi*, *zadnji* i *blok*, a izlazni je *izlaz*. Sledi opis ovih interfejsa:

Signal *clk* se koristi kao signal takta. Uzlazna ivica signala takta se koristi da označi početak izvršavanja naredbi u kodu.

Signal *reset* se koristi onda kada je potrebno zaustaviti izvršavanje koda u iščekivanju poruke koja treba da se obradi.

Signal *prvi* se koristi da označi da je stigao prvi blok poruke čiji se heš računa.

Signalom *zadnji* se označava zadnji blok poruke. U slučaju da se poruka sastoji od samo jednog bloka, signali *prvi* i *zadnji* će biti aktivni istovremeno.

Sadržaj signala *blok* su biti jednog bloka poruke čiji heš treba da se izračuna.

Izlazni signal *izlaz* sadrži heš date poruke.

3.2. UNUTRAŠNJOST CRNE KUTIJE

U ovom potpoglavlju će biti opisani korišćeni tipovi promenljivih kao i konačni automat. Tipovi promenljivih su korisnički definisani tipovi podataka i generisani su sa ciljem da olakšaju implementaciju algoritma.

3.2.1. Tipovi promenljivih

Za potrebe dizajna, pored standardnih tipova podataka, korišćeni su i korisnički definisani tipovi podataka. Dati tipovi su definisani u paketu *paket_const.vhd* koji je priložen na CD-u. Sledi opis tih tipova:

- *stanje* - Ovaj tip je osnova *CubeHash* algoritma. Tip *stanje* se transformiše kroz runde i prvih *h* bita na kraju svih transformacija daju heš poruke. To je niz od 32 reči od kojih svaka ima dužinu od 32 bita.
- *konacni_automat* - Ovo je enumerisani tip koji sadrži logičke nazive stanja automata. Ovaj tip će biti detaljnije opisan u sledećem odeljku.

3.2.2. Konačni automat

Promenljiva tipa *konacni_automat* može imati jednu od tri vrednosti:

- *idle*
- *obrada*
- *finalizacija*

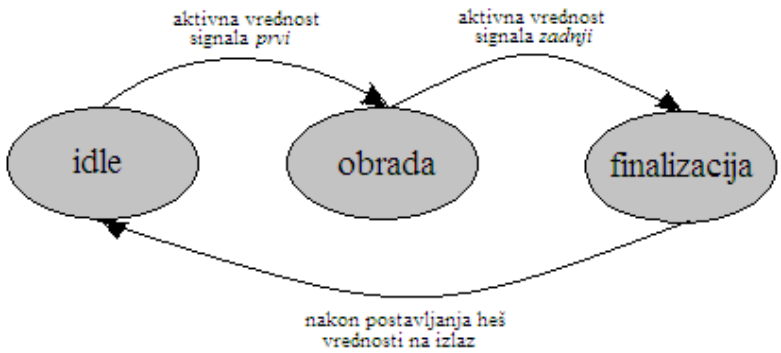
Ova tri stanja označavaju različite faze obrade poruke.

Stanje *idle* je neaktivno stanje. U ovom stanju se algoritam nalazi onda kada čeka prijem poruke. Takođe, u ovom stanju se vrši generisanje vektora inicijalizacije. To znači da se ovaj vektor može unapred generisati tako da ne usporava dizajn svojim računanjem nakon prijema prvog bloka poruke.

Dizajn prelazi u stanje *obrada* u trenutku kada primi prvi blok poruke. U ovom stanju ostaje do prijema zadnjeg bloka poruke.

Stanje *finalizacija* traje jedan takt i označava izvršavanje *f* rundi finalizacije nakon čega se dobija heš.

Na Slici 3.2.2.1. je prikazan skup stanja konačnog automata kao i vrednosti signala koje su potrebne da bi se iz jednog stanja prešlo u drugo.



Slika 3.2.2.1. Stanja konačnog automata i uslovi prelaska između stanja

3.3. OPIS ALGORITMA

U ovom potpoglavlju će prvo biti navedene i objašnjene procedure korišćene u kodu. Nakon toga će biti objašnjen i kod koji prati tok obrade poruke, kao i uloga pre toga objašnjenih procedura. Procedure i konstante korišćene u realizaciji ovog dizajna definisane su u već pomenutom *paket_const.vhd* paketu.

3.3.1. Opis procedura

U ovom odeljku će biti objašnjene procedure koje će se koristiti u kodu.

Procedura *init_napravi* vrši generisanje vektora inicijalizacije. Ova procedura ima samo izlazni argument *init*. Izlaz je tipa *stanje*. Ovaj tip se (kao što je već objašnjeno) sastoji od 32 reči od po 32 bita dužine. Vektor inicijalizacije se dobija tako što se prve tri reči stanja (promenljive *pomocna*) postave na vrednosti *h*/8, *b* i *r*, respektivno. Ostale reči dobijaju vrednost „sve nule“. Funkcija *conv_std_logic_vector* pretvara celobrojne vrednosti *h*/8, *b* i *r* u niz bita tipa *std_logic_vector*. Prvi argument ove funkcije je ceo broj koji se pretvara u niz bita, a drugi dužina tog niza. Pošto ova funkcija predstavlja *integer* u *big-endian* formatu, korisno je bilo napraviti proceduru *rotiraj* koja pretvara *big-endian* zapis u *little-endian*. Ova procedura će biti opisana nakon procedure *init_napravi*. Po zahtevu algoritma, kada se dodele vrednosti prvim rečima u stanju, stanje treba transformisati kroz *i* rundi inicijalizacije. To radi procedura *runda* koja će biti objašnjena kasnije u ovom odeljku.

```
PROCEDURE init_napravi (VARIABLE init: OUT stanje) IS
    VARIABLE pomocna:stanje;
BEGIN
    pomocna:=(OTHERS=>"00000000000000000000000000000000");
    pomocna(0):=conv_std_logic_vector(h/8,32);
    rotiraj(pomocna(0),pomocna(0));
    pomocna(1):=conv_std_logic_vector(b,32);
    rotiraj(pomocna(1),pomocna(1));
    pomocna(2):=conv_std_logic_vector(r,32);
    rotiraj(pomocna(2),pomocna(2));
    FOR j IN 0 TO i-1 LOOP
        runda(pomocna, pomocna);
    END LOOP;
    init:=pomocna;
END init_napravi;
```

Procedura *rotiraj* ima ulazni i izlazni argument. Ulazni argument je *std_logic_vector* niz dužine 32 bita koji je obično u *big-endian* formatu. Razlika između *big-endian* i *little-endian* formata je u poretku bajtova u nizu. U *big-endian* formatu bajt najveće težine se nalazi levo u nizu, dok se u *little-endian* formatu on nalazi krajnje desno. Poredak bita u bajtu je isti za oba formata (bit najveće težine je levo, a bit najmanje težine desno). Izlaz procedure je dati niz u *little-endian* poretku. Procedura funkcioniše i u slučaju kad se od *little-endian* poretka želi dobiti *big-endian*. U priloženom kodu procedure se vidi da procedura menja poredak bajtova u reči dužine 4 bajta tako da prvi bajt postaje četvrti, drugi postaje treći, itd.

```
PROCEDURE rotiraj(VARIABLE rotiraj_ulaz: IN STD_LOGIC_VECTOR(31 DOWNTO 0);
VARIABLE rotiraj_izlaz: OUT STD_LOGIC_VECTOR(31 DOWNTO 0)) IS
    VARIABLE pomocna: STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
    pomocna:=rotiraj_ulaz;
    rotiraj_izlaz(31 DOWNTO 24):=pomocna(7 DOWNTO 0);
    rotiraj_izlaz(23 DOWNTO 16):=pomocna(15 DOWNTO 8);
    rotiraj_izlaz(15 DOWNTO 8):=pomocna(23 DOWNTO 16);
    rotiraj_izlaz(7 DOWNTO 0):=pomocna(31 DOWNTO 24);
END rotiraj;
```

Kako bi se bolje razumele procedure koje opisuju runde transformacije, potrebno je objasniti kako funkcioniše procedura *rotiraj_stanje*. Ova procedura menja poredak bajtova u svakoj reči stanja. U priloženom kodu se vidi da se ova procedura sastoji od primene prethodno opisane procedure *rotiraj* na svaku reč u stanju. Tokom pisanja koda za ovaj algoritam javila se potreba za pretvaranjem *little-endian* formata u *big-endian* na nivou celog stanja, kako bi bilo jednostavnije vršiti operacije nad rečima u stanju. Ova funkcija će se koristiti u okviru većine procedura.

```
PROCEDURE rotiraj_stanje(VARIABLE stanje_ulaz: IN stanje; VARIABLE stanje_izlaz:
OUT stanje) IS
    VARIABLE pomocna:stanje;
    VARIABLE j: INTEGER RANGE 0 TO 31;
BEGIN
    pomocna:=stanje_ulaz;
    FOR j IN 0 TO 31 LOOP
        rotiraj(pomocna(j),stanje_izlaz(j));
    END LOOP;
END rotiraj_stanje;
```

Runda 1 je opisana procedurom *r1*. Po zahtevu algoritma, u ovoj rundi se reči x_{1jklm} dodaje reč x_{0jklm} po modulu 2^{32}.

```
PROCEDURE r1(VARIABLE r1_ulaz: IN stanje; VARIABLE r1_izlaz: OUT stanje) IS
    VARIABLE pomocna: STD_LOGIC_VECTOR(31 DOWNTO 0);
    VARIABLE pom: stanje;
    VARIABLE i: INTEGER RANGE 0 TO 15;
BEGIN
    rotiraj_stanje(r1_ulaz,pom);
    FOR i IN 0 TO 15 LOOP
        pomocna:=pom(i)+pom(i+16);
        pom(i+16):=pomocna(31 DOWNTO 0);
    END LOOP;
    rotiraj_stanje(pom,r1_izlaz);
END r1;
```

Procedura *rotiraj_stanje* se koristi na početku procedure zato što je zgodnije vršiti transformacije nad rečima u *big-endian* poretku bajtova. Na kraju procedure će se još jednom pozvati *rotiraj_stanje* kako bi se reči vratile u *little-endian* poredak. Ovo važi i za naredne runde (zaključno sa *r10* procedurom).

Dodavanje prvih 16 bita reči drugoj polovini reči se realizuje na sledeći način:

```
FOR i IN 0 TO 15 LOOP
    pomocna:=pom(i)+pom(i+16);
    pom(i+16):=pomocna(31 DOWNTO 0);
END LOOP;
```

Promenljiva *pomocna* sadrži zbir reči x_{0jklm} i x_{1jklm}. Promenljiva *pom* je tipa stanje. U *for* petlji se dodeljuju vrednosti rečima x_{1jklm}:

```
pom(i+16):=pomocna(31 DOWNTO 0);
```

Runda 1 je identična rundi 6 tako da nije potrebno pisati proceduru za rundu 6.

Procedurom *r2* je implementirana druga runda transformacije. U njoj se biti u rečima x_{0jklm} rotiraju ulevo za 7 pozicija.

```
PROCEDURE r2(VARIABLE r2_ulaz: IN stanje; VARIABLE r2_izlaz: OUT stanje) IS
    VARIABLE i: INTEGER RANGE 0 TO 15;
    VARIABLE pomocna: STD_LOGIC_VECTOR(31 DOWNTO 0);
    VARIABLE pom:stanje;
BEGIN
    rotiraj_stanje(r2_ulaz,pom);
    FOR i IN 0 TO 15 LOOP
        pomocna(6 DOWNTO 0):=pom(i)(31 DOWNTO 25);
        pomocna(31 DOWNTO 7):=pom(i)(24 DOWNTO 0);
        pom(i):=pomocna;
    END LOOP;
    rotiraj_stanje(pom,r2_izlaz);
END r2;
```

Ovde se biti u rečima x_{0jklm} rotiraju ulevo za 7 pozicija. To znači npr. da bit sa indeksom 0 dolazi na poziciju 7, bit na poziciji 1 dolazi na poziciju 8, itd. Ovo je realizovano na sledeći način:

```
pomocna(6 DOWNTO 0):=pom(i)(31 DOWNTO 25);
pomocna(31 DOWNTO 7):=pom(i)(24 DOWNTO 0);
```

Na kraju se u petlji vrednost promenljive *pomocna* dodeljuje reči u stanju.

U okviru runde 3 reči na pozicijama x_{00klm} treba da zamene mesta sa odgovarajućim rečima na pozicijama x_{01klm} . Ovo je ostvareno procedurom *r3*.

```
PROCEDURE r3(VARIABLE r3_ulaz: IN stanje; VARIABLE r3_izlaz: OUT stanje) IS
    VARIABLE i: INTEGER RANGE 0 TO 7;
    VARIABLE pom: stanje;
    VARIABLE pomocna: STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
    rotiraj_stanje(r3_ulaz,pom);
    FOR i IN 0 TO 7 LOOP
        pomocna:=pom(i);
        pom(i):=pom(i+8);
```

```
        pom(i+8):=pomocna;
    END LOOP;
    rotiraj_stanje(pom,r3_izlaz);
END r3;
```

Primetimo (kada se pogledaju indeksi reči u stanju) da se zamena može izvršiti tako što se zamene reči na pozicijama od 0 do 7 sa svojim parnjacima 8 pozicija iznad. Ovo je ostvareno sledećim kodom:

```
FOR i IN 0 TO 7 LOOP
    pomocna:=pom(i);
    pom(i):=pom(i+8);
    pom(i+8):=pomocna;
END LOOP;
```

Na ovaj način nije bilo potrebno uključiti u *for* petlju sve reči stanja i tražiti one reči koje odgovaraju indeksima. Rotiranje reči je izvršeno pomoću promenljive *pomocna* u koju se skladišti vrednost reči *pom(i)* kako bi mogla da joj se dodeli vrednost reči *pom(i+8)*. Nakon toga reč *pom(i+8)* dobija vrednost promenljive *pomocna* (a to je reč koja je bila na poziciji *pom(i)*).

Runda 4 vrši logičku *xor* operaciju nad rečima x_{0jklm} i x_{1jklm} i rezultat dodeljuje reči x_{0jklm}. Runda 9 je ista kao i runda 4 tako da će se za njenu implementaciju iskoristiti procedura *r4* napisana za rundu 4.

```
PROCEDURE r4(VARIABLE r4_ulaz: IN stanje; VARIABLE r4_izlaz: OUT stanje) IS
    VARIABLE i: INTEGER RANGE 0 TO 15;
    VARIABLE pom:stanje;
BEGIN
    rotiraj_stanje(r4_ulaz,pom);
    FOR i IN 0 TO 15 LOOP
        pom(i):=pom(i) XOR pom(i+16);
    END LOOP;
    rotiraj_stanje(pom,r4_izlaz);
END r4;
```

Logička *xor* operacija se vrši na sledeći način:

```
    pom(i):=pom(i) XOR pom(i+16);
```

U okviru runde 5 reči na pozicijama x_{1jk0m} treba da zamene mesta sa odgovarajućim rečima na pozicijama x_{1jk1m} . Ovo je ostvareno procedurom *r5*.

```
PROCEDURE r5(VARIABLE r5_ulaz: IN stanje; VARIABLE r5_izlaz: OUT stanje) IS
    VARIABLE i: INTEGER RANGE 16 TO 31;
    VARIABLE indeks: STD_LOGIC_VECTOR (4 DOWNTO 0);
    VARIABLE pom:stanje;
    VARIABLE pomocna:STD_LOGIC_VECTOR (31 DOWNTO 0);
BEGIN
```

```
    rotiraj_stanje(r5_ulaz,pom);
    FOR i IN 16 TO 29 LOOP
        indeks:=CONV_STD_LOGIC_VECTOR(i,5);
        IF indeks(1)='0' THEN
            pomocna:=pom(i);
            pom(i):=pom(i+2);
            pom(i+2):=pomocna;
        END IF;
    END LOOP;
    rotiraj_stanje(pom,r5_izlaz);
END r5;
```

Zamena mesta rečima u stanju je ostvarena na sledeći način:

```
FOR i IN 16 TO 29 LOOP
    indeks:=CONV_STD_LOGIC_VECTOR(i,5);
    IF indeks(1)='0' THEN
        pomocna:=pom(i);
        pom(i):=pom(i+2);
        pom(i+2):=pomocna;
    END IF;
END LOOP;
```

Primetimo, kao i kod implementacije runde 3, da *for* petlja ne treba da pretraži sve reči u stanju. Po indeksima reči se vidi da se pretraga može vršiti počev od reči sa indeksom 16. Takođe, može se videti da reč koja na mestu „*l*“ u indeksu ima nulu menja poziciju sa rečju dve pozicije iznad. Da bi se pronašla reč koja ima nulu na mestu „*l*“ u indeksu napravljena je promenljiva *indeks* koja je *std_logic_vector* tipa i dužine 5 bita. Naredbom *conv_std_logic_vector* brojač u *for* petlji je pretvoren u oblik „*ijklm*“ tako da je vrednost promenljive *indeks(1)* zapravo vrednost bita na poziciji „*l*“. Zamena reči u stanju izvršena je pomoću promenljive *pomocna*.

Runda 7 je slična rundi 2. U njoj se biti u reči rotiraju ulevo za 11 pozicija. Pošto je rotiranje objašnjeno u opisu procedure *r2*, biće naveden samo kod procedure *r7*.

```
PROCEDURE r7(VARIABLE r7_ulaz: IN stanje; VARIABLE r7_izlaz: OUT stanje) IS
    VARIABLE i: INTEGER RANGE 0 TO 15;
    VARIABLE pomocna:STD_LOGIC_VECTOR(31 DOWNTO 0);
    VARIABLE pom:stanje;
BEGIN
    rotiraj_stanje(r7_ulaz,pom);
    FOR i IN 0 TO 15 LOOP
        pomocna(10 DOWNTO 0):=pom(i)(31 DOWNTO 21);
        pomocna (31 DOWNTO 11):=pom(i)(20 DOWNTO 0);
        pom(i):=pomocna;
    END LOOP;
    rotiraj_stanje(pom,r7_izlaz);
END r7;
```

U okviru runde 8 reči na pozicijama x_{0j0lm} treba da zamene mesta sa odgovarajućim rečima na pozicijama x_{0j11m}. Način implementacije ove runde je isti kao i runde 5, tako da će biti naveden samo kod. Procedura koja vrši ovu funkciju je *r8*.

```
PROCEDURE r8(VARIABLE r8_ulaz: IN stanje; VARIABLE r8_izlaz: OUT stanje) IS
    VARIABLE i: INTEGER RANGE 0 TO 11;
    VARIABLE indeks: STD_LOGIC_VECTOR (4 DOWNTO 0);
    VARIABLE pom:stanje;
    VARIABLE pomocna: STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
    rotiraj_stanje(r8_ulaz,pom);
    FOR i IN 0 TO 11 LOOP
        indeks:=CONV_STD_LOGIC_VECTOR(i,5);
        IF indeks(2)='0' THEN
            pomocna:=pom(i);
            pom(i):=pom(i+4);
            pom(i+4):=pomocna;
        END IF;
    END LOOP;
    rotiraj_stanje(pom,r8_izlaz);
END r8;
```

Runda 10 (*r10*) je implementirana na isti način kao i runde 5 i 8 tako da će biti naveden samo kod. Pozicije menjaju reči x_{1jkl0} i x_{1jkl1}.

```
PROCEDURE r10(VARIABLE r10_ulaz: IN stanje; VARIABLE r10_izlaz: OUT stanje) IS
    VARIABLE i: INTEGER RANGE 16 TO 31;
    VARIABLE indeks: STD_LOGIC_VECTOR (4 DOWNTO 0);
    VARIABLE pom:stanje;
    VARIABLE pomocna: STD_LOGIC_VECTOR(31 DOWNTO 0);
BEGIN
    rotiraj_stanje(r10_ulaz,pom);
    FOR i IN 16 TO 30 LOOP
        indeks:=CONV_STD_LOGIC_VECTOR(i,5);
        IF indeks(0)='0' THEN
            pomocna:=pom(i);
            pom(i):=pom(i+1);
            pom(i+1):=pomocna;
        END IF;
    END LOOP;
    rotiraj_stanje(pom,r10_izlaz);
END r10;
```

Uloga procedure *runda* je da izvrši prethodno opisanih deset rundi jednu za drugom. Ova procedura se koristi za generisanje vektora inicijalizacije, finalizacije stanja, kao i nakon izvršavanja *xor* operacije između dolaznog bloka i stanja.

```
PROCEDURE runda(VARIABLE runda_ulaz: IN stanje; VARIABLE runda_izlaz: OUT
stanje) IS
    VARIABLE pomocna:stanje;
BEGIN
    r1(runda_ulaz, pomocna);
    r2(pomocna, pomocna);
    r3(pomocna, pomocna);
    r4(pomocna, pomocna);
    r5(pomocna, pomocna);
```

```
    r1(pomocna, pomocna);
    r7(pomocna, pomocna);
    r8(pomocna, pomocna);
    r4(pomocna, pomocna);
    r10(pomocna, runda_izlaz);
END runda;
```

U kodu procedure *xorovanje* se poziva nekoliko procedura koje će biti objašnjene redosledom kojim se pojavljuju. Ova procedura vrši logičku *xor* operaciju između dolaznog bloka i stanja i transformiše stanje kroz *r* rundi transformacije.

```
PROCEDURE xorovanje(VARIABLE xorovanje_ulaz: IN stanje; SIGNAL blok: IN
STD_LOGIC_VECTOR(b*8-1 DOWNTO 0); VARIABLE xorovanje_izlaz: OUT stanje) IS
    VARIABLE pomocna:stanje;
    VARIABLE razvucena: STD_LOGIC_VECTOR(1023 DOWNTO 0);
BEGIN
    pomocna:=xorovanje_ulaz;
    obrni(pomocna,pomocna);
    razvuci(pomocna,razvucena);
    razvucena(1023 DOWNTO 1023-b*8+1):=razvucena(1023 DOWNTO 1023-b*8+1) XOR
blok;
    skupi(razvucena,pomocna);
    obrni(pomocna,pomocna);
    FOR j IN 0 TO r-1 LOOP
        runda(pomocna,pomocna);
    END LOOP;
    xorovanje_izlaz:=pomocna;
END xorovanje;
```

Jedan od ulaznih argumenata procedure je *blok*. Ova promenljiva je tipa *std_logic_vector* i ima dužinu $b \cdot 8$. Parametar *b* je definisan kao konstanta u fajlu *paket_const.vhd* i predstavlja dužinu jednog bloka poruke u bajtovima. Na ovaj način vrednost parametra *b* se može lako promeniti kako bi odgovarao potrebama korisnika ovog algoritma.

Procedura *obrni* vrši promenu redosleda reči u stanju. Budući da se vrši *xor* operacija između bloka i stanja počev od prve reči u stanju, pogodno je obrnuti redosled reči u stanju i napraviti jednodimenzioni niz u kome će sve reči počev od prve biti poređane jedna za drugom.

```
PROCEDURE obrni(VARIABLE obrni_ulaz: IN stanje; VARIABLE obrni_izlaz: OUT
stanje) IS
BEGIN
    FOR j IN 0 TO 31 LOOP
        obrni_izlaz(j):=obrni_ulaz(31-j);
    END LOOP;
END obrni;
```

Ova procedura se može iskoristiti i za vraćanje stanja u prvobitni oblik.

Procedura *razvuci* pravi od stanja jednodimenzioni niz.

```
PROCEDURE razvuci(VARIABLE razvuci_ulaz: IN stanje; VARIABLE razvuci_izlaz: OUT
STD_LOGIC_VECTOR (1023 DOWNTO 0)) IS
BEGIN
    razvuci_izlaz(31 DOWNTO 0):=razvuci_ulaz(0);
    razvuci_izlaz(63 DOWNTO 32):=razvuci_ulaz(1);
    razvuci_izlaz(95 DOWNTO 64):=razvuci_ulaz(2);
    razvuci_izlaz(127 DOWNTO 96):=razvuci_ulaz(3);
    razvuci_izlaz(159 DOWNTO 128):=razvuci_ulaz(4);
    razvuci_izlaz(191 DOWNTO 160):=razvuci_ulaz(5);
    razvuci_izlaz(223 DOWNTO 192):=razvuci_ulaz(6);
    razvuci_izlaz(255 DOWNTO 224):=razvuci_ulaz(7);
    razvuci_izlaz(287 DOWNTO 256):=razvuci_ulaz(8);
    razvuci_izlaz(319 DOWNTO 288):=razvuci_ulaz(9);
    razvuci_izlaz(351 DOWNTO 320):=razvuci_ulaz(10);
    razvuci_izlaz(383 DOWNTO 352):=razvuci_ulaz(11);
    razvuci_izlaz(415 DOWNTO 384):=razvuci_ulaz(12);
    razvuci_izlaz(447 DOWNTO 416):=razvuci_ulaz(13);
    razvuci_izlaz(479 DOWNTO 448):=razvuci_ulaz(14);
    razvuci_izlaz(511 DOWNTO 480):=razvuci_ulaz(15);
    razvuci_izlaz(543 DOWNTO 512):=razvuci_ulaz(16);
    razvuci_izlaz(575 DOWNTO 544):=razvuci_ulaz(17);
    razvuci_izlaz(607 DOWNTO 576):=razvuci_ulaz(18);
    razvuci_izlaz(639 DOWNTO 608):=razvuci_ulaz(19);
    razvuci_izlaz(671 DOWNTO 640):=razvuci_ulaz(20);
    razvuci_izlaz(703 DOWNTO 672):=razvuci_ulaz(21);
    razvuci_izlaz(735 DOWNTO 704):=razvuci_ulaz(22);
    razvuci_izlaz(767 DOWNTO 736):=razvuci_ulaz(23);
    razvuci_izlaz(799 DOWNTO 768):=razvuci_ulaz(24);
    razvuci_izlaz(831 DOWNTO 800):=razvuci_ulaz(25);
    razvuci_izlaz(863 DOWNTO 832):=razvuci_ulaz(26);
    razvuci_izlaz(895 DOWNTO 864):=razvuci_ulaz(27);
    razvuci_izlaz(927 DOWNTO 896):=razvuci_ulaz(28);
    razvuci_izlaz(959 DOWNTO 928):=razvuci_ulaz(29);
    razvuci_izlaz(991 DOWNTO 960):=razvuci_ulaz(30);
    razvuci_izlaz(1023 DOWNTO 992):=razvuci_ulaz(31);
END razvuci;
```

Procedura *skupi* vraća jednodimenzioni niz u formu tipa *stanje*.

```
PROCEDURE skupi(VARIABLE skupi_ulaz: IN STD_LOGIC_VECTOR(1023 DOWNTO 0);
VARIABLE skupi_izlaz: OUT stanje) IS
BEGIN
    skupi_izlaz(0):=skupi_ulaz(31 DOWNTO 0);
    skupi_izlaz(1):=skupi_ulaz(63 DOWNTO 32);
    skupi_izlaz(2):=skupi_ulaz(95 DOWNTO 64);
    skupi_izlaz(3):=skupi_ulaz(127 DOWNTO 96);
    skupi_izlaz(4):=skupi_ulaz(159 DOWNTO 128);
    skupi_izlaz(5):=skupi_ulaz(191 DOWNTO 160);
    skupi_izlaz(6):=skupi_ulaz(223 DOWNTO 192);
    skupi_izlaz(7):=skupi_ulaz(255 DOWNTO 224);
    skupi_izlaz(8):=skupi_ulaz(287 DOWNTO 256);
    skupi_izlaz(9):=skupi_ulaz(319 DOWNTO 288);
    skupi_izlaz(10):=skupi_ulaz(351 DOWNTO 320);
    skupi_izlaz(11):=skupi_ulaz(383 DOWNTO 352);
    skupi_izlaz(12):=skupi_ulaz(415 DOWNTO 384);
    skupi_izlaz(13):=skupi_ulaz(447 DOWNTO 416);
    skupi_izlaz(14):=skupi_ulaz(479 DOWNTO 448);
```

```
    skupi_izlaz(15):=skupi_ulaz(511 DOWNTO 480);
    skupi_izlaz(16):=skupi_ulaz(543 DOWNTO 512);
    skupi_izlaz(17):=skupi_ulaz(575 DOWNTO 544);
    skupi_izlaz(18):=skupi_ulaz(607 DOWNTO 576);
    skupi_izlaz(19):=skupi_ulaz(639 DOWNTO 608);
    skupi_izlaz(20):=skupi_ulaz(671 DOWNTO 640);
    skupi_izlaz(21):=skupi_ulaz(703 DOWNTO 672);
    skupi_izlaz(22):=skupi_ulaz(735 DOWNTO 704);
    skupi_izlaz(23):=skupi_ulaz(767 DOWNTO 736);
    skupi_izlaz(24):=skupi_ulaz(799 DOWNTO 768);
    skupi_izlaz(25):=skupi_ulaz(831 DOWNTO 800);
    skupi_izlaz(26):=skupi_ulaz(863 DOWNTO 832);
    skupi_izlaz(27):=skupi_ulaz(895 DOWNTO 864);
    skupi_izlaz(28):=skupi_ulaz(927 DOWNTO 896);
    skupi_izlaz(29):=skupi_ulaz(959 DOWNTO 928);
    skupi_izlaz(30):=skupi_ulaz(991 DOWNTO 960);
    skupi_izlaz(31):=skupi_ulaz(1023 DOWNTO 992);
END;
```

Promenljiva *razvucena* u proceduri *xorovanje* sadrži jednodimenzioni niz dužine 1024 bita koji je dobijenređanjem reči stanja jedne za drugom počev od prve. Nad ovako dobijenim nizom i dolaznim blokom se vrši logička *xor* operacija:

```
razvucena(1023 DOWNTO 1023-b*8+1):=razvucena(1023 DOWNTO 1023-b*8+1) XOR blok;
```

Nakon toga je potrebno vratiti jednodimenzioni niz u formu tipa *stanje* i obrnuti redosled reči kako bi se vratile u poredak sa početka procedure:

```
skupi(razvucena,pomocna);
obrni(pomocna,pomocna);
```

Na kraju se izvršava *r* rundi transformacije:

```
FOR j IN 0 TO r-1 LOOP
    runda(pomocna,pomocna);
END LOOP;
```

Parametar *r* je, kao i *b*, opisan u odeljku za definisanje konstanti i može mu se menjati vrednost.

Procedura *hes* kao izlazni parametar daje heš vrednost poruke.

```
PROCEDURE hes(VARIABLE hes_ulaz: IN stanje; VARIABLE hes_izlaz: OUT
STD_LOGIC_VECTOR (h-1 DOWNTO 0)) IS
    VARIABLE pomocna: STD_LOGIC_VECTOR (1023 DOWNTO 0);
    VARIABLE pom_sta: stanje;
BEGIN
    obrni(hes_ulaz, pom_sta);
    razvuci(pom_sta,pomocna);
```

```
    hes_izlaz:=pomocna(1023 DOWNTO 1023-h+1);
END;
```

Pošto heš vrednost čine prvih *h* bita stanja, potrebno je prvo obrnuti redosled reči u stanju i pretvoriti dato stanje u jednodimenzioni niz:

```
obrni(hes_ulaz, pom_sta);
razvuci(pom_sta,pomocna);
```

Heš vrednost se onda dobija jednostavnim „odsecanjem“ prvih *h* bita dobijenog niza.

I na kraju, procedura *final* vrši finalizaciju stanja.

```
PROCEDURE final(VARIABLE final_ulaz: IN stanje; VARIABLE final_izlaz: OUT
STD_LOGIC_VECTOR (h-1 DOWNTO 0)) IS
    VARIABLE pom_rec: STD_LOGIC_VECTOR(31 DOWNTO 0);
    VARIABLE pomocna: stanje;
BEGIN
    pomocna:=final_ulaz;
    pom_rec:=pomocna(31);
    rotiraj(pom_rec,pom_rec);
    pom_rec:=pom_rec XOR conv_std_logic_vector(1,32);
    rotiraj(pom_rec,pomocna(31));
    FOR j IN 0 TO f-1 LOOP
        runda(pomocna, pomocna);
    END LOOP;
    hes(pomocna, final_izlaz);
END;
```

Potrebno je izvršiti logičku *xor* operaciju između zadnje reči u stanju i celog broja 1. Kako je redosled bita u reči u *big-endian* poretku, potrebno je prvo obrnuti redosled u *little-endian*. Nakon toga može da se izvrši *xor* operacija. Zatim se poredak bita vraća u *big-endian*.

```
pom_rec:=pomocna(31);
rotiraj(pom_rec,pom_rec);
pom_rec:=pom_rec XOR conv_std_logic_vector(1,32);
rotiraj(pom_rec,pomocna(31));
```

Nakon toga stanje prolazi kroz *f* rundi transformacije (vrednost parametra *f* se podešava u odeljku za opis konstanti). Na kraju se poziva procedura *hes* tako da se na izlazu procedure *final* dobija heš vrednost poruke.

3.3.2. Opis rada top-level entiteta

Pre nego što se počne sa opisom koda *top-level* entiteta biće navedena njegova arhitektura. Procedure korišćene u kodu su objašnjene u prethodnom odeljku.

```
ARCHITECTURE shema OF master IS
BEGIN
    PROCESS (clk,reset)
        VARIABLE pomocna: stanje;
        VARIABLE stanje_automata: konacni_automat;
        VARIABLE izlaz_var: STD_LOGIC_VECTOR(h-1 DOWNTO 0);
    BEGIN
        IF (reset='1') THEN
            stanje_automata:= idle;
            init_napravi(pomocna);
        ELSE
            IF (clk'EVENT AND clk='1') THEN
                IF (prvi='1') THEN
                    stanje_automata:= obrada;
                END IF;
                IF (stanje_automata= obrada) THEN
                    xorovanje(pomocna,blok,pomocna);
                    IF (zadnji='1') THEN
                        stanje_automata:= finalizacija;
                    END IF;
                END IF;
                IF (stanje_automata= finalizacija) THEN
                    final(pomocna,izlaz_var);
                    izlaz<=izlaz_var;
                    stanje_automata:= idle;
                END IF;
            END IF;
        END IF;
    END PROCESS;
END shema;
```

U pitanju je sekvencijalni kod i on se nalazi unutar procesa. U listi osetljivosti se nalaze signali *clk* i *reset*. Signal *clk* je signal takta što znači da se kod unutar procesa izvršava na svaki signal takta. U implementaciji se nalazi i signal asinhronog reseta.

U slučaju aktivne vrednosti signala *reset*, stanje automata prelazi u *idle* i to znači da obrada blokova poruke nije u toku. Takođe, vrši se i generisanje vektora inicijalizacije tako da se sa obradom blokova poruke može početi čim oni stignu. Ovo je moguće uraditi zato što su parametri potrebni za generisanje ovog vektora unapred zadati i ne zavise od pristiglog bloka poruke.

U cilju lakšeg objašnjenja koda koji sledi (u slučaju neaktivne vrednosti signala *reset*), biće navedeni signali koji se koriste u kodu.

```
ENTITY master IS
    PORT
    (
        clk, reset, prvi, zadnji:IN STD_LOGIC;
        izlaz: OUT STD_LOGIC_VECTOR(h-1 DOWNTO 0);
        blok: IN STD_LOGIC_VECTOR(b*8-1 DOWNTO 0)
    );
END master;
```

Signali *clk* i *reset* su već objašnjeni. Aktivna vrednost signala *prvi* signalizira da se prvi blok poruke nalazi na ulazu dizajna, a aktivna vrednost signala *zadnji* da se zadnji blok poruke nalazi na ulazu. Signal *izlaz* predstavlja konačnu vrednost obrade poruke, to jest heš. On je tipa *std_logic_vector* dužine *h*. Parametar *h* predstavlja dužinu heša i može se podesiti u odeljku za definisanje konstanti u paketu *paket_const.vhd*. Signal *blok* predstavlja jedan blok poruke i dužine je $b \cdot 8$. Vrednost parametra *b* predstavlja dužinu bloka poruke u oktetima. Vrednost parametra *b* se može menjati na isti način kao i vrednost parametra *h*.

Obrada poruke se izvršava onda kada signal reseta ima neaktivnu vrednost. Uzlazna ivica signala takta je okidač za izvršavanje koda. Koji će se deo koda izvršiti zavisi od vrednosti signala na ulazu dizajna kao i stanja konačnog automata. Sledi objašnjenje.

Ako je vrednost signala *prvi* '1' , to znači da je na ulaz stigao prvi blok poruke za obradu. Stanje automata tada treba da dobije vrednost *obrada* što znači da se počelo sa računanjem heša poruke.

```
IF (prvi='1') THEN
    stanje_automata:= obrada;
END IF;
```

Ako je vrednost stanja automata *obrada* onda to znači da se treba izvršiti logička *xor* operacija između bloka i stanja, kao i transformacija stanja kroz *r* rundi transformacije. To radi procedura *xorovanje*, kao što je već objašnjeno. Ovu proceduru je potrebno izvršiti za svaki pristigli blok. Konačni automat je u stanju *obrada* dokle god postoje blokovi na ulazu u dizajn. Ako je na ulazu u dizajn aktivan signal *zadnji*, to znači da je stigao zadnji blok poruke i onda je neophodno stanje automata postaviti na vrednost *finalizacija* kako bi se finalizacija izvršila.

```
IF (stanje_automata= obrada) THEN
    xorovanje(pomocna,blok,pomocna);
    IF (zadnji='1') THEN
        stanje_automata:= finalizacija;
    END IF;
END IF;
```

Kada je na ulazu u dizajn aktivna vrednost signala *finalizacija*, potrebno je izbaciti heš i postaviti vrednost konačnog automata na *idle*. Ovim se daje do znanja da je završena obrada poruke i da se čeka dospeće nove poruke i njenog prvog bloka kako bi obrada počela i stanje izašlo iz *idle* vrednosti. Kontrolna izlazna linija za preuzimanje heša ne postoji (nema potrebe za tim signalom) zato što se rezultujući heš uvek postavlja u taktu iza prijema zadnjeg bloka poruke. Heš vrednost kao rezultat procedure *final* se dodeljuje signalu *izlaz*.

```
IF (stanje_automata= finalizacija) THEN
    final(pomocna,izlaz_var);
    izlaz<=izlaz_var;
    stanje_automata:= idle;
END IF;
```

4. OPIS PERFORMANSI I VERIFIKACIJA DIZAJNA

4.1. OPIS PERFORMANSI

Kao što je već rečeno, parametri algoritma (konstante *i, r, f, b* i *h*) mogu da se menjaju kako bi algoritam odgovarao zahtevima korisnika. U ovom potpoglavlju će biti analizirane performanse nekoliko verzija algoritma u cilju poređenja performansi dizajna u zavisnosti od izabranih parametara.

Za potrebe opisa performansi nisu korišćene vrednosti koje je autor preporučio iz razloga što se sa povećanjem parametara *i, r* i *f* povećava i vreme potrebno za analizu i sintezu dizajna. Performanse će se analizirati pomoću proizvoljno izabranih parametara manjih vrednosti bez gubitka opštosti analize. Za sve četiri verzije algoritma izabran je uređaj *XQ6VLX550T* familije *Defence-Grade Virtex-6Q*, proizvođača *Xilinx*.

Tabela 4.1.1. Poređenje performansi za različite vrednosti parametara

	Vrednost			
CubeHash i+r/b+f-h	***10+1/1+10-512***	***30+3/1+30-512***	***10+1/32+10-512***	***10+1/1+10-224***
Broj slajs registara	1538/687360 (0%)	1537/687360 (0%)	1537/687360 (0%)	1249/687360 (0%)
Broj slajs LUT-ova	16313/343680 (4%)	45829/343680 (13%)	16211/343680 (13%)	15506/343680 (4%)
Broj potpuno iskorišćenih parova LUT-FF	1396/31395 (8%)	1203/46163 (2%)	1260/16488 (7%)	1103/15652 (7%)
Broj globalnih taktova (BUFG/BUFGCTRL)	1/32 (3%)	1/32 (3%)	1/32 (3%)	1/32 (3%)
Broj pinova	524/840 (62%)	524/840 (62%)	772/840 (91%)	236/840 (28%)
Maksimalna frekvencija	38.611MHz	13.657MHz	38.513MHz	38.679MHz

Prema podacima iz Tabele 4.1.1. može se primetiti da se broj pinova na čipu menja sa izborom parametara *h* i *b*. Na primer, verzija *CubeHash 10+1/1+10-512* ima 524 pina. Heš zauzima 512, blok od jednog okteta zauzima 8 pinova i 4 pina odlaze na signale *clk*, *reset*, *prvi* i *zadnji*. Implementacija *10+1/32+10-512* zauzima 248 pinova više zato što je dužina bloka 32 okteta (što je za 31 bajt više od dužine bloka u prethodno opisanoj implementaciji).

Vidi se i da se sa povećanjem parametara *i*, *r* i *f* proporcionalno povećava i broj resursa, najviše kombinacione logike, tj. LUT elemenata.

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*.

Primećuje se da je maksimalna frekvencija dizajna usko povezana sa izborom parametara *i*, *f* i *r*. U implementaciji *30+3/1+30-512* je oko tri puta niža nego u ostalim verzijama zato što su dati parametri tri puta veći. Stoga je potrebno u svakom taktu izvršiti tri puta više rundi nego u ostalim slučajevima i ovakva vrednost maksimalne frekvencije je bila očekivana jer je kombinaciona logika veća, a samim tim i kašnjenje kroz nju.

Vrednosti parametara koje je autor algoritma predložio su bile:

- *CubeHash 160+16/32+160-224*,
- *CubeHash 160+16/32+160-256*,
- *CubeHash 160+16/32+160-384*,
- *CubeHash 160+16/32+160-512*,
- *CubeHash 160+16/1+160-384*,
- *CubeHash 160+16/1+160-512*.

4.2. VERIFIKACIJA DIZAJNA

U ovom potpoglavlju će biti opisana verifikacija procedura kao i celokupnog dizajna *CubeHash* algoritma. Testiraće se poklapanje vrednosti dobijenih nakon pokretanja funkcionalne simulacije u okviru *ISim* simulatora sa vrednostima u referentnom fajlu. Referentni fajl je preuzet sa zvaničnog sajta instituta *NIST* i kompletan fajl će biti priložen na CD-u pod imenom *intermediate-values.txt*.

Izabran je primer algoritma sa sledećim parametrima:

h=512

b=32

r=16

i=160

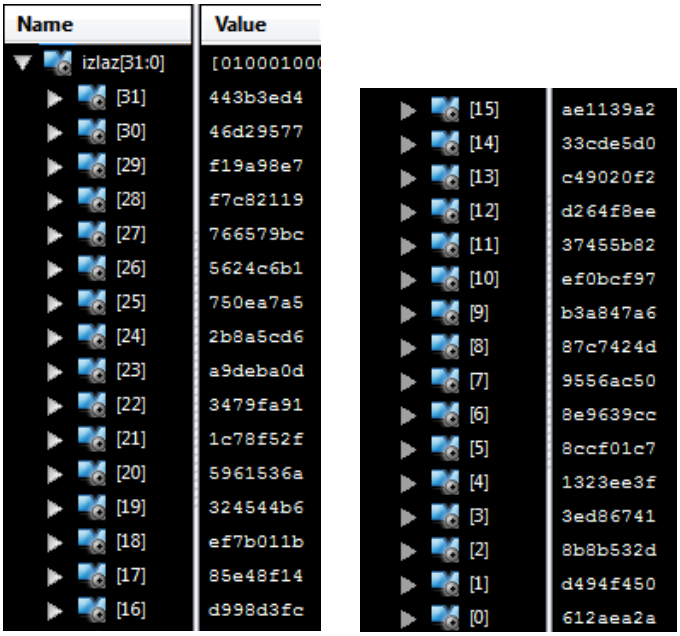
f=160

Poruka čiji se heš traži je dužine 24 bita tako da će postojati samo jedan blok (jer je dužina bloka 32 okteta).

Procedura *init_napravi* u sebi sadrži *i*=160 ponavljanja procedura *r1-r10* tako da će te procedure biti verifikovane verifikacijom procedure *init_napravi*.

Sledi poređenje referentnih [9] i dobijenih vrednosti vektora iniijalizacije. Prvo će biti navedena vrednost stanja u referentnom fajlu (reči su poređane od nulte do trideset prve), a zatim prikaz prozora *ISim* simulatora

```
612aea2a d494f450 8b8b532d 3ed86741 1323ee3f 8ccf01c7 8e9639cc 9556ac50
87c7424d b3a847a6 ef0bcf97 37455b82 d264f8ee c49020f2 33cde5d0 ae1139a2
d998d3fc 85e48f14 ef7b011b 324544b6 5961536a 1c78f52f 3479fa91 a9deba0d
2b8a5cd6 750ea7a5 5624c6b1 766579bc f7c82119 f19a98e7 46d29577 443b3ed4
```

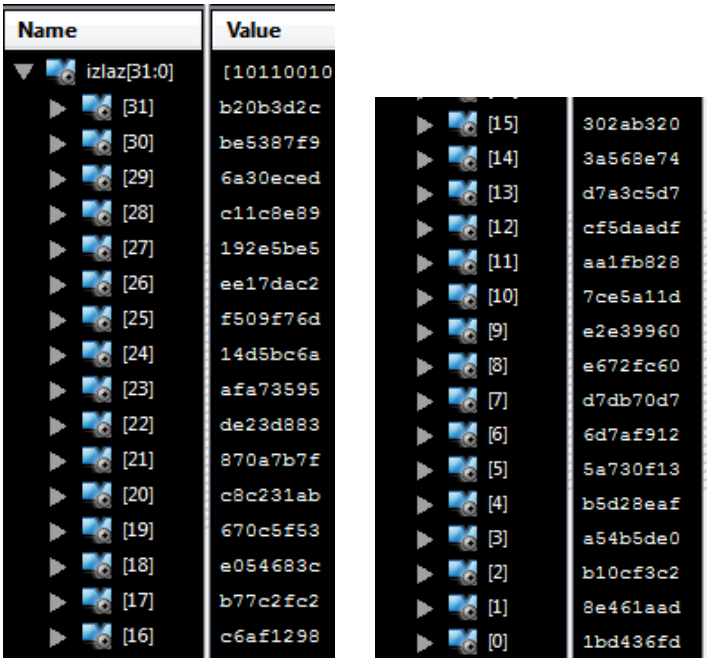


Slika 4.2.1. Vrednost stanja nakon procedure *init_napravi*

Primećuje se da su dobijene vrednosti jednake referentnim i time je procedura *init_napravi* verifikovana (zajedno sa procedurama *r1-r10*).

Sada je na redu verifikacija procedure *xorovanje*. Ova procedura vrši logičku *xor* operaciju između bloka poruke i prvih *b* bajtova stanja. Zatim se stanje transformiše kroz *r* rundi transformacije (u ovom slučaju 16).

```
1bd436fd 8e461aad b10cf3c2 a54b5de0 b5d28eaf 5a730f13 6d7af912 d7db70d7
e672fc60 e2e39960 7ce5a11d aa1fb828 cf5daadf d7a3c5d7 3a568e74 302ab320
c6af1298 b77c2fc2 e054683c 670c5f53 c8c231ab 870a7b7f de23d883 afa73595
14d5bc6a f509f76d ee17dac2 192e5be5 c11c8e89 6a30eced be5387f9 b20b3d2c
```

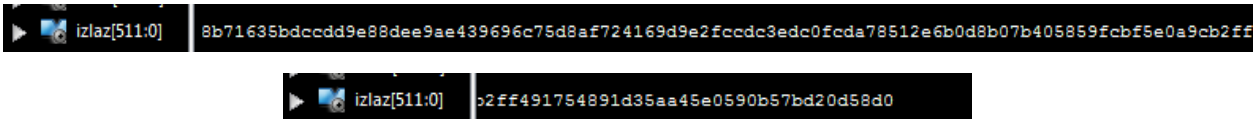


Slika 4.2.2. Vrednost stanja nakon procedure *xorovanje*

Vidi se da se i ove vrednosti podudaraju tako da je procedura xorovanje na ovaj način verifikovana.

Ostaje još da se verifikuje procedura *final*. Ona vrši logičku *xor* operaciju između broja 1 i zadnje reči stanja. Nakon toga stanje se propušta kroz *f* rundi transformacije (u ovom slučaju 160). Na izlazu, procedura daje traženu vrednost heša dužine *h*.

```
8b71635bdccdd9e88dee9ae439696c75d8af724169d9e2fccdc3edc0fcda78512e6b0d8b07b40585
9fcbf5e0a9cb2ff491754891d35aa45e0590b57bd20d58d0
```



Slika 4.2.3. Vrednost stanja nakon procedure *final* (heš vrednost poruke)

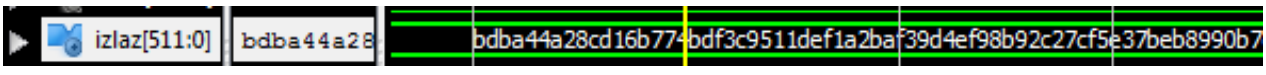
Poređenjem vrednosti se vidi da se heš vrednosti poklapaju u referentnom fajlu i simulaciji.

Autor algoritma nije priložio međuvrednosti stanja u slučaju da se poruka sastoji od više od jednog bloka. Stoga će heš vrednosti poruke koja sadrži više od jednog bloka biti upoređene sa primerima sa *Wikipedia*-e [1].

String čija se heš vrednost traži je „*The quick brown fox jumps over the lazy dog*“.

CubeHash 160+16/32+160-512:

bdba44a28cd16b774bdf3c9511def1a2baf39d4ef98b92c27cf5e37beb8990b7\
cdb6575dae1a548330780810618b8a5c351c1368904db7ebdf8857d596083a86

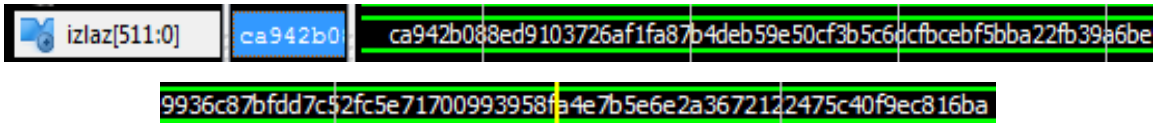


Slika 4.2.4. Heš vrednost poruke za *CubeHash 160+16/32+160-512*

CubeHash 80+8/1+80-512:

ca942b088ed9103726af1fa87b4deb59e50cf3b5c6dcfbcebf5bba22fb39a6be\

9936c87bfdd7c52fc5e71700993958fa4e7b5e6e2a3672122475c40f9ec816ba



Slika 4.2.5. Heš vrednost poruke za *CubeHash 80+8/1+80-512*

Primećuje se da se vrednosti poklapaju. Ovim je verifikovan dizajn u slučaju da poruka sadrži više od jednog bloka. U prvom slučaju poruka je imala 2 bloka, a u drugom 44.

5. ZAKLJUČAK

Kao što je već rečeno u uvodu, *CubeHash* algoritam je bio jedan od 14 kandidata druge runde za izbor novog *SHA-3* algoritma za heširanje. Cilj svakog dizajnera heš funkcija je da napravi funkciju koja će biti otporna na napade, tj. da bilo koja vrsta napada bude računarski neizvodiva. U ovom poglavlju će biti navedeni subjektivni utisci autora ovog rada o kompleksnosti implementacija kao i o mogućim poboljšanjima algoritma.

Pre svega treba napomenuti da je *CubeHash* algoritam izuzetno dobro dokumentovan. Autor ovog rada je na Internetu mogao da prikupi dovoljno materijala koji su pomogli u razumevanju ovog algoritma.

Dobra strana ovog algoritma je jednostavnost implementacije. Glavne celine algoritma (inicijalizacija, finalizacija i operacije nad pristiglim blokovima) su mogle jednostavno da se podele na manje celine. Odatle sledi da se ceo programski kod sastoji od nekoliko procedura koje u sebi sadrže određeni broj potprocedura. Ovakva organizacija programskog koda omogućava lakše razumevanje i snalaženje u programskom kodu.

U Potpoglavlju 4.1. je rečeno da se u analizi performansi nisu koristile vrednosti koje je autor preporučio iz razloga što je vreme potrebno za analizu i sintezu dizajna bilo predugo. Npr. za implementaciju *CubeHash 80+8/1+80-512* proces analize i sinteze je zaustavljen jer nakon više od 24 sata nije dao nikakve rezultate. Autor ovog rada je, takođe, radio i hardversku implementaciju *Keccak* algoritma (pobednika *NIST* konkursa) i nije naišao na probleme ovakvog tipa. Problem predugog trajanja analize i sinteze *CubeHash* implementacije potiče od suviše velike kombinacione logike. Uvođenjem registara između pojedinih delova kombinacione logike ovaj problem se može prevazići ali tada se povećavaju upotrebljeni hardverski resursi, kao i latencija koju unosi *CubeHash* implementacija.

Takođe, jedan od problema sa kojima se autor ovog rada susreo su i poteškoće pri pronalaženju odgovarajućeg čipa koji bi bio u stanju da podrži dizajn. Ovaj problem se pojavio već pri izboru čipa za verziju *CubeHash 10+1/1+10-512*. Očigledno je da dati dizajn troši previše resursa na čipu. U Potpoglavlju 4.1. se može videti da se iskorišćenost resursa na čipu povećava proporcionalno sa povećanjem broja rundi u dizajnu. Ako se uzme u obzir da se sigurnost funkcije povećava sa povećanjem rundi i smanjenjem dužine bloka poruke, može se zaključiti da „sigurnije“ verzije algortima troše previše resursa. Dakle, dati algoritam sa vrednostima parametara koje je autor preporučio je nepraktičan za izvedbu na čipu (bilo koje veličine) na način predstavljen u ovom radu.

Budući da se u svakom dizajnu teži povećanju maksimalne frekvencije, to se može ostvariti upotrebom *pipelining*-a. Ovakva struktura omogućava izvršavanje više operacija algoritma istovremeno, povećavajući na taj način maksimalnu frekvenciju dizajna. Mana ove strukture je ta što se maksimalna frekvencija dizajna povećava na račun zauzeća više resursa na čipu. Kako je zauzeće previše resursa na čipu u početku bio problem ove implementacije, povećanje frekvencije upotrebom *pipelining* strukture se preporučuje jedino ako je povećanje frekvencije od ključne važnosti.

LITERATURA

[1] Wikipedia: CubeHash. Preuzeto sa: http://en.wikipedia.org/wiki/CubeHash

[2] B. Preneel, The First 30 Years of Cryptographic Hash Functions and the NIST SHA-3 Competition. Preuzeto sa: https://www.cosic.esat.kuleuven.be/publications/article-1532.pdf

[3] John Edward Silva, An Overview of Cryptographic Hash Functions and Their Uses. Preuzeto sa http://www.sans.org/reading-room/whitepapers/vpns/overview-cryptographic-hash-functions-879

[4] Wikipedia: Information security. Preuzeto sa http://en.wikipedia.org/wiki/Information_security#Cryptography

[5] Miodrag Milić, Vojin Šenk, Uniform Logical Cryptoanalysis of CubeHash Function. Preuzeto sa http://www.doiserbia.nb.rs/img/doi/0353-3670/2010/0353-36701003357M.pdf

[6] Daniel Rešetar, Heš funkcije. Preuzeto sa http://www.vladimirbozovic.net/univerzitet/bozovic/wp-content/uploads/2011/06/dipV10.doc

[7] Daniel J. Bernstein, CubeHash specification (2.B.1). Preuzeto sa http://cubehash.cr.yp.to/submission2/spec.pdf

[8] Pipelining, Toan Nguyen. Preuzeto sa http://www.cs.sjsu.edu/faculty/lee/cs147/Pipelining%20Toan.ppt

[9] intermediate-values.txt. Preuzeto sa http://ehash.iaik.tugraz.at/wiki/CubeHash

[10] Metadata for authenticity: hash functions and digital signatures. Preuzeto sa http://www.paradigm.ac.uk/workbook/metadata/authenticity-fixity.html