ELEKTROTEHNIČKI FAKULTET UNIVERZITETA U BEOGRADU

HARDVERSKA IMPLEMENTACIJA *FUGUE* ALGORITMA ZA HEŠIRANJE

– Master rad –

Kandidat:

Nina Bijelić 2012/3092

Mentor:

doc. dr Zoran Čiča

Beograd, Septembar 2014.

SADRŽAJ

1. UVOD

O sigurnosti u digitalnom svetu ranije se govorilo samo u oblasti vojske, vazduhoplovnih i svemirskih tehnologija, ali kako je Internet postao svakodnevica poput vode i struje, o njoj govore vlade zemalja, kompanije, pa i prosečni korisnici Interneta. Pojavom e-trgovine i e-bankarstva, veliki deo privatnog života, pored poslovnog, oslanja se na korišćenje Interneta. Kako se svakodnevno može čuti o virusima koji se šire putem elektronske pošte ili o hakerima koji kradu podatke sa kreditnih kartica, neophodno je da komunikacija putem Interneta i njeni korisnici budu zaštićeni od zlonamernih napada. Kao odgovor na raznovrsne zlonamerne napade, razvijeni su mnogobrojni zaštitni mehanizmi. Neki od mehanizama funkcionišu na osnovu heš algoritama. Heš algoritmi u kriptografiji nalaze primenu prilikom šifrovanja komunikacije, provere integriteta poruka, autentifikacije, itd...

Ovaj rad se bavi hardverskom implementacijom *Fugue* algoritma za heširanje, koji je bio jedan od kandidata za novi SHA-3 (*Secure Hash Algorithm*-3) standard u kriptovanju podataka. Za realizaciju implementacije koristi se VHDL programski jezik, a razvoj i verifikacija dizajna vrši se u ISE razvojnom okruženju za FPGA čipove proizvođača *Xilinx*. Svi projekti (implementacija *Fugue* algoritma za četiri različite dužine izlaza) će biti dati u elektronskoj formi na priloženom CD-u.

Rezultat rada, pored implementacije pomenutog algoritma, je i verifikacija i analiza performansi implementacije. Praktičnu primenu implementacija može naći u mehanizmima mrežne zaštite na mrežnim uređajima.

Ostatak rada je organizovan na sledeći način: Drugo poglavlje daje definiciju i osobine heš algoritama, njihovu primenu i poznate predstavnike. Potom sledi objašnjenje *Fugue* algoritma primenljivo na sve četiri dužine izlaza (224, 256, 384 i 512 bita). Treće poglavlje se bavi opisivanjem realizovane implementacije, na primeru verzije sa 256-bitnom heš vrednošću. Nakon opisa dizajna po principu crne kutije, detaljno su opisane funkcije i procedure napisane za realizaciju koraka *Fugue* algoritma, a potom i kod koji vrši celokupno heširanje. Potom su navedene razlike u kodu za ostale dužine heš vrednosti. U četvrtom poglavlju dat je pregled performansi za sve dužine izlaza: upotrebljeni resursi na čipu i maksimalna frekvencija na kojoj dizajn može da radi. Takođe je prikazana i verifikacija dizajna za stariju verziju *Fugue* algoritma, za koju su dostupni test vektori. Peto poglavlje sadrži subjektivne utiske autora ovog rada o realizovanom algoritmu, kao i predloge za njegovu optimizaciju.

2. HEŠ ALGORITMI

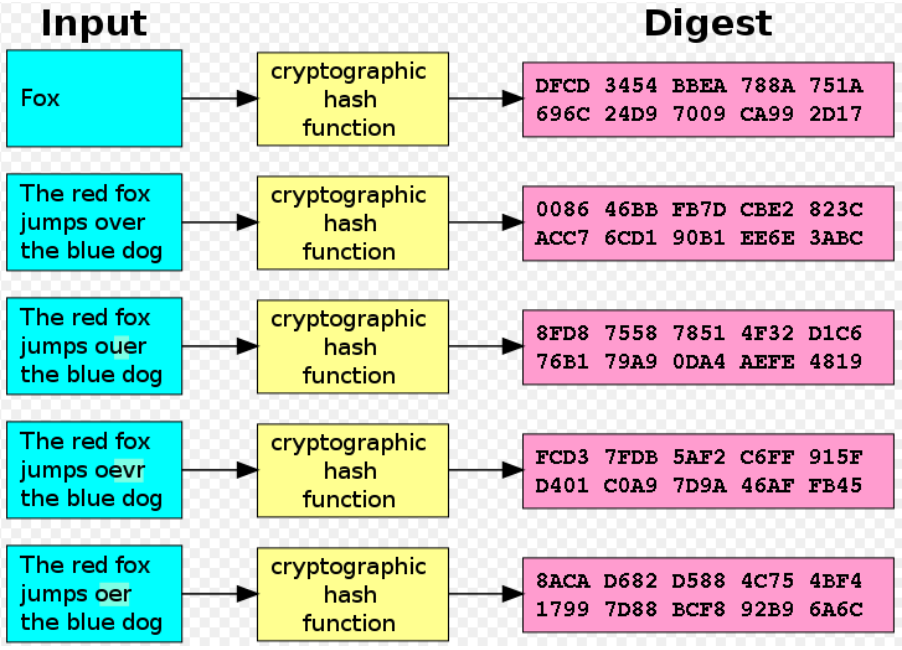
2.1. Definicija i osobine

Prve definicije i analize kriptografskih heš algoritama mogu se naći u kasnim 1970-im. Može se reći da su pokretači bili *Diffie* i *Hellman*, koji su 1976. godine ukazali na potrebu za heš algoritmom, koji bi služio kao osnova za digitalni potpis [1]. U literaturi se za heš algoritam često upotrebljava naziv heš funkcija.

Ulazni podaci nad kojima se primenjuje heš funkcija se nazivaju poruka, a vrednost koja se dobija nakon primene funkcije se zove heš vrednost ili heš kod. Jedna od karakteristika heš funkcija je ta da poruka može biti proizvoljne dužine, ali heš vrednost je fiksne dužine. Za primenu u kriptografiji, glavne osobine koje bi idealna heš funkcija trebalo da poseduje su sledeće [2]:

- Lako izračunavanje heš vrednosti za bilo koju poruku
- Rekonstrukcija poruke iz heš vrednosti je neizvodljiva
- Promena poruke uz zadržavanje iste heš vrednosti nije izvodljiva
- Nemoguće (verovatnoća je zanemarljivo mala) je naći dve poruke sa istom heš vrednošću

Stoga, kriptografska heš funkcija bi trebalo da se ponaša što je više moguće kao random funkcija, ali da i dalje bude deterministička (za istu poruku funkcija proizvodi istu heš vrednost) i moguća za računanje. Na slici 2.1.1 se može videti kako mala promena u poruci (reč *over*) rezultira drastičnom promenom u heš vrednosti u slučaju SHA-1 kriptografskog heš algoritma.



Slika 2.1.1. Poruke i njihove heš vrednosti nakon primene SHA-1 algoritma [2]

Nakon pojavljivanja, broj dizajniranih heš funkcija je rastao u 1980-im i 1990-im, mada mnoge nisu izdržale kriptoanalitičke napade. Kriptografska heš funkcija mora biti u stanju da izdrži sve poznate tipove kriptoanalitičkih napada, a to se postiže zadovoljavanjem gore navedenih osobina. Od 1987. godine objavljivani su radovi o heš funkcijama otpornim na napade [1].

2.2. Predstavnici kriptografskih heš algoritama

Krajem osamdesetih i početkom devedesetih godina, najčešće korišćeni algoritam je bio MD5. Institut NIST (*National Institute for Standards and Technology*) objavljuje poboljšanu verziju ovog algoritma pod imenom SHA-0, zbog sumnje u sigurnost MD5 algoritma. Narednih godina, iz istih razloga objavljuju se novi algoritmi, SHA-1 i SHA-2 [1]. O tome koliko se napora ulaže da bi se proverila otpornost heš funkcije na kriptoanalitičke napade, govori sledeća činjenica: da bi se utvrdilo da li je SHA-0 algoritam otporan na jednu vrstu kriptoanalitičkih napada, utrošeno je 80,000 CPU sati (jedan CPU sat označava stopostotni rad procesora u trajanju od jednog sata) na superkompjuteru sa 256 *Itanium 2* procesora (što je ekvivalentno neprekidnom radu kompjutera tokom 13 dana) [2]. NIST je 2007. godine organizovao javno takmičenje za izbor nove familije algoritama koja će nositi ime SHA-3 (kandidat je trebalo da podrži heš izlaze od 224, 256, 384 i 512 bita). Od 64 pristiglih kandidata, *Keccak* algoritam je izabran za pobednika 2012. godine. *Fugue* heš algoritam, koji je predmet ovog rada, prošao je u drugu rundu ovog takmičenja, zajedno sa 13 drugih kandidata [1]. Kao i prethodna takmičenja, ovo takmičenje je donelo veliki korak napred u teorijskom i praktičnom radu u oblasti kriptografskih heš algoritama.

2.3. Primena

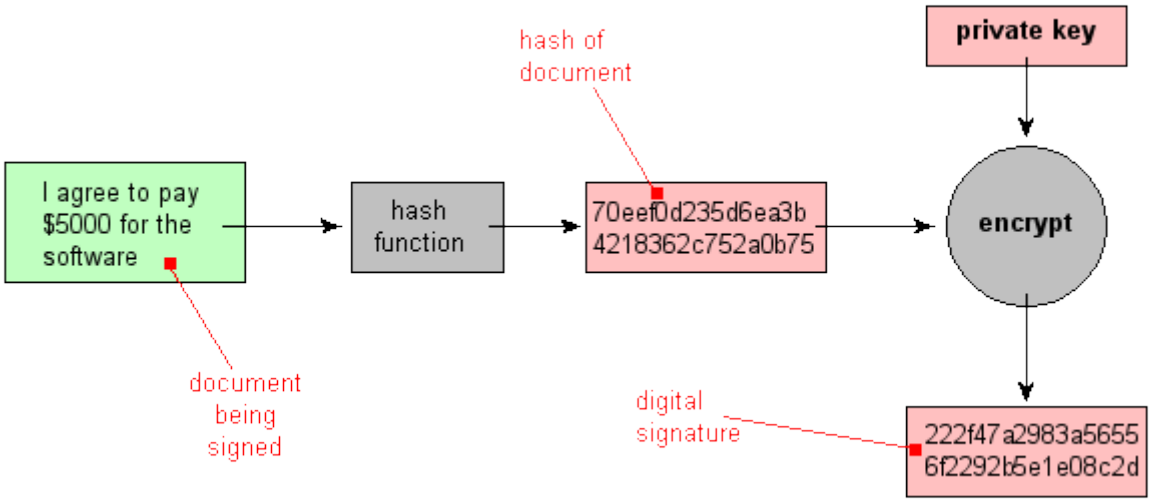
Heš algoritmi imaju široku primenu [3]:

- indeksiranje podataka u heš tabelama: Heš vrednost skladišti referencu na originalni podatak, čime se postiže konstantno vreme potrebno za pretragu, nezavisno od količine podataka koja se pretražuje.
- detektovanje dupliranih ili sličnih zapisa u velikoj datoteci
- pronalaženje dupliranih ili sličnih delova u okviru jednog fajla
- detektovanje sličnih segmenata u DNK sekvenci: Poznato je da DNK sekvence mogu biti veoma duge, pa se fiksna dužina heš vrednosti za poruku proizvoljne dužine ovde pokazuje kao veoma korisna osobina.
- geometrijsko heširanje: Nalazi primenu pri rešavanju problema u ravni i prostoru, u kompjuterskoj grafici, računarskoj geometriji, a u telekomunikacijama se koristi za kodiranje i kompresiju multidimenzionalnih signala.

Velika oblast u kojoj heš algoritmi takođe nalaze primenu je kriptografija. Kriptografske heš funkcije se koriste svuda gde postoji potreba da se korisnici osiguraju od falsifikovanja podataka od strane zlonamernih učesnika u komunikaciji. Neke od primena su [2]:

- verifikacija integriteta poruka i fajlova: Poređenjem heš vrednosti izračunatih pre i posle transmisije ili neke druge akcije, utvrđuje se da li je poruka ili fajl modifikovan.
- verifikacija korisničkih lozinki: Vrši se heširanje lozinke koju je korisnik ukucao sa namerom da se autentifikuje, zatim se dobijena heš vrednost poredi sa uskladištenom heš vrednosti. Ukoliko se vrednosti poklapaju, smatra se da je autentifikacija uspešna.

- digitalni potpis: Za označavanje stvaraoca dokumenta ili davanje saglasnosti sa sadržajem dokumenta u digitalnom svetu, koristi se digitalni potpis. Pomoću digitalnog potpisa proverava se da li je dokument zaista potpisala određena osoba. Za realizaciju digitalnog potpisa, heš funkcija se koristi u kombinaciji sa asimetričnom kriptografijom. Primer digitalnog potpisivanja dokumenta je prikazan na slici 2.3.1.



Slika 2.3.1.Primer digitalnog potpisivanja dokumenta [4]

Heš vrednost dokumenta se šifruje privatnim ključem osobe koja potpisuje dokument i rezultat predstavlja digitalni potpis. Kasnije se može dokazati da je baš ta osoba potpisala dokument dešifrovanjem dokumenta javnim ključem, čime se otkriva heš vrednost dokumenta. Poređenjem novodobijene heš vrednosti i heš vrednosti dokumenta, dobija se odgovor da li je ta osoba potpisala dokument [4].

- izvođenje novih ključeva za šifrovanje iz jednog, sigurnog ključa

2.4. *Fugue* algoritam

Fugue algoritam je, kao što je rečeno, bio jedan od kandidata takmičenja za izbor SHA-3 algoritma. Prva verzija algoritma objavljena je 2009. godine, dok je druga verzija, *Fugue* 2.0, nastala 2012. godine. Predmet ovog rada biće hardverska implementacija novije verzije algoritma. U nastavku ovog poglavlja biće opisani koraci algoritma [6].

Fugue 2.0 algoritam se može primeniti na poruke proizvoljne dužine, pri čemu je maksimalna dužina poruke 2^{64}-1 bita. Dužina poruke mora biti umnožak od 32 bita (32 bita predstavlja jedan blok poruke, reč). Ukoliko to nije slučaj, vrši se tzv. *padding* (dopunjavanje 0 bitima do dužine koja je umnožak od 32 bita). Nakon toga, originalna dužina poruke (pre eventualnog dopunjavanja) se predstavlja osmobajtnim brojem i dodaje se na kraj poruke. Struktura koja se koristi kao interno stanje je matrica čije kolone odgovaraju dimenziji bloka poruke. Broj kolona varira u zavisnosti od izabrane dužine heš vrednosti. Po jednoj iteraciji jedan blok poruke se ubacuje u stanje i nad stanjem se vrši tzv. *round* funkcija. Nakon što se cela poruka obradi, na stanje se primenjuje obimna finalna transformacija i deo novodobijenog stanja predstavlja heš vrednost.

Heš vrednost može biti dužine do 512 bita. Sledi uopšteni opis koraka ovog algoritma (primenljiv na sve dužine heš vrednosti), dok će detaljan opis koraka biti dat u narednom poglavlju.

Pre ubacivanja blokova poruke u stanje, potrebno je izvršiti inicijalizaciju stanja pomoću vektora inicijalizacije (*Initialization vector,* IV). Za svaku dužinu heš vrednosti postoji različiti IV. Stanje se inicijalizuje tako što se u određene kolone postavlja vrednost vektora IV, a vrednost ostalih kolona se postavlja na vrednost 0.

Nakon inicijalizovanja, može se početi sa ubacivanjem blokova poruke u stanje. Obrada koja, između ostalog, uključuje i ubacivanje pojedinačnog bloka u stanje naziva se *round* transformacija R (u daljem tekstu R transformacija). Koraci od kojih se sastoji se nazivaju TIX, ROR3, CMIX i SMIX.

U TIX koraku, blok poruke se ubacuje u jednu od kolona stanja i vrši se sabiranje odrgovarajućih kolona stanja. Kako se u ovom algoritmu sve operacije vrše u polju $GF(2^8)$, sabiranje je predstavljeno *xor* operacijom. Kako je u pitanju sabiranje četvorobajtnih vektora, ono je ekvivalentno *xor* operaciji nad 32 bita.

ROR3 korak vrši rotaciju kolona stanja za tri mesta udesno.

CMIX korak vrši *xor* operacije nad odgovarajućim kolonama.

SMIX korak predstavlja glavni korak u *Fugue* 2.0 algoritmu. Operiše sa prve četiri kolone stanja, koje se mogu posmatrati kao matrica *W* dimenzija 4×4. Prvo, svaki bajt (element matrice stanja je veličine jednog bajta) prolazi kroz S-box zamenu. S-box zamena je permutacija koja se sastoji od dva mapiranja i čiji rezultat se može predstaviti tabelom 2.4.1. Prva četiri bita sa leve strane bajta (*big endian* format) predstavljena su redovima, a druga četiri bita u kolonama, tako da je S-box[00]= 63, S-box[01]=7c, itd...

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
A	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
B	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
C	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
D	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
E	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
F	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Tabela 2.4.1. S-box tabela permutacije bajtova [5]

Nakon S-box zamene, matrica *W* prolazi kroz Super-Mix linearnu transformaciju. Super-Mix transformacija se može predstaviti stavljanjem elemenata matrice *W* u vektor kolonu i množenjem sleva matricom *N* koja je dimenzija 16×16 bajtova. Bajt *i*-te vrste i *j*-te kolone u matrici *W* biće (i+4·j)-ti bajt u vektor koloni. Matrica *N* je data jednačinom 2.4.1.

$$N = \begin{pmatrix}
1 & 4 & 7 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 1 & 1 & 4 & 7 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 7 & 1 & 1 & 4 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 4 & 7 & 1 & 1 \\
\\
0 & 0 & 0 & 0 & 0 & 4 & 7 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 4 & 7 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 7 & 1 & 0 & 4 \\
4 & 7 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\\
0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 6 & 4 & 7 & 1 & 7 & 0 & 0 & 0 \\
0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 & 0 & 1 & 6 & 4 & 7 \\
7 & 1 & 6 & 4 & 0 & 0 & 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 7 & 0 \\
0 & 0 & 0 & 7 & 4 & 7 & 1 & 6 & 0 & 0 & 0 & 7 & 0 & 0 & 0 & 0 \\
\\
0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 5 & 4 & 7 & 1 \\
1 & 5 & 4 & 7 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 & 0 & 0 \\
0 & 0 & 4 & 0 & 7 & 1 & 5 & 4 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 \\
0 & 0 & 0 & 4 & 0 & 0 & 0 & 4 & 4 & 7 & 1 & 5 & 0 & 0 & 0 & 0
\end{pmatrix} \tag{2.4.1}$$

Niz od tri koraka, ROR3, CMIX i SMIX, čini podrundu.

Nakon primene R transformacije za svaki primljeni blok poruke, stanje prolazi kroz finalnu G rundu. Ona se sastoji iz dve faze, G_1 i G_2.

Faza G_1 se sastoji od odgovarajućeg broja podrundi koje se primenjuju na stanje.

U fazi G_2 primenjuje se odgovarajući broj sledećeg skupa operacija: *xor* operacije nad određenim kolonama, rotiranje udesno za odgovarajući broj mesta i SMIX korak. Nakon toga, obavljaju se *xor* operacije nad određenim kolonama stanja.

Odgovarajući broj kolona novodobijenog stanja predstavlja izlaz *Fugue* 2.0 algoritma, tj. heš vrednost.

3. IMPLEMENTACIJA *FUGUE* 2.0 ALGORITMA

U ovom poglavlju biće opisan programski kod napisan za potrebe implementacije algoritma. Kod je pisan u VHDL programskom jeziku. Detaljan opis sledi za verziju čija je heš vrednost dužine 256 bita (*Fugue* 2.0-256), a potom će biti navedene razlike u delovima koda za *Fugue* 2.0-224, *Fugue* 2.0-384 i *Fugue* 2.0-512. Informacija o razlikama u koracima za različite dužine heš izlaza dobijena je iz originalnog dokumenta [6].

Pretpostavka je da poruka stiže sa već urađenom dopunom i dodatih 8 bajtova koji predstavljaju dužinu originalne poruke.

Dizajn se najpre može predstaviti crnom kutijom sa interfejsima.

3.1. Interfejsi

Dizajn sadrži ulazne i izlazne interfejse. Ulazni interfejsi su interfejsi sa imenom *clk, reset, prvi, poslednji* i *rec*. Izlazni interfejsi dizajna su oni sa imenom *preuzmi, cekaj* i *izlaz*.

Signal *clk* je korišćen kao signal takta. Uzlazna ivica ovog signala koristi se kao okidač za izvršavanje određenog koda, u zavisnosti od faze obrade. U svakom taktu prima se jedan blok poruke i vrši se R transformacija. G runda se izvršava u onoliko taktova, koliko iteracija postoji.

Signal *reset* postavlja dizajn u neaktivno stanje, tj. stanje u kojem se čeka prijem poruke. Izlazni signali *preuzmi* i *cekaj*, koji će biti objašnjeni nešto kasnije, postavljaju se na neaktivnu vrednost. Signal *izlaz*, koji predstavlja heš vrednost, postavlja se na nultu vrednost.

Signal *prvi* signalizira da je na ulazu prisutan prvi blok poruke i predstavlja okidač za prelazak iz neaktivnog u aktivno stanje, tj. stanje u kome se vrši obrada.

Signalom *poslednji* signalizira se da je na ulazu prisutan poslednji blok poruke. Ovaj signal predstavlja i okidač za prelazak u sledeću fazu obrade (nakon izvršene R transformacije, prelazi se na izvršavanje G runde).

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

Signal *cekaj* označava da je prijem cele poruke završen i da je u toku izvršavanje G runde.

Signalom *preuzmi* signalizira se da je heš vrednost izračunata i da trenutna vrednost signala *izlaz* predstavlja heš vrednost.

Kao što je rečeno, signal *izlaz* po završetku obrade predstavlja heš vrednost ulazne poruke. Dok je obrada u toku ili se čeka prijem poruke, signal ima nultu vrednost.

Funkcija crne kutije je izračunavanje heš vrednosti primenom *Fugue* 2.0 algoritma na primljenu poruku. Zavisno od vrednosti signala sa ulaznih interfejsa, menjaju se i vrednosti internih signala. Dalje, pojedini interni signali predstavljaju okidače za promenu vrednosti drugih internih signala. Uloga internih signala je signaliziranje u kojoj fazi obrade se nalazi dizajn, tako da se u odnosu na te vrednosti primenjuje odgovarajuća faza obrade poruke (inicijalizacija stanja, R transformacija, G runda, postavljanje heš vrednosti kao vrednost signala *izlaz*). Jedan od internih signala predstavlja stanje čiji je sadržaj rezultat trenutne obrade. Sledi opis crne kutije.

3.2. Unutrašnjost crne kutije

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

3.2.1. 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:

- *stanje_niz*: Tip promenljive koja predstavlja interno stanje. Autori algoritma su interno stanje definisali kao matricu, ali pošto je u VHDL programskom jeziku komplikovano raditi sa matricama, interno stanje je definisano kao niz bajtova (bajt odgovara elementu matrice). Matrica je dimenzija 4×30, tako da se niz sastoji iz 120 bajtova.
- *s_box_tip*: Tip promenljive koja predstavlja S-box tabelu. S-box tabela predstavlja preslikavanje 256 različitih bajtova, pa je ovaj tip definisan kao niz od 256 bajtova.
- *N_tip*: Tip promenljive koja predstavlja matricu *N*. Matrica *N* je dimenzija 16×16, a elementi su bajtovi. Kako je poželjno raditi sa nizovima, a ne matricama, ovaj tip je definisan kao niz od 256 bajtova.
- *kolona_tip*: Tip promenljive koja predstavlja 4 kolone matrice smeštene u jednu vektor kolonu. Kako kolonu matrice čini 4 elemenata, ovaj tip je definisan kao niz od 16 bajtova.
- *konacni_automat:* Tip promenljive koja predstavlja stanje konačnog automata. Ovaj enumerisani tip sadrži logičke nazive za stanja automata. Detaljan opis i razlog upotrebe konačnog automata u ovom dizajnu sledi u narednom odeljku.

3.2.2. Konačni automat

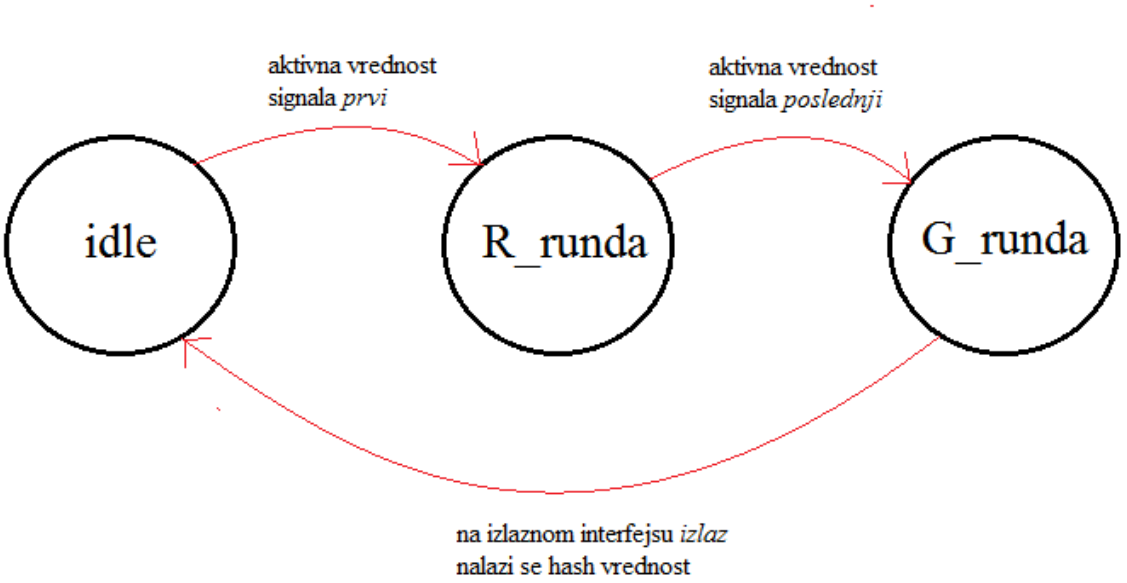
Na osnovu trenutne vrednosti promenljive koja predstavlja stanje konačnog automata, primenjuje se odgovarajući kod, tj. odgovarajuća faza u obradi poruke. Stanja koja su definisana su: *idle, R_runda* i *G_runda*.

Stanje *idle* je neaktivno stanje, tj. stanje u kome se nalazi dizajn pre prijema bilo kakve poruke. U ovom stanju se vrši inicijalizacija stanja.

Kada počne prijem poruke, dizajn prelazi u stanje *R_runda* i za svaki primljeni blok poruke primenjuje se R transformacija nad stanjem. Dizajn ostaje u ovom stanju dok se ne izvrši R transformacija nad poslednjim blokom poruke.

Potom, dizajn prelazi u stanje *G_runda* u kome stanje prolazi kroz G rundu. Ovo stanje se zadržava sve dok se izračunata heš vrednost ne postavi kao vrednost signala *izlaz*. Nakon toga, dizajn se vraća u stanje *idle* u kome se čeka prijem nove poruke.

Slika 3.2.2.1. prikazuje stanja konačnog automata i signale koji utiču na prelazak iz jednog u drugo stanje.



Slika 3.2.2.1. Stanja konačnog automata i uslovi prelaska između stanja

Napomena: Uslovi prelaska između stanja su opisani za slučaj kada se poruka sastoji iz više blokova. Ukoliko poruku čini jedan blok, iz stanja *idle* se prelazi direktno u stanje *G_runda*, jer se u jednom taktu izvršava i prijem bloka i R transformacija.

3.3. Opis algoritma

Najpre će biti izlistane i objašnjene funkcije i procedure koje predstavljaju korake algoritma (TIX, CMIX, SMIX,...), a potom će biti opisan kod koji prati celokupan tok obrade poruke. Funkcije, procedure, kao i pojedine konstante korišćene pri implementaciji ovog algoritma definisane su u pomenutom paketu.

3.3.1. Opis funkcija i procedura

Funkcija *f_inicijalizacija* vrši inicijalizovanje stanja pomoću vektora IV. Ulazni argument je IV, a izlazni argument je stanje. IV je definisan kao konstanta u pomenutom paketu i predstavljen je tipom *std_logic_vector* od 256 elemenata. Inicijalizacijom se prve 22 kolone matrice stanja postavljaju na nultu vrednost, a ostale kolone (od 22. do 29. kolone) se pune vrednostima sadržanim u vektoru IV. Pošto je stanje implementirano kao niz, a ne matrica, element stanja sa indeksom (*i**30+*j*) odgovara elementu matrice u *i*-toj vrsti i *j*-toj koloni. Radi dodele vrednosti elementima se pristupa pomoću petlji (jedna petlja kontroliše broj vrste, a druga kontroliše broj kolone). Kod kojim se vrši postavljanje na nultu vrednost je sledeći:

```
FOR i IN 0 TO 3 LOOP
      FOR j IN 0 TO 21 LOOP
            stanje (i*30+j):= X"00";
      END LOOP;
END LOOP;
```

Svakom elementu, počevši od 22. kolone, pa do 29. kolone, dodeljuje se vrednost 8 krajnje levih bita vektora IV. To se postiže tako što se, nakon svake dodele, sadržaj varijable *iv_var* koja inicijalno ima vrednost vektora IV, pomera za 8 mesta ulevo (upražnjena mesta se popunjavaju nulama):

```
FOR j IN 22 TO 29 LOOP
      FOR i IN 0 TO 3 LOOP
            stanje (i*30+j):= iv_var(255 DOWNTO 248);
            iv_var:= SHL(iv_var, "1000");
      END LOOP;
END LOOP;
```

Ovim se izbegava računanje odgovarajućih indeksa varijable *iv_var* pri svakoj novoj dodeli.

Procedura *tix* izvršava TIX korak algoritma. Stanje je ulazno/izlazni argument, a blok poruke (signal *rec*) je ulazni argument. Koraci koje je potrebno izvršiti su dati jednačinama 3.3.1-5:

$$\begin{aligned} S_4 &+= S_0 \\ S_0 &= \text{blok poruke} \\ S_{14} &+= S_0 \\ S_{20} &+= S_0 \\ S_8 &+= S_1 \end{aligned} \qquad (3.3.1\text{-}5)$$

Napomena: Sa S_0 označena je nulta kolona stanja, S_1 označava prvu kolonu stanja, itd... Izraz $S_4+=S_0$ je ekvivalentan izrazu $S_4=S_4+S_0$. Na postojeću vrednost 4. kolone, dodaje se vrednost dobijena *xor*-ovanjem vrednosti 4. i nulte kolone.

U varijablu *rec_var* smešta se sadržaj bloka poruke. Elementima se pristupa pomoću jedne petlje koja kontroliše broj vrste, a broj kolone je unapred određen. Izraz $S_4+=S_0$ je realizovan sledećim kodom:

```
FOR i IN 0 TO 3 LOOP
      stanje(i*30+4):= stanje(i*30+4)XOR stanje(i*30+0);
END LOOP;
```

Analogno su realizovani ostali izrazi u kojima se vrši *xor* operacija.

Svakom elementu nulte kolone dodaje se 8 krajnje levih bita bloka poruke. Nakon svake dodele, sadržaj varijable *rec_var* se pomera za osam mesta ulevo, a upražnjena mesta se popunjavaju nulama:

```
FOR i IN 0 TO 3 LOOP
      stanje(i*30+0):=rec_var(31 DOWNTO 24);
      rec_var:= SHL(rec_var,"1000");
END LOOP;
```

Ovim je, kao i u prethodnoj funkciji, izbegnuto računanje odgovarajućih indeksa prilikom svake nove dodele.

Procedura *ror3* vrši rotiranje kolona stanja za 3 mesta udesno. Stanje je ulazno/izlazni argument. U varijablu *novo_stanje* smešta se sadržaj stanja, pomeren za tri kolone udesno: treća kolona novog stanja dobija sadržaj nulte kolone stanja, četvrta kolona novog stanja dobija sadržaj prve kolone, itd. Ponovo, elementima stanja se pristupa pomoću dve petlje. Jedna petlja kontroliše broj vrste, a druga petlja broj kolone. Kod koji izvršava rotiranje je sledeći:

```
FOR i IN 0 TO 3 LOOP
      FOR j IN 29 DOWNTO 3 LOOP
            novo_stanje(i*30+j):= stanje(i*30+j-3);
      END LOOP;
      FOR j IN 2 DOWNTO 0 LOOP
            novo_stanje(i*30+j):= stanje(i*30+j+27);
      END LOOP;
END LOOP;
```

Procedura *cmix* izvršava CMIX korak algoritma. Ulazno/izlazni argument ove procedure jeste stanje. Koraci koje je potrebno izvršiti su dati jednačinama 3.3.6-11:

$$\begin{aligned} S_0 &+= S_4 \\ S_1 &+= S_5 \\ S_2 &+= S_6 \\ S_{15} &+= S_4 \\ S_{16} &+= S_5 \\ S_{17} &+= S_6 \end{aligned} \qquad (3.3.6\text{-}11)$$

Kod kojim se ovo realizuje je sledeći:

```
FOR i IN 0 TO 3 LOOP
      stanje(i*30+0):= stanje(i*30+0)XOR stanje(i*30+4);
      stanje(i*30+1):= stanje(i*30+1)XOR stanje(i*30+5);
      stanje(i*30+2):= stanje(i*30+2)XOR stanje(i*30+6);
      stanje(i*30+15):= stanje(i*30+15)XOR stanje(i*30+4);
      stanje(i*30+16):= stanje(i*30+16)XOR stanje(i*30+5);
      stanje(i*30+17):= stanje(i*30+17)XOR stanje(i*30+6);
END LOOP;
```

Elementima stanja se pristupa pomoću jedne petlje koja kontroliše broj vrste, dok su brojevi kolona unapred određeni.

Procedura *s_box_zamena* izvršava S-box zamenu jednog elementa stanja i koristiće se u okviru procedure koja vrši S-box zamenu bajtova prve četiri kolone stanja. Element stanja je ulazno/izlazni argument. U okviru procedure se koristi i konstanta *s_box* koja je definisana u paketu. Kako S-box tabela ima 16 vrsti i 16 kolona, a u ovom radu je definisana kao niz od 256 bajtova, elementu tabele u *i*-toj vrsti i *j*-toj koloni odgovara element niza sa indeksom (*i**16+*j*). S-box tabela je definisana tako da se bajt 00 preslikava u bajt koji se nalazi u nultoj vrsti i nultoj koloni tabele. Analogno, bajt 10 se preslikava u bajt koji se nalazi u 1. vrsti i nultoj koloni, itd... U ovom kodu, bajt 00 se preslikava u nulti član niza *s_box*, bajt 10 se preslikava u 16. član niza *s_box*, itd... Kod kojim se vrši preslikavanje je sledeći:

```
indeks:=conv_integer(element_stanja);
element_stanja:= s_box(indeks);
```

Procedura *s_box_zamena_4x4* izvršava S-box zamenu bajtova prve četiri kolone stanja. Stanje predstavlja ulazno/izlazni argument. Procedura *s_box_zamena* se poziva u okviru petlji. Jedna petlja kontroliše broj vrste, a druga petlja broj kolone elementa koji prolazi kroz S-box zamenu. Kod kojim se ovo izvršava je sledeći:

```
FOR i IN 0 TO 3 LOOP
```

```
        FOR j IN 0 TO 3 LOOP
                s_box_zamena(stanje(i*30+j));
        END LOOP;
END LOOP;
```

Procedura *mnozenje* vrši množenje dva bajta, koji u polju GF(2^8) predstavljaju polinome do sedmog stepena. Primera radi, bajt 10010110 predstavlja polinom $x^7+x^4+x^2+x$. Za rezultat množenja u ovom polju nije dovoljno pomnožiti dva polinoma, već dobijeni proizvod treba podeliti ireducibilnim polinomom $x^8+x^4+x^3+x+1$. Dobijeni ostatak predstavlja rezultat množenja. Procedura se koristi u SMIX koraku, kada je potrebno pomnožiti matricu *N* i vektor kolonu.Ulazni argumenti su dva bajta: element matrice *N* i element vektor kolone. Izlazni argument je ostatak pri deljenju proizvoda ova dva elementa, takođe u obliku bajta. Ideja za kod koji izvršava množenje preuzeta je sa stranice sajta *Wikipedia*, čija su tema operacije u konačnim poljima [7]. Kod je prilagođen za potrebe ovog rada na sledeći način:

```
proizvod:=X"00";
a:=bajt1;
b:=bajt2;
FOR i IN 0 to 7 LOOP
        IF(b(0)='1')THEN
                proizvod:=proizvod XOR a;
        END IF;
        b:= '0' & b(7 DOWNTO 1);
        IF(a(7)='1')THEN
                a:=(a(6 DOWNTO 0)&'0') XOR X"1B";
        ELSE
                a:=(a(6 DOWNTO 0)&'0');
        END IF;
END LOOP;
izlaz:=proizvod;
```

Varijable definisane unutar procedure su *proizvod*, *a* i *b* i predstavljene su kao bajtovi. U varijablu *a* se smešta vrednost elementa matrice *N*, a u varijablu *b* vrednost elementa vektor kolone. Inicijalna vrednost varijable *proizvod* je 0. Postoji 8 iteracija, da bi se obuhvatili svi bitovi jednog bajta. U svakoj iteraciji se proverava da li je najniži bit promenljive *b* jednak 1 i ako jeste vrši se množenje (rezultat *xor* operacije nad prethodnim proizvodom i promenljivom *a* postaje nova vrednost proizvoda). Zbog tako realizovanog množenja, u svakoj iteraciji sadržaj promenljive *b* pomera se za jedno mesto udesno, a sadržaj promenljive *a* za jedno mesto ulevo. Ukoliko se pređe opseg u polju (uslov *a*(7)=’1’), tada se vrši *xor* operacija nad *a* i bajtom “1B“ (koji predstavlja polinom x^4+x^3+x+1). Ovim je osigurano da vrednost promenljive *proizvod* ne pređe opseg u polju, tako da ona predstavlja krajnji rezultat množenja u polju GF(2^8).

Procedura *super_mix* izvršava deo SMIX koraka nakon S-box zamene, a to je množenje matrice *N* i vektor kolone. Ulazno/izlazni argument je stanje. Najpre se iz stanja izdvajaju elementi prve četiri kolone i smeštaju u vektor kolonu. U tu svrhu, definisana je varijabla *kolona*, za koju se koristi korisnički definisan tip podataka *kolona_tip*. Kod koji vrši navedeno izdvajanje je sledeći:

```
FOR i IN 0 TO 3 LOOP
        FOR j IN 0 TO 3 LOOP
                kolona(i+4*j):= stanje(i*30+j);
        END LOOP;
END LOOP;
```

Petljama se kontrolišu indeksi elemenata u stanju i vektor koloni (kako je ranije navedeno, element stanja u *i*-toj vrsti i *j*-toj koloni biće ($i+4\cdot j$)-ti element u vektor koloni).

Kod koji vrši množenje matrica i rezultat vraća u prve četiri kolone stanja je sledeći:

```
vrsta_stanja=0;
kolona_stanja=0;
FOR i IN 0 TO 15 LOOP
      suma:="00000000";
      FOR j IN 0 TO 15 LOOP
            mnozenje(N(i*16+j), kolona(j), rezultat_mnozenja);
            suma:= suma XOR rezultat_mnozenja;
      END LOOP;
      stanje(vrsta_stanja*30+ kolona_stanja):= suma;
      IF (vrsta_stanja=3) THEN
            vrsta_stanja:=0;
            IF (kolona_stanja<3) THEN
                  kolona_stanja:=kolona_stanja+1;
            END IF;
      ELSE
            vrsta_stanja:=vrsta_stanja+1;
      END IF;
END LOOP;
```

Varijable *vrsta_stanja* i *kolona_stanja* inicijalno imaju nultu vrednost, a predstavljaju indekse koji određuju na koja mesta u stanju će se smeštati rezultat množenja dveju matrica. Unutar petlje koju kontroliše promenljiva *i*, a koja predstavlja broj vrste matrice *N*, nalazi se sledeći kod:

```
suma:="00000000";
FOR j IN 0 TO 15 LOOP
      mnozenje(N(i*16+j), kolona(j), rezultat_mnozenja);
      suma:= suma XOR rezultat_mnozenja;
END LOOP;
stanje(vrsta_stanja*30+ kolona_stanja):= suma;
```

Varijabla *suma* na početku množenja svake vrste matricom kolonom ima nultu vrednost. U svakoj iteraciji petlje čiji brojač (promenljiva *j*) predstavlja broj elementa u vektor koloni, vrši se množenje elementa matrice *N* i elementa vektor kolone. Rezultat množenja se u svakoj iteraciji dodaje sumi, da bi nakon završetka petlje odgovarajućem elementu stanja bila dodeljena vrednost sume, koja predstavlja proizvod jedne vrste matrice *N* i vektor kolone. Vrednosti varijabli *vrsta_stanja* i *kolona_stanja* kontrolišu se sledećim kodom:

```
IF(vrsta_stanja=3)THEN
      vrsta_stanja:=0;
      IF(kolona_stanja<3)THEN
            kolona_stanja:=kolona_stanja+1;
      END IF;
ELSE
      vrsta_stanja:=vrsta_stanja+1;
END IF;
```

Nakon što je završeno množenje jedne vrste matrice *N* vektor kolonom i rezultat smešten u element stanja, ispituje se da li je napunjena cela kolona stanja. Ako jeste, prelazi se na sledeću kolonu stanja, a ako nije, povećava se broj vrste.

Procedure *s_box_zamena_4x4* i *super_mix*, pozvane tim redosledom, izvršavaju SMIX korak algoritma.

Procedura *R_transformacija*, kao što joj ime kaže, vrši R transformaciju. Pošto se ona vrši za svaki primljeni blok poruke, blok poruke je ulazni argument (signal *rec*), dok je stanje

ulazno/izlazni argument. Kako se R transformacija sastoji od koraka TIX, ROR3, CMIX, SMIX, tako se i procedure koje izvršavaju ove korake pozivaju u okviru procedure *R_transformacija*, kodom:

```
tix(rec, stanje);
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Procedura pod nazivom *G_runda_prva_petlja* se poziva pri izvršavanju G_1 faze finalne G runde. G_1 faza se sastoji od 26 podrundi (podsećanja radi, podrundu čini niz koraka: ROR3, CMIX i SMIX). Ova procedura izvršava jednu podrundu, tako da se u arhitekturi top-level entiteta poziva u okviru petlje koja ima 26 iteracija. Ulazno/izlazni argument je stanje. Kod koji izvršava podrundu je sledeći:

```
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Procedura *ror15* izvršava korak ROR15, koji predstavlja rotiranje kolona stanja za 15 mesta udesno. ROR15 se izvršava u okviru G_2 faze finalne G runde. Ulazno/izlazni argument procedure je stanje. Procedura je analogna proceduri *ror3*, jedinu razliku predstavlja broj koji se pojavljuje u petlji i indeksu. Kod koji vrši rotiranje je sledeći:

```
FOR i IN 0 TO 3 LOOP
      FOR j IN 29 DOWNTO 15 LOOP
            novo_stanje(i*30+j):= stanje(i*30+j-15);
      END LOOP;
      FOR j IN 14 DOWNTO 0 LOOP
            novo_stanje(i*30+j):= stanje(i*30+j+15);
      END LOOP;
END LOOP;
```

Procedura *ror14* izvršava korak ROR14, odnosno rotiranje za 14 mesta udesno. Ovaj korak se izvršava u okviru G_2 faze finalne G runde. Stanje je ulazno/izlazni argument. Procedura je analogna procedurama *ror3* i *ror15*, sa razlikama u brojevima u petlji i indeksu:

```
FOR i IN 0 TO 3 LOOP
      FOR j IN 29 DOWNTO 14 LOOP
            novo_stanje(i*30+j):= stanje(i*30+j-14);
      END LOOP;
      FOR j IN 13 DOWNTO 0 LOOP
            novo_stanje(i*30+j):= stanje(i*30+j+16);
      END LOOP;
END LOOP;
```

Procedura *G_runda_druga_petlja* se poziva pri izvršavanju G_2 faze finalne G runde. G_2 faza se delom sastoji od 13 iteracija, pri čemu se u svakoj iteraciji izvršavaju sledeći koraci: *xor* operacije prikazane jednačinama 3.3.12-13, nakon kojih slede koraci ROR15 i SMIX, potom *xor* operacije prikazane jednačinama 3.3.14-15 i konačno, koraci ROR14 i SMIX.

$$S_4 \mathrel{+}= S_0 \qquad (3.3.12\text{-}13)$$
$$S_{15} \mathrel{+}= S_0$$

$$S_4 \mathrel{+}= S_0 \qquad (3.3.14\text{-}15)$$
$$S_{16} \mathrel{+}= S_0$$

Ovaj deo G_2 faze se izvršava pozivanjem procedure *G_runda_druga_petlja* u arhitekturi top-level entiteta u okviru petlje koja ima 13 iteracija. Ulazno/izlazni argument procedure je stanje. Kod koji izvršava procedura je sledeći:

```
FOR i IN 0 TO 3 LOOP
      stanje(i*30+4):= stanje(i*30+4) XOR stanje(i*30+0);
      stanje(i*30+15):= stanje(i*30+15) XOR stanje(i*30+0);
END LOOP;
ror15(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
FOR i IN 0 TO 3 LOOP
      stanje(i*30+4):= stanje(i*30+4) XOR stanje(i*30+0);
      stanje(i*30+16):= stanje(i*30+16) XOR stanje(i*30+0);
END LOOP;
ror14(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Preostali deo G_2 faze čine sledeći koraci, prikazani jednačinama 3.3.16-17:

$$S_4 \mathrel{+}= S_0 \qquad (3.3.16\text{-}17)$$
$$S_{15} \mathrel{+}= S_0$$

Pošto kod koji izvršava ove korake nije napisan u okviru funkcije ili procedure, nego se nalazi u okviru arhtekture top-level entiteta, biće dat u sledećem odeljku, pri opisu koda u arhitekturi top-level entiteta.

Procedura *rezultat* se koristi da iz stanja izdvoji odgovarajuće kolone koje predstavljaju heš vrednost. Ulazni argument procedure je stanje. Kako se u ovom potpoglavlju opisuje kod za *Fugue* 2.0 algoritam sa heš vrednošću dužine 256 bita, izlazni argument je signal *izlaz* koji predstavlja niz od 256 bita. Kolone koje treba izdvojiti su sledeće i poređane ovim redosledom predstavljaju heš vrednost algoritma:

S_1 S_2 S_3 S_4 S_{15} S_{16} S_{17} S_{18}

Signal *izlaz* je predstavljen u *big endian* formatu, tako da se sadržaj kolona smešta u signal *izlaz* počevši od indeksa 255, ka nižim indeksima. Tako je prvi bajt 1. kolone stanja predstavljen članovima niza *izlaz* sa indeksima od 255 do 248, dok je poslednji element 18. kolone stanja predstavljen članovima niza sa indeksima od 7 do 0. Varijabla *brojac* broji koliko je elemenata prebačeno u niz *izlaz* (osam kolona sadrži 32 elementa). Na osnovu njene vrednosti računaju se indeksi niza *izlaz*, tj. mesto u nizu gde se smešta sledeći element stanja. Petljama se kontroliše broj vrste i kolone stanja (*k* i *j* kontrolišu broj kolone, a *i* kontroliše broj vrste). Kod kojim se vrši izdvajanje heš vrednosti je sledeći:

```
FOR k IN 18 DOWNTO 15 LOOP
      FOR i IN 3 DOWNTO 0 LOOP
            izlaz ((brojac*8 + 7) DOWNTO (brojac*8))<= stanje(i*30+k);
            brojac:= brojac+1;
      END LOOP;
END LOOP;
```

```
FOR j IN 4 DOWNTO 1 LOOP
      FOR i IN 3 DOWNTO 0 LOOP
            izlaz ((brojac*8 + 7) DOWNTO (brojac*8))<= stanje(i*30+j);
            brojac:= brojac+1;
      END LOOP;
END LOOP;
```

3.3.2. Opis rada top-level entiteta

Nakon što su objašnjene funkcije i 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 shema OF Fugue_256 IS
      SIGNAL stanje_automata: konacni_automat;
      SIGNAL brojac_prva_petlja: INTEGER;
      SIGNAL brojac_druga_petlja: INTEGER;
BEGIN
      PROCESS(clk)
            VARIABLE stanje_var: stanje_niz;
      BEGIN
            IF (reset='1')THEN
                  stanje_automata<= idle;
                  preuzmi<='0';
                  cekaj<='0';
                  izlaz<=(others=>'0');
            ELSIF (clk'EVENT AND clk='1' ) THEN
                  CASE (stanje_automata) IS
                        WHEN idle =>
                              stanje_var:= f_inicijalizacija(iv);
                              IF (prvi='1') THEN
                                    R_transformacija(rec, stanje_var);
                                    IF(poslednji='1')THEN
                                          stanje_automata<=G_runda;
                                          brojac_prva_petlja<=0;
                                          brojac_druga_petlja<=0;
                                          cekaj<='1';
                                    ELSE
                                          stanje_automata<=R_runda;
                                    END IF;
                              END IF;
                              preuzmi<='0';
                        WHEN R_runda =>
                              R_transformacija(rec, stanje_var);
                              IF (poslednji='1')THEN
                                    stanje_automata<=G_runda;
                                    brojac_prva_petlja<=0;
                                    brojac_druga_petlja<=0;
                                    cekaj<='1';
                              END IF;
                        WHEN G_runda =>
                              IF (brojac_prva_petlja=26) THEN
                                    IF (brojac_druga_petlja<13) THEN
                                          G_runda_druga_petlja(stanje_var);

      brojac_druga_petlja<=brojac_druga_petlja+1;
                                    END IF;
                                    IF (brojac_druga_petlja=13) THEN
```

```
                                        FOR i IN 0 TO 3 LOOP
                                              stanje_var(i*30+4):=
stanje_var(i*30+4) XOR stanje_var(i*30+0);
                                              stanje_var(i*30+15):=
stanje_var(i*30+15) XOR stanje_var(i*30+0);
                                        END LOOP;
                                        rezultat(stanje_var,izlaz);
                                        preuzmi<='1';
                                        cekaj<='0';
                                        stanje_automata<= idle;
                                  END IF;
                            END IF;
                            IF (brojac_prva_petlja<26) THEN
                                  G_runda_prva_petlja(stanje_var);
                                  brojac_prva_petlja<=brojac_prva_petlja+1;
                            END IF;
                  END CASE;
        END IF;
    END PROCESS;
END shema;
```

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 određeni signali postavljaju na inicijalne vrednosti (dalje u odeljku sledi detaljniji opis koraka koji se preduzimaju kada je aktivan signal za reset).

Signali definisani unutar arhitekture su: *stanje_automata*, *brojac_prva_petlja* i *brojac_druga_petlja*.

Signal *stanje_automata* je tipa *konacni_automat* i njegova trenutna vrednost predstavlja fazu obrade u kojoj se dizajn nalazi (*idle*, *R_runda*, *G_runda*).

Signali *brojac_prva_petlja* i *brojac_druga_petlja* su tipa *integer* i kontrolišu broj iteracija u petljama korišćenim u G_1 i G_2 fazi finalne G runde.

U deklarativnom delu procesa definisana je varijabla *stanje_var*, tipa *stanje_niz* iz razloga što su za većinu procedura argumenti varijable.

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

```
stanje automata<= idle;
preuzmi<='0';
cekaj<='0';
izlaz<=(others=>'0');
```

Nakon reseta, dizajn se prebacuje u stanje *idle*, tj. stanje u kome se čeka prijem poruke. Ukoliko je u trenutku reseta obrada bila u toku, ona će se prekinuti. Signal *izlaz* se postavlja na nultu vrednost, kao i signali *preuzmi* i *cekaj*.

Ukoliko je signal za reset neaktivan, u zavisnosti od vrednosti signala *stanje_automata*, 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 *stanje_automata*.

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

```
stanje_var:= f_inicijalizacija(iv);
```

```
IF (prvi='1') THEN
      R_transformacija(rec, stanje_var);
      IF(poslednji='1')THEN
            stanje_automata<=G_runda;
            brojac_prva_petlja<=0;
            brojac_druga_petlja<=0;
            cekaj<='1';
      ELSE
            stanje_automata<=R_runda;
      END IF;
END IF;
preuzmi<='0';
```

Stanje se inicijalizuje i sadržaj se smešta u varijablu *stanje_var*. Ukoliko je na ulazu prisutan prvi blok poruke (aktivna vrednost signala *prvi*), poziva se procedura *R_transformacija* čiji je ulazni argument blok poruke prisutan na ulazu (signal *rec*). Ako je taj blok ujedno i poslednji blok (poruka se sastoji iz jednog bloka), nakon R transformacije dizajn prelazi u stanje *G_runda*, a brojači koji se koriste u finalnoj G rundi (*brojac_prva_petlja* i *brojac_druga_petlja*) postavljaju se na nultu vrednost. Takođe, izlazni signal *cekaj* se postavlja na aktivnu vrednost. Ukoliko se poruka sastoji iz više blokova, dizajn prelazi iz stanja *idle* u stanje *R_runda*, tako da će pri prijemu sledećeg bloka biti u odgovarajućem stanju.

Ako je dizajn u stanju *R_runda*, izvršava se sledeći kod:

```
R_transformacija(rec, stanje_var);
IF (poslednji='1')THEN
      stanje_automata<=G_runda;
      brojac_prva_petlja<=0;
      brojac_druga_petlja<=0;
      cekaj<='1';
END IF;
```

Ulazni argument za proceduru *R_transformacija* predstavlja blok koji je trenutno na ulazu. Ukoliko je taj blok i poslednji, nakon izvršene R transformacije dizajn prelazi u stanje *G_runda*, a već pomenuti brojači postavljaju se na nultu vrednost. Takođe, aktivira se signal *cekaj*.

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

```
IF (brojac_prva_petlja=26) THEN
      IF (brojac_druga_petlja<13) THEN
            G_runda_druga_petlja(stanje_var);
            brojac_druga_petlja<=brojac_druga_petlja+1;
      END IF;
      IF (brojac_druga_petlja=13) THEN
            FOR i IN 0 TO 3 LOOP
                  stanje_var(i*30+4):= stanje_var(i*30+4) XOR
stanje_var(i*30+0);
                  stanje_var(i*30+15):= stanje_var(i*30+15) XOR
stanje_var(i*30+0);
            END LOOP;
            rezultat(stanje_var,izlaz);
            preuzmi<='1';
            cekaj<='0';
            stanje_automata<= idle;
      END IF;
END IF;
IF (brojac_prva_petlja<26) THEN
      G_runda_prva_petlja(stanje_var);
```

```
      brojac_prva_petlja<=brojac_prva_petlja+1;
END IF;
```

Sledeći deo koda izvršava G_1 fazu finalne G runde (izvršavanje 26 iteracija je osigurano uslovom u okviru IF naredbe i inkrementiranjem promenljive *brojac_prva_petlja*):

```
IF (brojac_prva_petlja<26) THEN
      G_runda_prva_petlja(stanje_var);
      brojac_prva_petlja<=brojac_prva_petlja+1;
END IF;
```

Deo koda koji vrši G_2 fazu obrade u G rundi je sledeći:

```
IF (brojac_prva_petlja=26) THEN
      IF (brojac_druga_petlja<13) THEN
            G_runda_druga_petlja(stanje_var);
            brojac_druga_petlja<=brojac_druga_petlja+1;
      END IF;
      IF (brojac_druga_petlja=13) THEN
            FOR i IN 0 TO 3 LOOP
                  stanje_var(i*30+4):= stanje_var(i*30+4) XOR
stanje_var(i*30+0);
                  stanje_var(i*30+15):= stanje_var(i*30+15) XOR
stanje_var(i*30+0);
            END LOOP;
            rezultat(stanje_var,izlaz);
            preuzmi<='1';
            cekaj<='0';
            stanje_automata<= idle;
      END IF;
END IF;
```

Ako je ispunjen uslov da je izvršena G_1 faza obrade (uslov u prvoj IF naredbi), izvršava se G_2 faza G runde (13 iteracija se kontroliše uslovom u drugoj IF naredbi i inkrementiranjem promenljive *brojac_druga_petlja*). Nakon 13 iteracija izvršava se preostali deo G_2 faze obrade, prikazan jednačinama 3.3.18-19:

$$S_4 += S_0 \qquad (3.3.18\text{-}19)$$
$$S_{15} += S_0$$

Kod kojim se izvršavaju ove operacije je sledeći (izvršava se ukoliko je ispunjen uslov da je 13 iteracija završeno):

```
FOR i IN 0 TO 3 LOOP
      stanje_var(i*30+4):= stanje_var(i*30+4) XOR stanje_var(i*30+0);
      stanje_var(i*30+15):= stanje_var(i*30+15) XOR stanje_var(i*30+0);
END LOOP;
```

Promenljivom *i* kontroliše se broj vrste u petlji, dok je broj kolona unapred određen.

Nakon završenih *xor* operacija, izvršava se sledeći kod (takođe pod uslovom da je 13 iteracija završeno):

```
rezultat(stanje_var,izlaz);
preuzmi<='1';
cekaj<='0';
stanje_automata<= idle;
```

Pozivanjem procedure *rezultat* signalu *izlaz* se dodeljuje heš vrednost. Pošto ona predstavlja krajnji rezultat primene algoritma, aktivira se signal *preuzmi*. Takođe, signal *cekaj* se postavlja na neaktivnu vrednost, a dizajn se vraća u *idle* stanje i time je spreman za prijem naredne poruke.

3.4. Razlike u kodu za ostale dužine heš vrednosti

Kako postoje određene razlike u koracima algoritma između *Fugue* 2.0-224, *Fugue* 2.0-256, *Fugue* 2.0-384 i *Fugue* 2.0-512, postoje i razlike u VHDL kodu.

Kako *Fugue* 2.0-224 i *Fugue* 2.0-256 koriste stanje iste veličine (dimenzije 4×30), u definisanju tipa *stanje_niz* nema promena. *Fugue* 2.0-384 i *Fugue* 2.0-512 koriste stanje dimenzija 4×36, drugačije je definisan tip *stanje_niz*:

```
TYPE stanje_niz IS ARRAY (0 TO 143 ) OF STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Vektori inicijalizacije se razlikuju za svaku verziju algoritma:

Fugue 2.0-224:

```
CONSTANT iv: STD_LOGIC_VECTOR (223 DOWNTO 0):=
X"3a1d28afdb9b0b7566673079ae45c71c7efbd0e1ad70e5be85430488";
```

Fugue 2.0-256:

```
CONSTANT iv: STD_LOGIC_VECTOR (255 DOWNTO 0):=
X"e952bdde6671135fe0d4f668d2b0b594f96c621dfbf929de9149e89934f8c248";
```

Fugue 2.0-384:

```
CONSTANT iv: STD_LOGIC_VECTOR (383 DOWNTO 0):=
X"8e4f1231837e3d2aec427f83925ac74169fadae515398593657d34f8667eef643d06ff8b144012
3a2d5101be9d119f61";
```

Fugue 2.0-512:

```
CONSTANT iv: STD_LOGIC_VECTOR (511 DOWNTO 0):=
X"9010bba7a7a999bce479d955d50e2474c0d1b8c63db445f36b00cb8ab1057fc7a2ef930570c632
f89834386cac3c9940b3c8ba4aecafa8e5ea32fa93530a723b";
```

Prilikom inicijalizovanja stanja, u različitim verzijama algoritma različite kolone uzimaju vrednost vektora IV.

U *Fugue* 2.0-224, prve 23 kolone se postavljaju na nultu vrednost, dok ostale dobijaju vrednost vektora IV:

```
FOR i IN 0 TO 3 LOOP
      FOR j IN 0 TO 22 LOOP
            stanje (i*30+j):= X"00";
      END LOOP;
END LOOP;
FOR j IN 23 TO 29 LOOP
      FOR i IN 0 TO 3 LOOP
            stanje (i*30+j):= iv_var(223 DOWNTO 216);
            iv_var:= SHL(iv_var, "1000");
      END LOOP;
END LOOP;
```

U *Fugue* 2.0-384, prve 24 kolone se postavljaju na nultu vrednost, dok ostale dobijaju vrednost vektora IV:

```
FOR i IN 0 TO 3 LOOP
      FOR j IN 0 TO 23 LOOP
            stanje (i*36+j):= X"00";
      END LOOP;
END LOOP;
FOR j IN 24 TO 35 LOOP
      FOR i IN 0 TO 3 LOOP
            stanje (i*36+j):= iv_var(383 DOWNTO 376);
            iv_var:= SHL(iv_var, "1000");
      END LOOP;
END LOOP;
```

U *Fugue* 2.0-512, prvih 20 kolona se postavljaju na nultu vrednost, dok ostale dobijaju vrednost vektora IV:

```
FOR i IN 0 TO 3 LOOP
      FOR j IN 0 TO 19 LOOP
            stanje (i*36+j):= X"00";
      END LOOP;
END LOOP;
FOR j IN 20 TO 35 LOOP
      FOR i IN 0 TO 3 LOOP
            stanje (i*36+j):= iv_var(511 DOWNTO 504);
            iv_var:= SHL(iv_var, "1000");
      END LOOP;
END LOOP;
```

TIX korak je isti za *Fugue* 2.0-224 i *Fugue* 2.0-256, dok se razlikuje za ostale dužine izlaza. Jednačine 3.4.1-5 i 3.4.6-11 predstavljaju TIX korak za preostale dužine izlaza, a ispod njih sledi kod procedure *tix* za odgovarajuću dužinu izlaza:

Fugue 2.0-384:

$$\begin{aligned} S_7 &+= S_0 \\ S_0 &= \text{blok poruke} \\ S_{10} &+= S_0 \\ S_{14} &+= S_0 \\ S_4 &+= S_1 \end{aligned} \qquad (3.4.1\text{-}5)$$

```
FOR i IN 0 TO 3 LOOP
      stanje(i*36+7):= stanje(i*36+7)XOR stanje(i*36+0);
END LOOP;
FOR i IN 0 TO 3 LOOP
      stanje(i*36+0):=rec_var(31 DOWNTO 24);
      rec_var:= SHL(rec_var,"1000");
END LOOP;
FOR i IN 0 TO 3 LOOP
      stanje(i*36+10):= stanje(i*36+10) XOR stanje(i*36+0);
      stanje(i*36+14):= stanje(i*36+14) XOR stanje(i*36+0);
      stanje(i*36+4):= stanje(i*36+4) XOR stanje(i*36+1);
END LOOP;
```

Fugue 2.0-512:

$$S_{10}+=S_0$$
$$S_0=\text{blok poruke}$$
$$S_7+=S_0 \qquad (3.4.6\text{-}3.4.11)$$
$$S_{11}+=S_0$$
$$S_4+=S_1$$
$$S_{22}+=S_1$$

```
FOR i IN 0 TO 3 LOOP
      stanje(i*36+10):= stanje(i*36+10) XOR stanje(i*36+0);
END LOOP;
FOR i IN 0 TO 3 LOOP
      stanje(i*36+0):=rec_var(31 DOWNTO 24);
      rec_var:= SHL(rec_var,"1000");
END LOOP;
FOR i IN 0 TO 3 LOOP
      stanje(i*36+7):= stanje(i*36+7) XOR stanje(i*36+0);
      stanje(i*36+11):= stanje(i*36+11) XOR stanje(i*36+0);
      stanje(i*36+4):= stanje(i*36+4) XOR stanje(i*36+1);
      stanje(i*36+22):= stanje(i*36+22) XOR stanje(i*36+1);
END LOOP;
```

CMIX korak je isti za *Fugue* 2.0-224 i *Fugue* 2.0-256, dok se za *Fugue* 2.0-384 i *Fugue* 2.0-512 koristi drugačiji kod. Jednačine 3.4.12-17 predstavljaju CMIX korak za *Fugue* 2.0-384 i *Fugue* 2.0-512:

$$S_0+=S_4$$
$$S_1+=S_5$$
$$S_2+=S_6 \qquad (3.4.12\text{-}3.4.17)$$
$$S_{18}+=S_4$$
$$S_{19}+=S_5$$
$$S_{20}+=S_6$$

Kod kojim se navedene operacije izvršavaju:

```
FOR i IN 0 TO 3 LOOP
      stanje(i*36+0):= stanje(i*36+0) XOR stanje(i*36+4);
      stanje(i*36+1):= stanje(i*36+1) XOR stanje(i*36+5);
      stanje(i*36+2):= stanje(i*36+2) XOR stanje(i*36+6);
      stanje(i*36+18):= stanje(i*36+18) XOR stanje(i*36+4);
      stanje(i*36+19):= stanje(i*36+19) XOR stanje(i*36+5);
      stanje(i*36+20):= stanje(i*36+20) XOR stanje(i*36+6);
END LOOP;
```

R transformacija ostaje ista u *Fugue* 2.0-224, dok se u *Fugue* 2.0-384 izvršavaju dve podrunde, a u *Fugue* 2.0-512 tri podrunde. Kod procedure *R_transformacija* je sledeći:

Fugue 2.0-384:

```
tix(rec, stanje);
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
```

```
super_mix(stanje);
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Fugue 2.0-512:

```
tix(rec, stanje);
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Broj podrundi koje se obavljaju u svakoj iteraciji G_1 faze se razlikuje u svakoj od verzija, kao i broj iteracija. *Fugue* 2.0-224 ima jednu podrundu i 15 iteracija. *Fugue* 2.0-384 ima dve podrunde i 14 iteracija, dok *Fugue* 2.0-512 ima tri podrunde i 14 iteracija. Sledi kod procedure *G_runda_prva_petlja* za ostale dužine izlaza:

Fugue 2.0-224:

```
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Fugue 2.0-384:

```
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Fugue 2.0-512:

```
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
ror3(stanje);
cmix(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

U kodu unutar arhitekture top-level entiteta, razlikuju se uslovi u IF naredbama koji se odnose na promenljivu *brojac_prva_petlja*. Brojač ne sme preći maksimalnu vrednost koja iznosi 15, 14 i 14 za *Fugue* 2.0-224, *Fugue* 2.0-384 i *Fugue* 2.0-512, respektivno.

G_2 faza obrade finalne G runde različita je za sve dužine izlaza. Skup koraka i operacija koje je potrebno izvršiti za svaku dužinu heš izlaza označen je brojevima 3.4.18, 3.4.19 i 3.4.20:

Fugue 2.0-224:

$$\begin{aligned} &13 \times \{\ S_4{+}{=}S_0;\ \ S_{15}{+}{=}S_0;\ \mathrm{ROR15};\ \mathrm{SMIX};\ S_4{+}{=}S_0;\ \ S_{16}{+}{=}S_0;\ \mathrm{ROR14};\ \mathrm{SMIX};\ \} \\ &S_4{+}{=}S_0 \\ &S_{15}{+}{=}S_0 \end{aligned} \tag{3.4.18}$$

Fugue 2.0-384:

$$\begin{aligned} &13 \times \{\ S_4{+}{=}S_0;\ \ S_{12}{+}{=}S_0;\ S_{24}{+}{=}S_0;\ \mathrm{ROR12};\ \mathrm{SMIX}; \\ &\quad S_4{+}{=}S_0;\ \ S_{13}{+}{=}S_0;\ S_{24}{+}{=}S_0;\ \mathrm{ROR12};\ \mathrm{SMIX}; \\ &\quad S_4{+}{=}S_0;\ \ S_{13}{+}{=}S_0;\ S_{25}{+}{=}S_0;\ \mathrm{ROR11};\ \mathrm{SMIX};\ \} \\ &S_4{+}{=}S_0 \\ &S_{12}{+}{=}S_0 \\ &S_{24}{+}{=}S_0 \end{aligned} \tag{3.4.19}$$

Fugue 2.0-512:

$$\begin{aligned} &13 \times \{\ S_4{+}{=}S_0;\ \ S_9{+}{=}S_0;\ S_{18}{+}{=}S_0;\ S_{27}{+}{=}S_0;\ \mathrm{ROR9};\ \mathrm{SMIX}; \\ &\quad S_4{+}{=}S_0;\ \ S_{10}{+}{=}S_0;\ S_{24}{+}{=}S_0;\ S_{27}{+}{=}S_0;\ \mathrm{ROR9};\ \mathrm{SMIX}; \\ &\quad S_4{+}{=}S_0;\ \ S_{10}{+}{=}S_0;\ S_{19}{+}{=}S_0;\ S_{27}{+}{=}S_0;\ \mathrm{ROR9};\ \mathrm{SMIX}; \\ &\quad S_4{+}{=}S_0;\ \ S_{10}{+}{=}S_0;\ S_{19}{+}{=}S_0;\ S_{28}{+}{=}S_0;\ \mathrm{ROR8};\ \mathrm{SMIX};\ \} \\ &S_4{+}{=}S_0 \\ &S_9{+}{=}S_0 \\ &S_{18}{+}{=}S_0 \\ &S_{27}{+}{=}S_0 \end{aligned} \tag{3.4.20}$$

Korake ROR8, ROR9, ROR11, ROR12, ROR14, ROR15 izvršavaju istoimene procedure, koje se od procedure *ror3* razlikuju samo po brojevima u petlji i indeksima.

Kod koji se izvršava u svakoj od 13 iteracija je deo procedure *G_runda_druga_petlja*, koja za različite dužine izlaza izgleda drugačije:

Fugue 2.0-224:

```
FOR i IN 0 TO 3 LOOP
      stanje(i*30+4):= stanje(i*30+4) XOR stanje(i*30+0);
      stanje(i*30+15):= stanje(i*30+15) XOR stanje(i*30+0);
END LOOP;
ror15(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

```
FOR i IN 0 TO 3 LOOP
      stanje(i*30+4):= stanje(i*30+4) XOR stanje(i*30+0);
      stanje(i*30+16):= stanje(i*30+16) XOR stanje(i*30+0);
END LOOP;
ror14(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Fugue 2.0-384:

```
FOR i IN 0 TO 3 LOOP
      stanje(i*36+4):= stanje(i*36+4) XOR stanje(i*36+0);
      stanje(i*36+12):= stanje(i*36+12) XOR stanje(i*36+0);
      stanje(i*36+24):= stanje(i*36+24) XOR stanje(i*36+0);
END LOOP;
ror12(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
FOR i IN 0 TO 3 LOOP
      stanje(i*36+4):= stanje(i*36+4) XOR stanje(i*36+0);
      stanje(i*36+13):= stanje(i*36+13) XOR stanje(i*36+0);
      stanje(i*36+24):= stanje(i*36+24) XOR stanje(i*36+0);
END LOOP;
ror12(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
FOR i IN 0 TO 3 LOOP
      stanje(i*36+4):= stanje(i*36+4)XOR stanje(i*36+0);
      stanje(i*36+13):= stanje(i*36+13)XOR stanje(i*36+0);
      stanje(i*36+25):= stanje(i*36+25)XOR stanje(i*36+0);
END LOOP;
ror11(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Fugue 2.0-512:

```
FOR i IN 0 TO 3 LOOP
      stanje(i*36+4):= stanje(i*36+4) XOR stanje(i*36+0);
      stanje(i*36+9):= stanje(i*36+9) XOR stanje(i*36+0);
      stanje(i*36+18):= stanje(i*36+18) XOR stanje(i*36+0);
      stanje(i*36+27):= stanje(i*36+27) XOR stanje(i*36+0);
END LOOP;
ror9(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
FOR i IN 0 TO 3 LOOP
      stanje(i*36+4):= stanje(i*36+4) XOR stanje(i*36+0);
      stanje(i*36+10):= stanje(i*36+10) XOR stanje(i*36+0);
      stanje(i*36+18):= stanje(i*36+18) XOR stanje(i*36+0);
      stanje(i*36+27):= stanje(i*36+27) XOR stanje(i*36+0);
END LOOP;
ror9(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
FOR i IN 0 TO 3 LOOP
      stanje(i*36+4):= stanje(i*36+4) XOR stanje(i*36+0);
      stanje(i*36+10):= stanje(i*36+10) XOR stanje(i*36+0);
      stanje(i*36+19):= stanje(i*36+19) XOR stanje(i*36+0);
```

```
      stanje(i*36+27):= stanje(i*36+27) XOR stanje(i*36+0);
END LOOP;
ror9(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
FOR i IN 0 TO 3 LOOP
      stanje(i*36+4):= stanje(i*36+4) XOR stanje(i*36+0);
      stanje(i*36+10):= stanje(i*36+10) XOR stanje(i*36+0);
      stanje(i*36+19):= stanje(i*36+19) XOR stanje(i*36+0);
      stanje(i*36+28):= stanje(i*36+28) XOR stanje(i*36+0);
END LOOP;
ror8(stanje);
s_box_zamena_4x4(stanje);
super_mix(stanje);
```

Nakon 13 iteracija, preostali deo G_2 faze izvršava se sledećim kodom:

Fugue 2.0-224:

```
FOR i IN 0 TO 3 LOOP
      stanje_var(i*30+4):= stanje_var(i*30+4) XOR stanje_var(i*30+0);
      stanje_var(i*30+15):= stanje_var(i*30+15) XOR stanje_var(i*30+0);
END LOOP;
```

Fugue 2.0-384:

```
FOR i IN 0 TO 3 LOOP
      stanje_var(i*36+4):= stanje_var(i*36+4) XOR stanje_var(i*36+0);
      stanje_var(i*36+12):= stanje_var(i*36+12) XOR stanje_var(i*36+0);
      stanje_var(i*36+24):= stanje_var(i*36+24) XOR stanje_var(i*36+0);
END LOOP;
```

Fugue 2.0-512:

```
FOR i IN 0 TO 3 LOOP
      stanje_var(i*36+4):= stanje_var(i*36+4) XOR stanje_var(i*36+0);
      stanje_var(i*36+9):= stanje_var(i*36+9) XOR stanje_var(i*36+0);
      stanje_var(i*36+18):= stanje_var(i*36+18) XOR stanje_var(i*36+0);
      stanje_var(i*36+27):= stanje_var(i*36+27) XOR stanje_var(i*36+0);
END LOOP;
```

Za svaku dužinu izlaza, kao heš vrednost izdvajaju se različite kolone iz stanja:

Fugue 2.0-224:

$S_1\ S_2\ S_3\ S_4\ S_{15}\ S_{16}\ S_{17}$

Fugue 2.0-384:

$S_1\ S_2\ S_3\ S_4\ S_{12}\ S_{13}\ S_{14}\ S_{15}\ S_{24}\ S_{25}\ S_{26}\ S_{27}$

Fugue 2.0-512:

$S_1\ S_2\ S_3\ S_4\ S_9\ S_{10}\ S_{11}\ S_{12}\ S_{18}\ S_{19}\ S_{20}\ S_{21}\ S_{27}\ S_{28}\ S_{29}\ S_{30}$

Razlike u kodu procedure *rezultat* jesu u opsegu vrednosti promenljivih :

Fugue 2.0-224:

Promenljiva *k* ima opseg 15÷17, dok promenljiva *j* nije promenjena. Promenljiva *brojac* ima opseg 0÷28.

Fugue 2.0-384:

Promenljiva *k* ima opseg 12÷15, dok promenljiva *j* nije promenjena. Definisana je još jedna petlja, koja po istom principu kao i prethodne izdvaja odgovarajuće kolone iz stanja. Petlju kontroliše promenljiva *l*, koja ima opseg 24÷27. Promenljiva *brojac* ima opseg 0÷48.

Fugue 2.0-512:

Promenljiva *k* ima opseg 9÷12, dok promenljiva *j* nije promenjena. Definisane su još dve petlje, koje po istom principu kao i prethodne izdvajaju odgovarajuće kolone iz stanja. Prvu petlju kontroliše promenljiva *l*, koja ima opseg 18÷21. Drugu petlju kontroliše promenljiva *m*, koja ima opseg 27÷30. Promenljiva *brojac* ima opseg 0÷64.

Signal *izlaz* je definisan kao niz od 224, 384 i 512 bita za *Fugue* 2.0-224, *Fugue* 2.0-384 i *Fugue* 2.0-512, respektivno.

4. OPIS PERFORMANSI I VERIFIKACIJA DIZAJNA

4.1. Opis performansi

Izvršavanjem procesa analize i sinteze u slučaju sve četiri dužine izlaza, 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 4.1.1. Za sve četiri dužine izlaza izabran je uređaj *XC5VFX70T* familije *Virtex 5*.

	Vrednost			
Naziv	*Fugue* 2.0-224	*Fugue* 2.0-256	*Fugue* 2.0-384	*Fugue* 2.0-512
Broj slajs registara	1252/44800 (2%)	1284/44800 (2%)	1608/44800 (3%)	1748/44800 (3%)
Broj slajs LUT-ova	5468/44800 (12%)	5409(44800) (12%)	9499/44800 (21%)	13462/44800 (30%)
Broj potpuno iskorišćenih parova LUT-FF	1209/5511 (21%)	1219/5474 (22%)	1465/9642 (15%)	1505/13705 (10%)
Broj RAM/FIFO memorijskih blokova	5/148 (3%)	5/148 (3%)	12/148 (8%)	15/148 (10%)
Broj globalnih taktova (BUFG/BUFGCTRL)	1/32 (3%)	1/32 (3%)	1/32 (3%)	1/32 (3%)
Broj pinova	262/640 (40%)	294/640 (45%)	422/640 (65%)	550/640 (85%)
Maksimalna frekvencija	89.478 MHz	89.478 MHz	67.125 MHz	48.592 MHz

Tabela 4.1.1. Procenjene performanse za različite dužine izlaza

Može se primetiti da povećanjem dužine heš izlaza dizajn troši više resursa, pre svega kombinacione logike tj. LUT elemente. Registri se prvenstveno troše na čuvanje stanja, pa je otuda nešto veća potrošnja registarskih bita u slučajevima većih izlaza jer oni imaju veći broj bita u stanju.

Za svaku od četiri implementacije korišćen je jedan globalni takt, *clk*. Maksimalna frekvencija je niža za veće dužine heš izlaza usled kompleksnije kombinacione logike.

4.2. Verifikacija dizajna

Kako do trenutka pisanja ovog rada autor nije na Internetu pronašao dostupne test vektore potrebne za verifikaciju dizajna, verifikacija će biti obavljena za prvu verziju algoritma, *Fugue*-256 [5]. Smatraće se da je na ovaj način verifikovan i kod napisan za *Fugue* 2.0-256, jer razlike postoje samo u TIX koraku, vektoru inicijalizacije i broju ponavljanja pojedinih koraka.

TIX korak *Fugue*-256 algoritma je prikazan jednačinama 4.2.1-5:

$$\begin{aligned} S_{10} &+= S_0 \\ S_0 &+= \text{blok poruke} \\ S_8 &+= S_0 \\ S_{18} &+= S_4 \\ S_1 &+= S_{24} \end{aligned} \qquad (4.2.1\text{-}5)$$

Kod unutar procedure *tix* je sledeći:

```
FOR i IN 0 TO 3 LOOP
      stanje(i*30+10):= stanje(i*30+10) XOR stanje(i*30+0);
END LOOP;
FOR i IN 0 TO 3 LOOP
      stanje(i*30+0):=rec_var(31 DOWNTO 24);
      rec_var:= SHL(rec_var,"1000");
END LOOP;
FOR i IN 0 TO 3 LOOP
      stanje(i*30+8):= stanje(i*30+8) XOR stanje(i*30+0);
      stanje(i*30+1):= stanje(i*30+1) XOR stanje(i*30+24);
END LOOP;
```

Vektor inicijalizacije koji se koristi u *Fugue*-256 algoritmu je sledeći:

```
CONSTANT iv: STD_LOGIC_VECTOR (255 DOWNTO 0):=
X"e952bdde6671135fe0d4f668d2b0b594f96c621dfbf929de9149e89934f8c248";
```

R transformaciju u ovom algoritmu čini TIX korak i dve podrunde, tako da je i procedura *R_transformacija* promenjena u skladu sa tim.

G_1 faza G runde se sastoji od 5 iteracija, pri čemu se u svakoj iteraciji obavljaju dve podrunde. Iz tog razloga, promenjena je procedura *G_runda_prva_petlja*, kao i brojač u petlji koji kontroliše broj pozivanja te procedure u arhitekturi top-level entiteta.

Verifikacija će se obaviti korišćenjem fajla sa test vektorima sa sajta instituta NIST [8]. Ime fajla je *test_debug.txt* i deo fajla je korišćen za verifikaciju. Celokupan fajl će biti priložen putem CD-a. U pomenutom fajlu prikazane su vrednosti stanja nakon svakog izvršenog koraka. Biće obavljena verifikacija funkcije za inicijalizaciju stanja, svaki od koraka (TIX, ROR3, CMIX, SMIX), a potom i procedure koje vrše R transformaciju, G_1 i G_2 fazu G runde. Finalno, obaviće se verifikacija celokupnog dizajna.

Test vektor koji se koristi kao poruka je niz brojeva od 1 do 64, predstavljenih u obliku bajtova u heksadecimalnom obliku: 01 02 03 04...40. Na kraju se dodaje i originalna dužina poruke predstavljena osmobajtnim brojem tipa *integer*: 00 00 00 00 00 00 02 00.

4.2.1. Verifikacija funkcija i procedura

i) Funkcija inicijalizacije

U fajlu *test_debug.txt*, stanje nakon inicijalizovanja je sledeće:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 e9 66 e0 d2 f9 fb 91 34

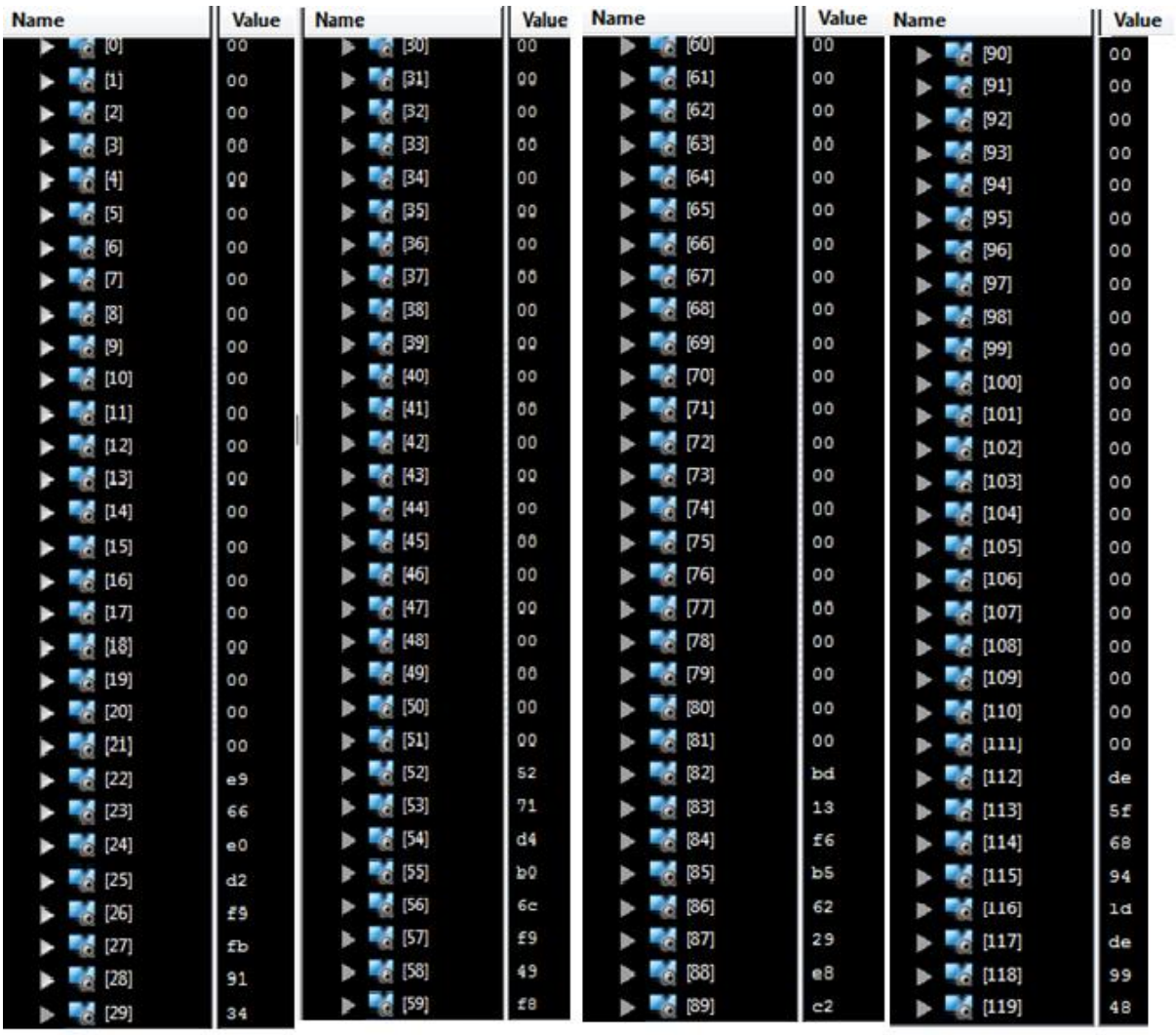
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 52 71 d4 b0 6c f9 49 f8

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 bd 13 f6 b5 62 29 e8 c2

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 de 5f 68 94 1d de 99 48

Napomena: Prvi red predstavlja nultu vrstu matrice stanja, drugi red prvu vrstu, itd... Kolone su odvojene prazninom.

Pozivanjem funkcije *f_inicijalizacija* i pokretanjem ISim simulatora u okviru ISE razvojnog okruženja, promenljiva koja predstavlja stanje prikazana je na slici 4.2.1.1.1.



Slika 4.2.1.1.1. Promenljiva koja predstavlja stanje nakon inicijalizovanja stanja

Napomena: Ova slika i sve naredne dobijene su korišćenjem opcije *Print Screen*, nakon pokretanja ISim simulatora. Kako je stanje prikazano kao niz, a ne matrica, elementi sa opsegom indeksa 0÷29 predstavljaju elemente nulte vrste matrice stanja, opsegom indeksa 30÷59 predstavljena je prva vrsta matrice, itd...

Poklapanjem podataka iz fajla *test_debug.txt* i podataka dobijenim u ISim simulatoru verifikovana je funkcija *f_inicijalizacija.*

ii) Procedura tix

Nakon inicijalizacije, prvi blok poruke prolazi kroz TIX korak koji je deo R transformacije, tako da će procedura *tix* biti verifikovana za TIX korak izvršen za primljen prvi blok poruke (01 02 03 04).

U fajlu *test_debug.txt*, stanje nakon izvršenog TIX koraka je sledeće:

01 e0 00 00 00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 e9 66 e0 d2 f9 fb 91 34

02 d4 00 00 00 00 00 00 02 00 00 00 00 00 00 00 00 00 00 00 00 52 71 d4 b0 6c f9 49 f8

03 f6 00 00 00 00 00 00 03 00 00 00 00 00 00 00 00 00 00 00 00 bd 13 f6 b5 62 29 e8 c2

04 68 00 00 00 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 00 de 5f 68 94 1d de 99 48

Pozivanjem procedure *tix* nakon izvršenih prethodnih koraka i pokretanjem simulatora, promenljiva koja predstavlja stanje prikazana je na slici 4.2.1.2.1.

Slika 4.2.1.2.1. Promenljiva koja predstavlja stanje nakon TIX koraka

Kako se podaci iz fajla sa test vektorom i podaci dobijeni pokretanjem ISim simulatora poklapaju, verifikovana je procedura *tix*.

iii) Procedure ror3 i cmix

Nakon koraka TIX, sledeće stanje prikazano u fajlu sa test vektorima je ono nakon izvršenih koraka ROR3 i CMIX. Vrednosti elemenata stanja nakon tih koraka su sledeće:

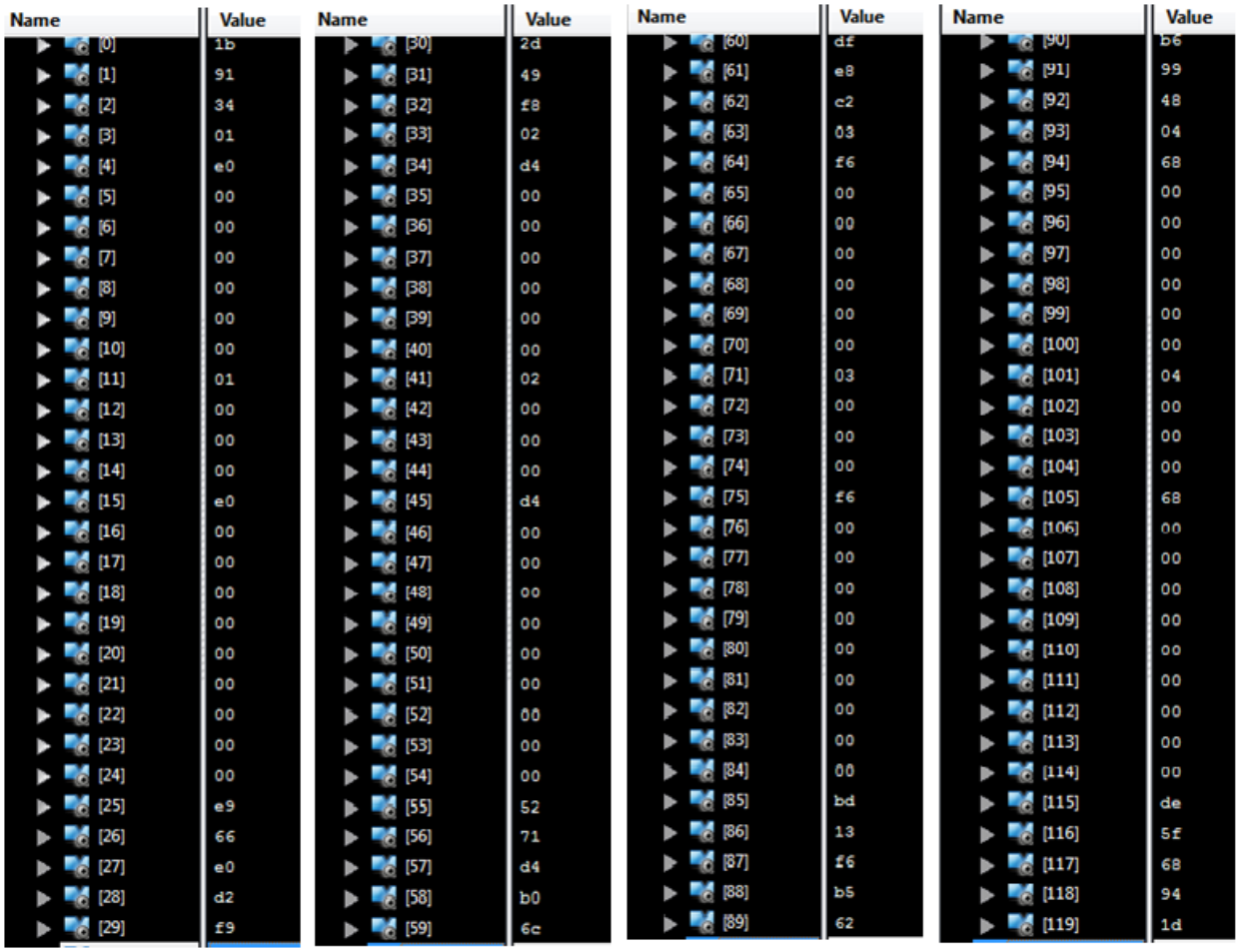
1b 91 34 01 e0 00 00 00 00 00 00 01 00 00 00 e0 00 00 00 00 00 00 00 00 00 e9 66 e0 d2 f9

2d 49 f8 02 d4 00 00 00 00 00 00 02 00 00 00 d4 00 00 00 00 00 00 00 00 00 52 71 d4 b0 6c

df e8 c2 03 f6 00 00 00 00 00 00 03 00 00 00 f6 00 00 00 00 00 00 00 00 00 bd 13 f6 b5 62

b6 99 48 04 68 00 00 00 00 00 00 04 00 00 00 68 00 00 00 00 00 00 00 00 00 de 5f 68 94 1d

Vrši se pozivanje procedura *ror3* i *cmix* nakon izvršenih prethodnih koraka i nakon pokretanja ISim simulacije, stanje je prikazano slikom 4.2.1.3.1.



Slika 4.2.1.3.1. Promenljiva koja predstavlja stanje nakon ROR3 i CMIX koraka

Poklapanjem dobijenih rezultata sa podacima fajla *test_debug.txt*, verifikovane su procedure *ror3* i *cmix.*

iv) Procedure s_box_zamena _4x4 i super_mix

Nakon SMIX koraka, stanje ima sledeću vrednost u *test_debug.txt* fajlu:

be 8a 23 8a e0 00 00 00 00 00 00 01 00 00 00 e0 00 00 00 00 00 00 00 00 00 e9 66 e0 d2 f9

b2 86 a8 5d d4 00 00 00 00 00 00 02 00 00 00 d4 00 00 00 00 00 00 00 00 00 52 71 d4 b0 6c

01 f8 64 7c f6 00 00 00 00 00 00 03 00 00 00 f6 00 00 00 00 00 00 00 00 00 bd 13 f6 b5 62

ce 96 0e 2e 68 00 00 00 00 00 00 04 00 00 00 68 00 00 00 00 00 00 00 00 00 de 5f 68 94 1d

Vrednost stanja dobijena pozivanjem procedura *s_box_zamena _4x4* i *super_mix* nakon izvršenih prethodnih koraka i pokretanjem ISim simulatora prikazane su na slici 4.2.1.4.1.

Slika 4.2.1.4.1. Promenljiva koja predstavlja stanje nakon SMIX koraka

Kako se podaci dobijeni pokretanjem simulacije i oni u fajlu poklapaju, verifikovane su procedure *s_box_zamena_4x4* i *super_mix*.

v) Procedura R_transformacija

U fajlu sa test vektorima, nakon R transformacije (TIX korak, posle koga slede dve podrunde) primenjene nakon prijema prvog bloka poruke, stanje je sledeće:

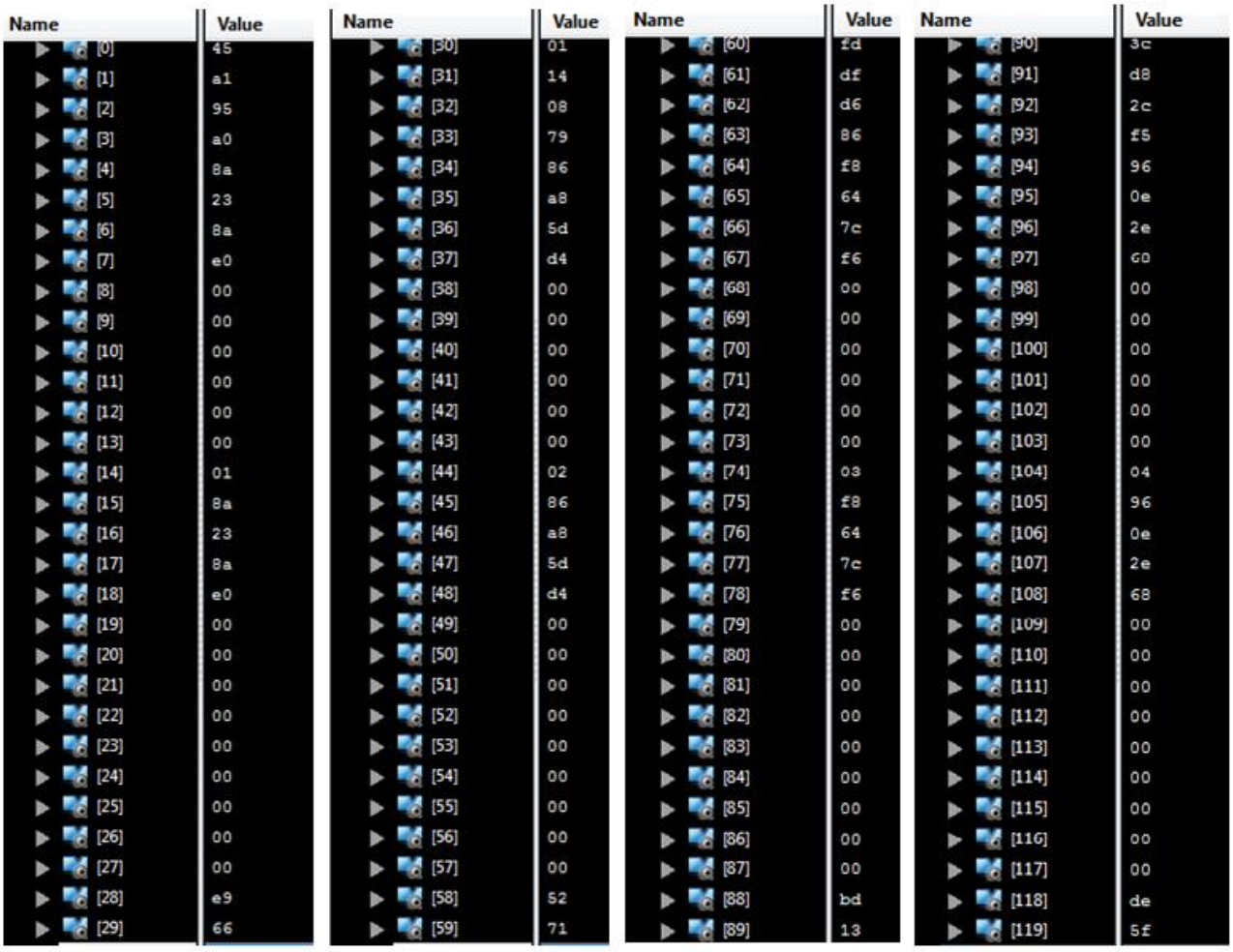
45 a1 95 a0 8a 23 8a e0 00 00 00 00 00 00 01 8a 23 8a e0 00 00 00 00 00 00 00 00 e9 66

01 14 08 79 86 a8 5d d4 00 00 00 00 00 00 02 86 a8 5d d4 00 00 00 00 00 00 00 00 52 71

fd df d6 86 f8 64 7c f6 00 00 00 00 00 00 03 f8 64 7c f6 00 00 00 00 00 00 00 00 bd 13

3c d8 2c f5 96 0e 2e 68 00 00 00 00 00 00 04 96 0e 2e 68 00 00 00 00 00 00 00 00 de 5f

Nakon pozivanja procedure *R_transformacija* nakon inicijalizovanja stanja i postavljanja prvog bloka poruke za ulazni argument, pokretanjem ISim simulatora dobija se stanje prikazano na slici 4.2.1.5.1.



Slika 4.2.1.5.1. Promenljiva koja predstavlja stanje nakon R transformacije

vi) Procedura G_runda_prva_petlja

Stanje nakon izvršenja G_1 faze finalne G runde, prema fajlu *test_debug.txt*, ima sledeću vrednost:

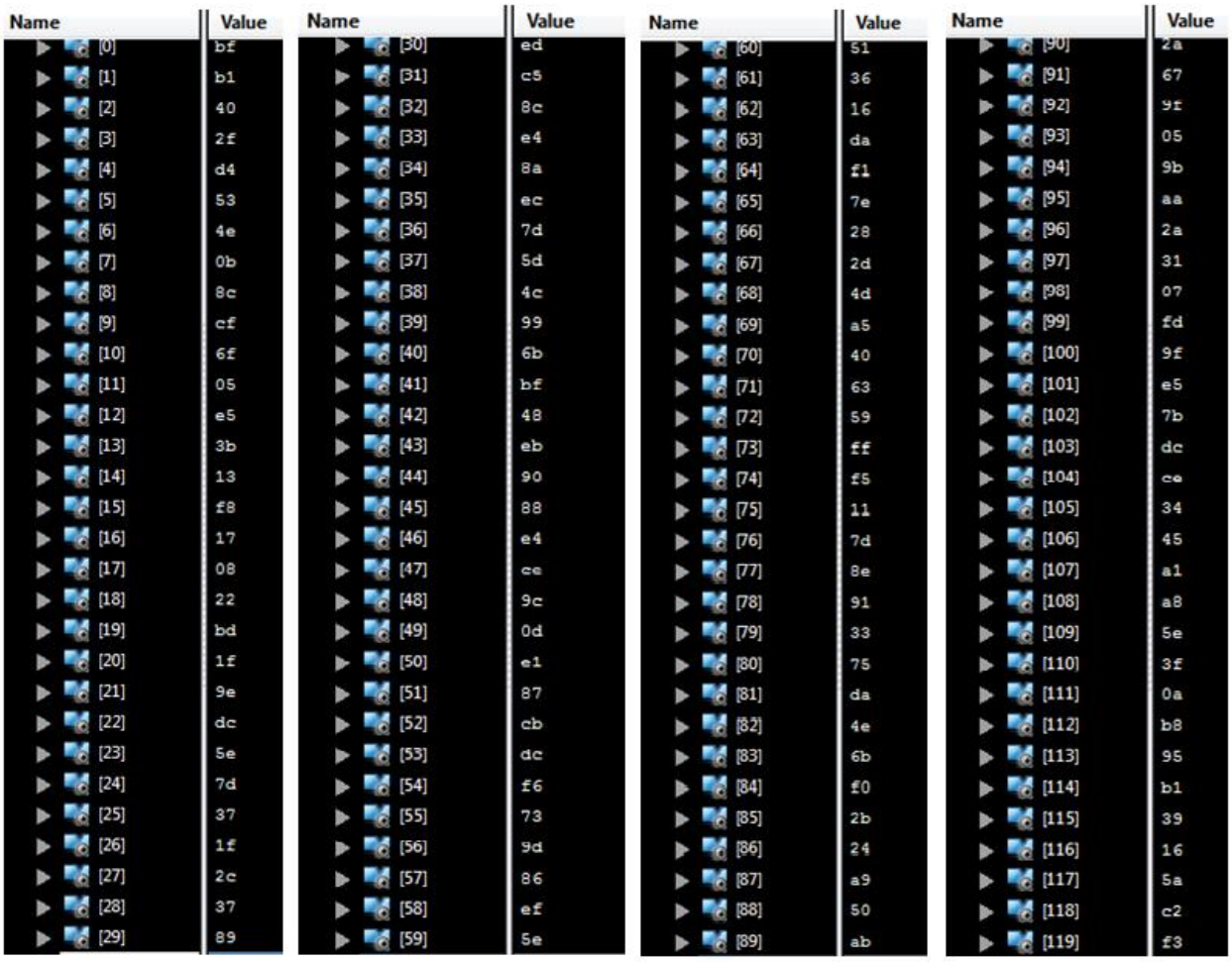
bf b1 40 2f d4 53 4e 0b 8c cf 6f 05 e5 3b 13 f8 17 08 22 bd 1f 9e dc 5e 7d 37 1f 2c 37 89

ed c5 8c e4 8a ec 7d 5d 4c 99 6b bf 48 eb 90 88 e4 ce 9c 0d e1 87 cb dc f6 73 9d 86 ef 5e

51 36 16 da f1 7e 28 2d 4d a5 40 63 59 ff f5 11 7d 8e 91 33 75 da 4e 6b f0 2b 24 a9 50 ab

2a 67 9f 05 9b aa 2a 31 07 fd 9f e5 7b dc ce 34 45 a1 a8 5e 3f 0a b8 95 b1 39 16 5a c2 f3

Nakon izvršenih prethodnih koraka, pozivanjem procedure *G_runda_prva_petlja* odgovarajući broj puta i pokretanjem ISim simulatora dobija se stanje prikazano na slici 4.2.1.6.1.



Slika 4.2.1.6.1. Promenljiva koja predstavlja stanje nakon G_1 faze finalne G runde

Kako se stanje prikazano u fajlu sa test vektorima i stanje dobijeno pokretanjem simulatora poklapaju, verifikovana je procedura *G_runda_prva_petlja* i kod u arhitekturi top-level entiteta koji se koristi za izvršavanje G_1 faze G runde.

vii) Procedura G_runda_druga_petlja

Nakon izvršenja G_2 faze obrade u G rundi, stanje ima sledeću vrednost, prema fajlu *test_debug.txt*:

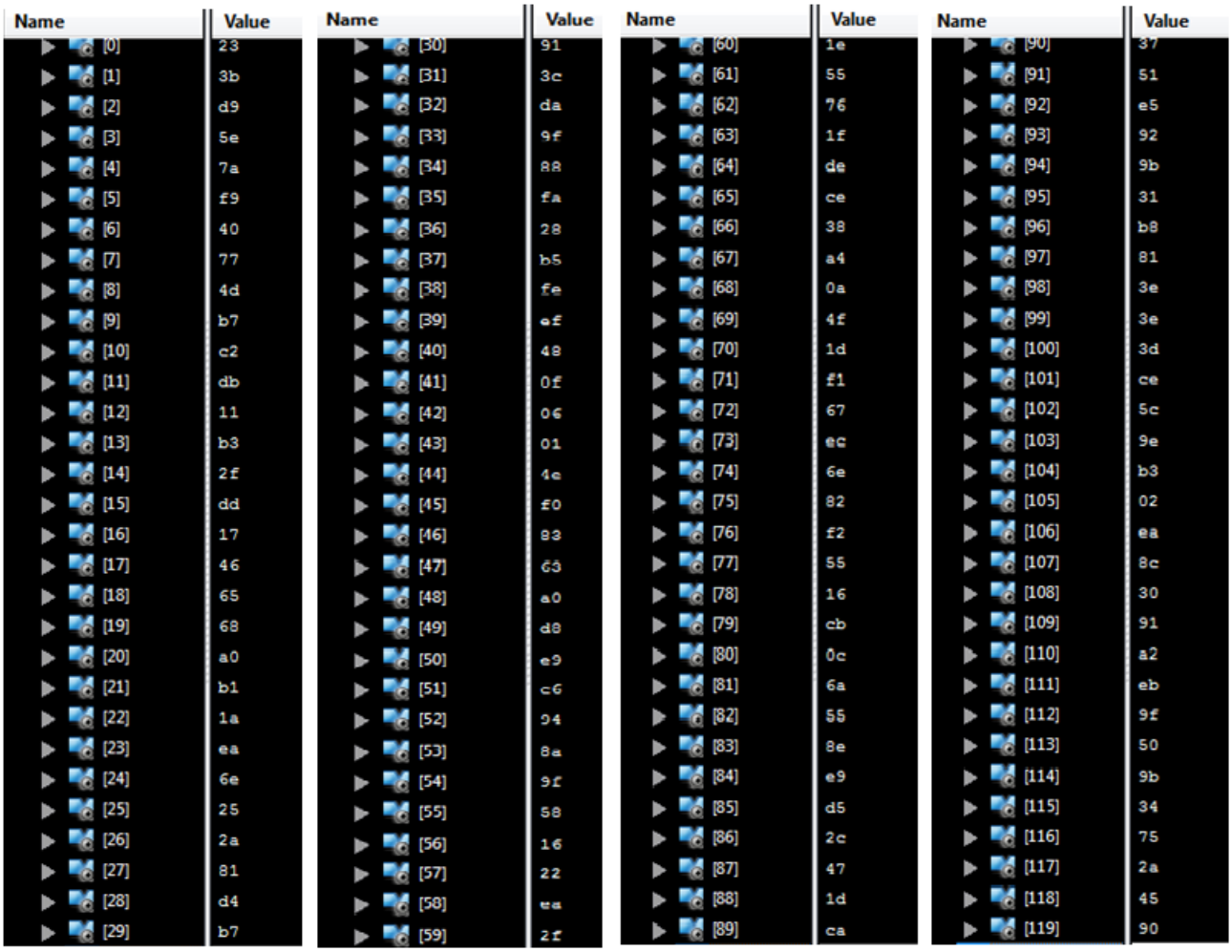
23 3b d9 5e 7a f9 40 77 4d b7 c2 db 11 b3 2f dd 17 46 65 68 a0 b1 1a ea 6e 25 2a 81 d4 b7

91 3c da 9f 88 fa 28 b5 fe ef 48 0f 06 01 4e f0 83 63 a0 d8 e9 c6 94 8a 9f 58 16 22 ea 2f

1e 55 76 1f de ce 38 a4 0a 4f 1d f1 67 ec 6e 82 f2 55 16 cb 0c 6a 55 8e e9 d5 2c 47 1d ca

37 51 e5 92 9b 31 b8 81 3e 3e 3d ce 5c 9e b3 02 ea 8c 30 91 a2 eb 9f 50 9b 34 75 2a 45 90

Nakon izvršenih prethodnih koraka, pozivanjem procedure *G_runda_druga_petlja* odgovarajući broj puta i izvršavanjem preostalih operacija G_2 faze obrade, pokretanjem ISim simulatora dobija se stanje prikazano na slici 4.2.1.7.1.



Slika 4.2.1.7.1. Promenljiva koja predstavlja stanje nakon G_2 faze finalne G runde

Poklapanjem stanja dobijenog simulacijom sa podacima u fajlu, verifikovana je procedura *G_runda_druga_petlja* i kod u arhitekturi top-level entiteta koji se odnosi na izvršavanje G_2 faze G runde.

viii) Ostale procedure

Kako su procedure *ror14* i *ror15* analogne proceduri *ror3*, smatraće se da su njenom verifikacijom verifikovane i pomenute procedure.

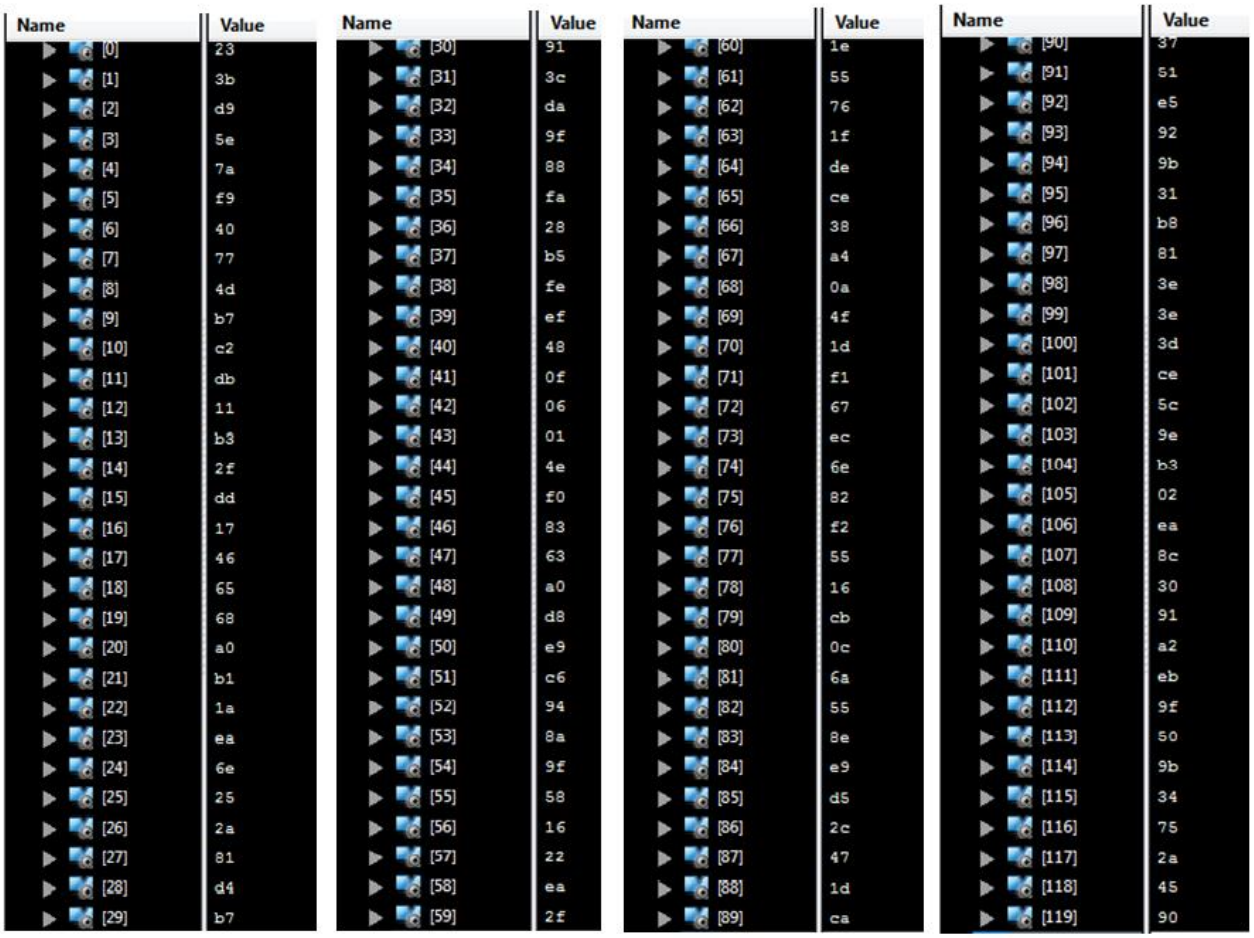
Procedura *rezultat* će biti verifikovana pri verifikaciji celokupnog dizajna, jer će se poklapanjem heš vrednosti pokazati da su iz stanja izdvojene odgovarajuće kolone i prikazane pravilnim redosledom.

4.2.2. Verifikacija celokupnog dizajna

Verifikacija celokupnog dizajna podrazumeva pokretanje ISim simulatora za celokupan kod. Kao što je rečeno, test vektor koji se koristi je niz brojeva od 1 do 64, koji su predstavljeni bajtovima u heksadecimalnom obliku. Kako je određeno algoritmom, poruci se dodaje osmobajtna predstava dužine poruke, u ovom slučaju 00 00 00 00 00 00 02 00.

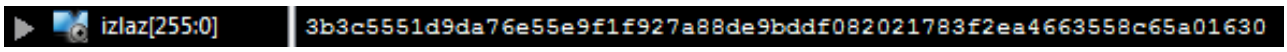
U fajlu sa test vektorima, finalno stanje ima sledeću vrednost:

23 3b d9 5e 7a f9 40 77 4d b7 c2 db 11 b3 2f dd 17 46 65 68 a0 b1 1a ea 6e 25 2a 81 d4 b7
91 3c da 9f 88 fa 28 b5 fe ef 48 0f 06 01 4e f0 83 63 a0 d8 e9 c6 94 8a 9f 58 16 22 ea 2f
1e 55 76 1f de ce 38 a4 0a 4f 1d f1 67 ec 6e 82 f2 55 16 cb 0c 6a 55 8e e9 d5 2c 47 1d ca
37 51 e5 92 9b 31 b8 81 3e 3e 3d ce 5c 9e b3 02 ea 8c 30 91 a2 eb 9f 50 9b 34 75 2a 45 90
Pokretanjem ISim simulatora, dobijeno je stanje prikazano na slici 4.2.2.1.



Slika 4.2.2.1. Promenljiva koja predstavlja stanje nakon završene obrade

Heš vrednost koju bi trebalo dobiti, prema fajlu sa test vektorima, je:
3b3c5551 d9da76e5 5e9f1f92 7a88de9b ddf08202 1783f2ea 4663558c 65a01630
Vrednost promenljive *izlaz* je prikazana slikom 4.2.2.2.

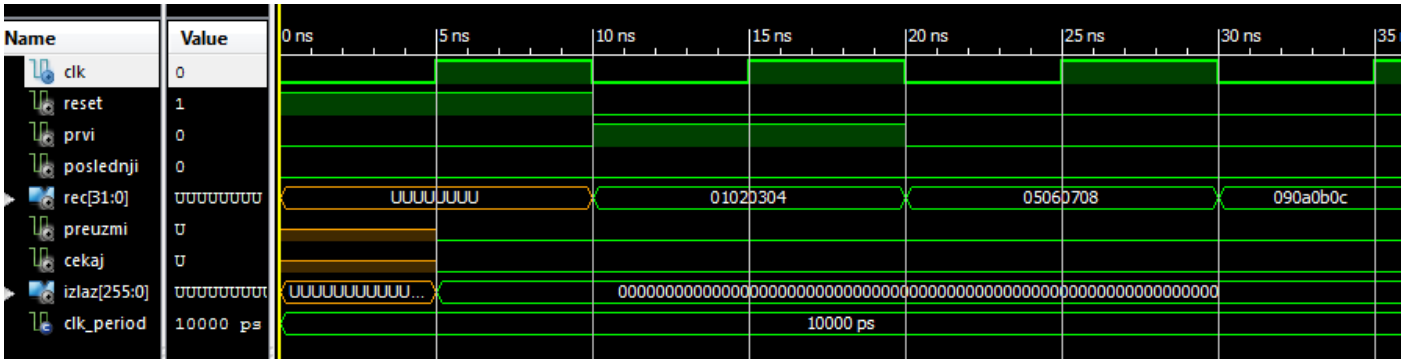


Slika 4.2.2.2. Promenljiva koja predstavlja heš vrednost

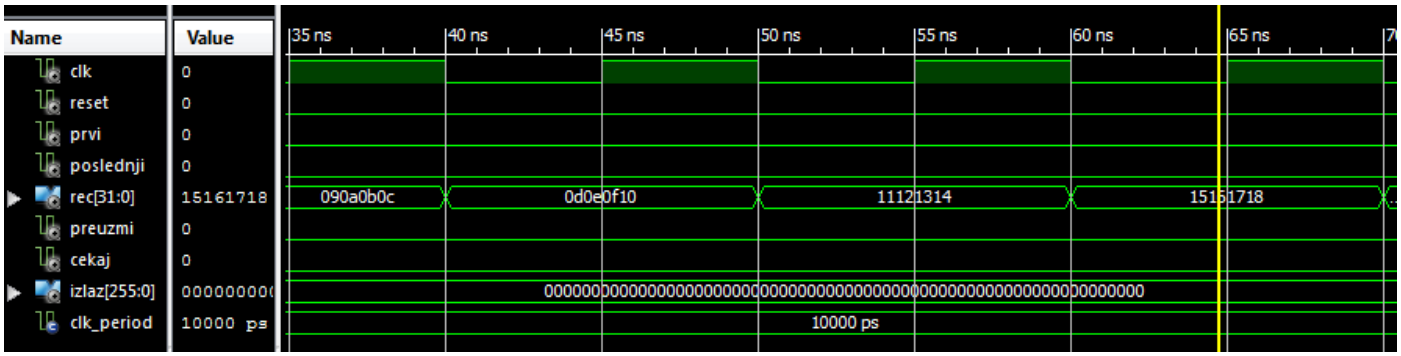
Poklapanjem podataka o stanju i heš vrednosti, verifikovan je celokupan dizajn.

Na narednim slikama (slike 4.2.2.3-14) biće prikazan rezultat simulacije od početka do kraja obrade, gde će se videti i promena vrednosti signala *cekaj* i *preuzmi* u zavisnosti od trenutne faze obrade.

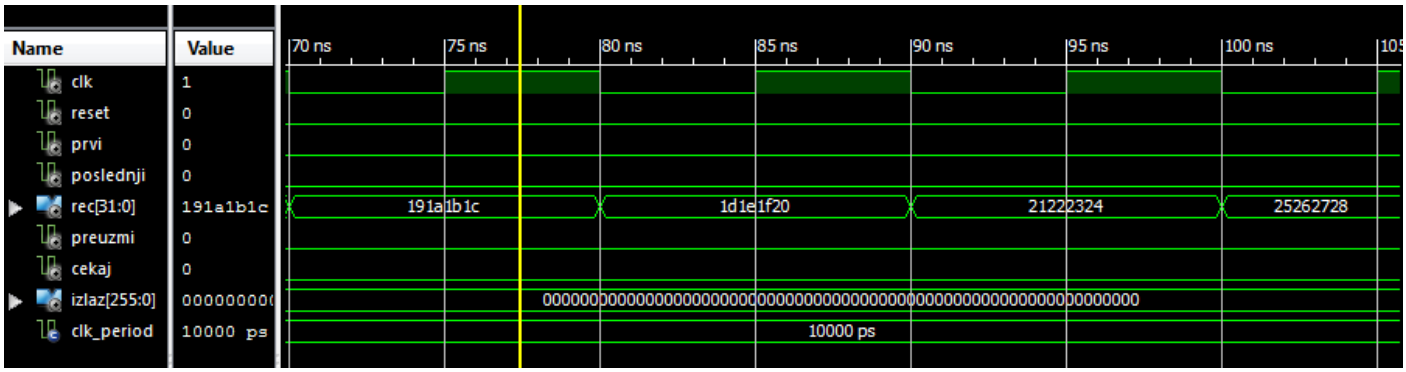
Na slici 4.2.2.3 može se videti da aktiviranjem signala *reset* signali *preuzmi*, *cekaj* i *izlaz* dobijaju inicijalne, nulte vrednosti. Takođe, na ovoj slici je aktiviran signal *prvi* i sve slike do slike 4.2.2.8, zaključno sa njom, predstavljaju stanje obrade *R_runda*.



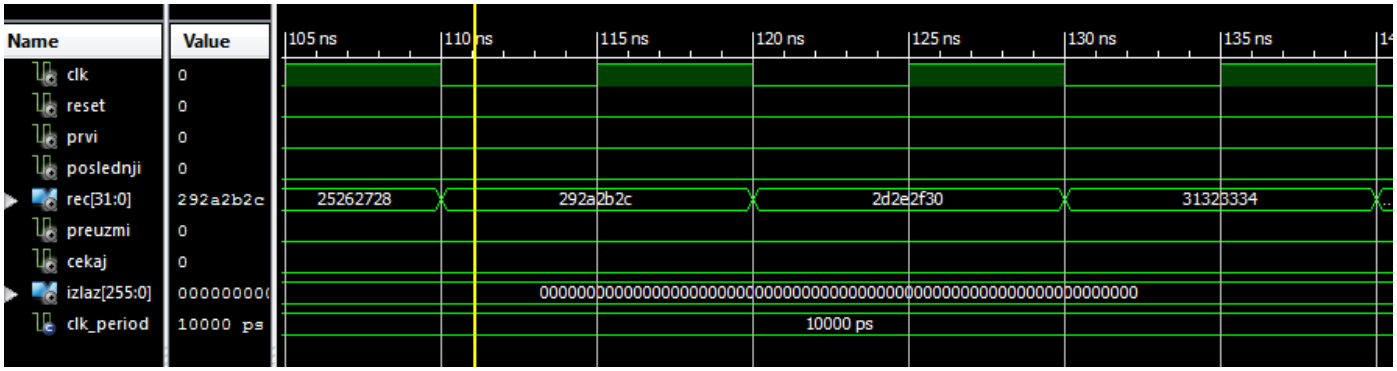
Slika 4.2.2.3. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 1



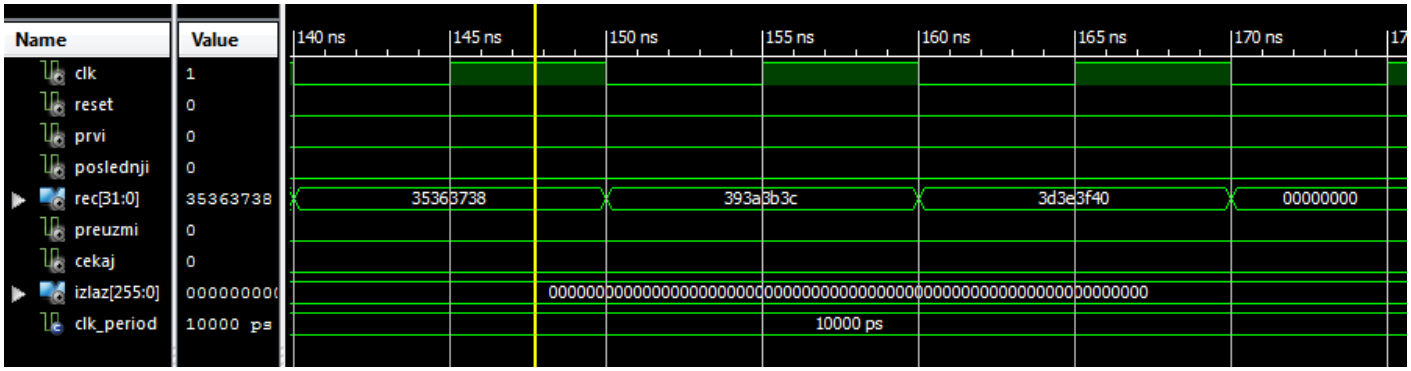
Slika 4.2.2.4. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 2



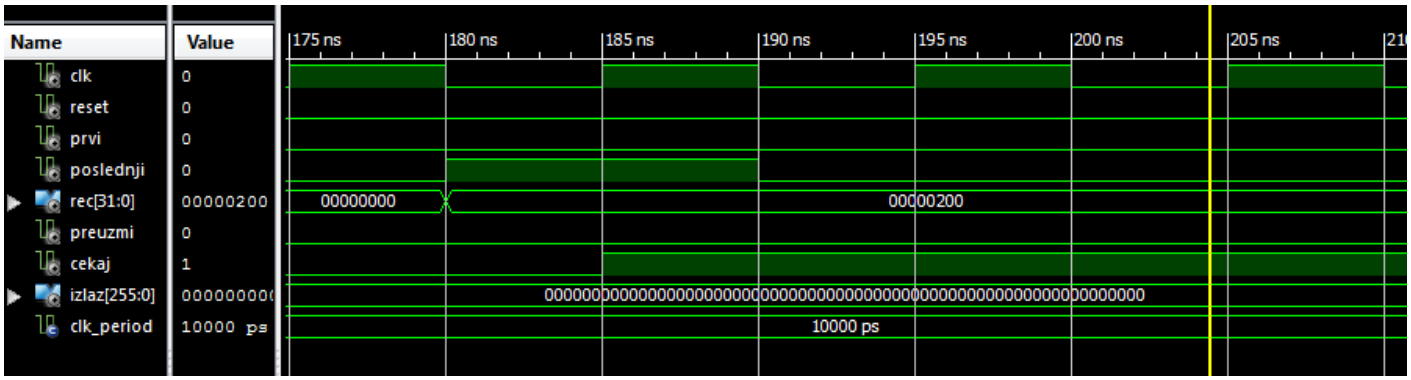
Slika 4.2.2.5. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 3



Slika 4.2.2.6. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 4

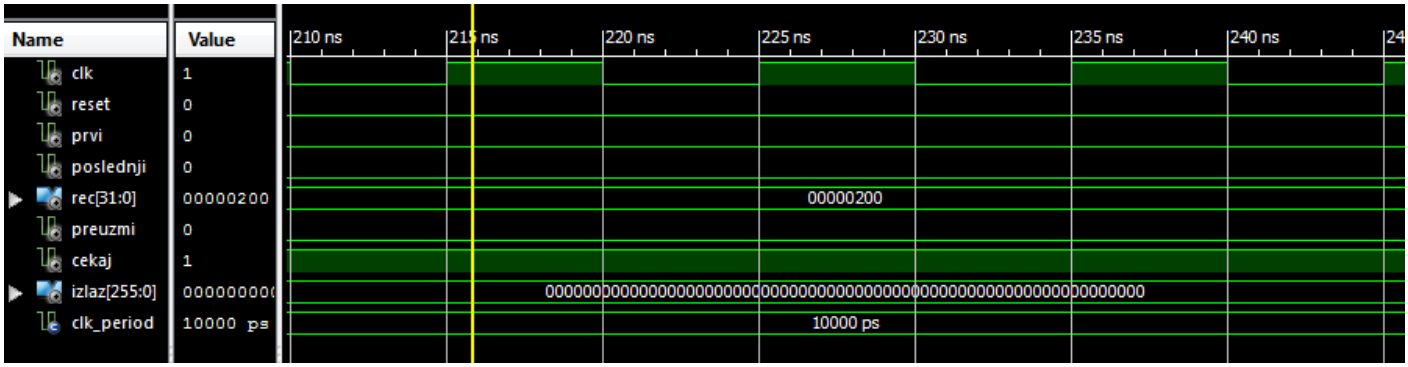


Slika 4.2.2.7. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 5

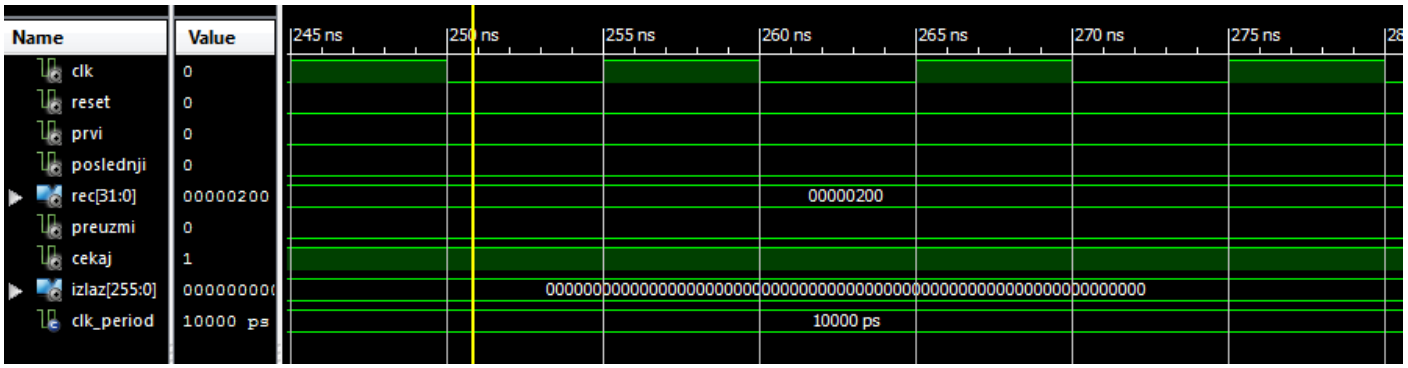


Slika 4.2.2.8. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 6

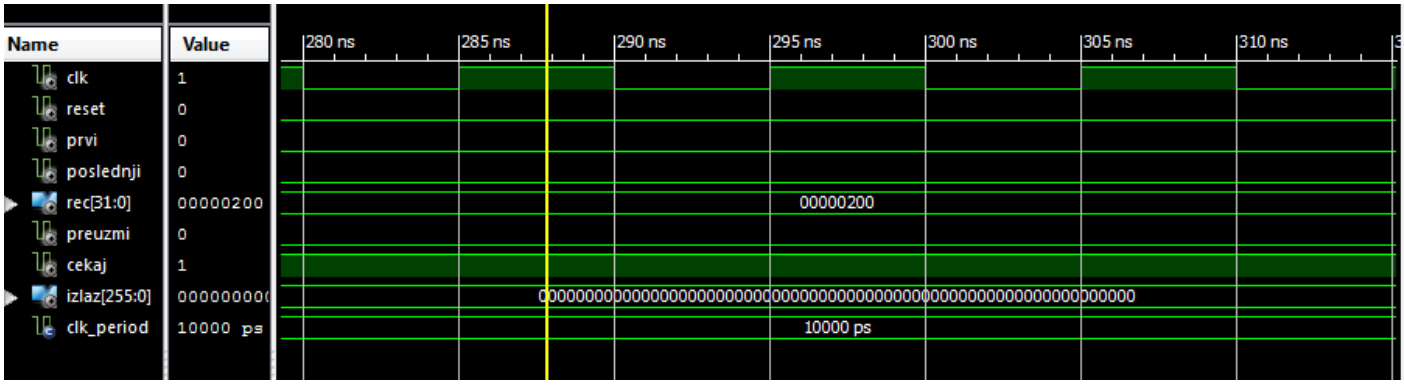
Na slici 4.2.2.8, nakon aktivnog signala *poslednji*, aktivira se signal *cekaj* i dizajn se nalazi u stanju obrade *G_runda*, u kome ostaje sve do trenutka prikazanog na slici 4.2.2.13.



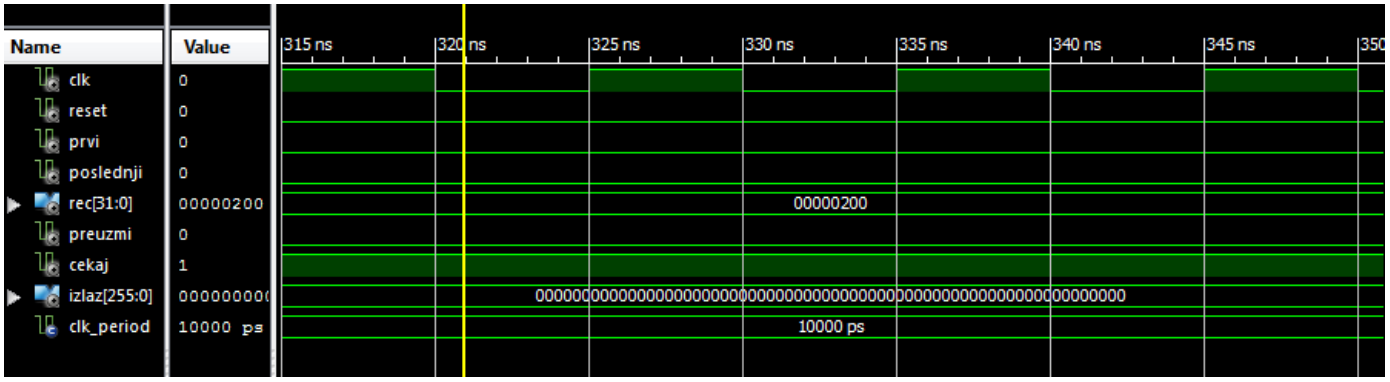
Slika 4.2.2.9. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 7



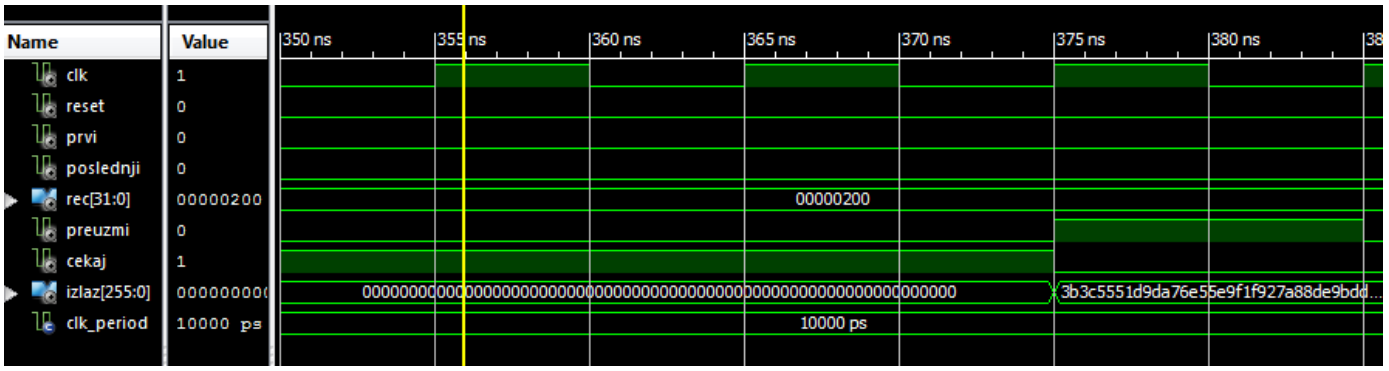
Slika 4.2.2.10. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 8



Slika 4.2.2.11. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 9



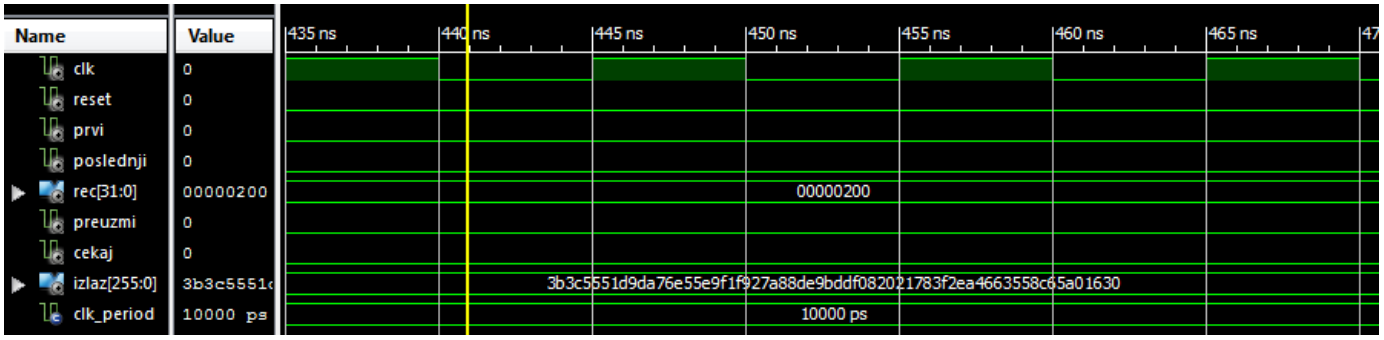
Slika 4.2.2.12. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 10



Slika 4.2.2.13. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 11

Na slici 4.2.2.13 vidi se da se nakon završetka obrade aktivira signal *preuzmi* i da je u tom trenutku dostupna heš vrednost u obliku vrednosti signala *izlaz*. Takođe, signal *cekaj* se postavlja na nultu vrednost.

Već u sledećem taktu (slika 4.2.2.14), signal *preuzmi* postaje neaktivan i dizajn prelazi u stanje *idle* do prijema sledeće poruke.



Slika 4.2.2.14. Izgled prozora ISim simulatora nakon pokretanja simulacije celokupnog dizajna – deo 12

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.

5. ZAKLJUČAK

S obzirom na veliki broj raznovrsnih zlonamernih napada na Internetu na raspolaganju su raznovrsni mehanizmi zaštite. Jedan od njih su i algoritmi za heširanje. Dizajniran je veliki broj različitih algoritama i svima se mogu naći i prednosti i mane. Autor ovog rada može dati subjektivni utisak o prednostima i manama *Fugue* algoritma, kao i o realizovanoj implementaciji:

Svaki od koraka algoritma može se raščlaniti na niz jednostavnih operacija: *xor* operacija, rotiranje sadržaja promenljive, obična dodela vrednosti. One čine algoritam nekomplikovanim za realizaciju.

Realizovana implementacija zahteva dosta vremena za izvršavanje analize i sinteze (45 min za *Fugue* 2.0-512) jer iako se koriste proste operacije kombinaciona logika je suviše velika.

Vreme potrebno za izvršavanje analize i sinteze bi se moglo smanjiti realizovanjem obrade u većem broju taktova (tačnije faze u kojoj se izvršava G runda). Ukoliko bi se faza prijema i obrade blokova poruke (R transformacija) izvršavala u većem broju taktova to bi zahtevalo povećanu upotrebu registara koji bi čuvali blokove poruke dok ne dođe red na njihovu obradu.

Kako se uvek teži što višoj frekvenciji na kojoj dizajn može da radi, u ovoj implementaciji ona bi se mogla povećati upotrebom *pipeline* strukture [9]. Implementacija *pipeline* strukture podrazumeva deljenje složene operacije na više jednostavnijih. Viša maksimalna frekvencija se postiže dodavanjem određenog broja flip-flopova, koji smanjuju kašnjenje kroz elemente kombinacione logike. Cena ove implementacije je kašnjenje između ulaza i izlaza, izraženo u određenom broju taktova. I u ovom slučaju bi bila neophodna povećana upotreba registara koji bi čuvali podatke do odgovarajućeg trenutka za obradu.

LITERATURA

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

[2] Wikipedia, Cryptographic Hash Function [Online]. Preuzeto sa: http://en.wikipedia.org/wiki/Cryptographic_hash_function

[3] Vikipedija, Heš Funkcija [Online]. Preuzeto sa: http://sr.wikipedia.org/sr/He%C5%A1_funkcija

[4] S. Friedl, An Illustrated Guide to Cryptographic Hashes [Online]. Preuzeto sa: http://www.unixwiz.net/techtips/iguide-crypto-hashes.html

[5] S. Halevi, W. E. Hall, C. S. Jutla, The Hash Function "Fugue" [Online]. Preuzeto sa: http://researcher.watson.ibm.com/researcher/files/us-csjutla/fugue_Oct09.pdf

[6] S. Halevi, W. E. Hall, C. S. Jutla, The Hash Function "Fugue 2.0" [Online]. Preuzeto sa: http://researcher.watson.ibm.com/researcher/files/us-csjutla/fugue_2012.pdf

[7] Wikipedia, Finite Field Arithmetic [Online]. Preuzeto sa: http://en.wikipedia.org/wiki/Finite_field_arithmetic

[8] *test_debug.txt* [Online]. Preuzeto sa: http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Fugue_Round2.zip

[9] VHDL coding tips and tricks, What is pipelining? Explanation with a simple example in VHDL. [Online]. Preuzeto sa: http://vhdlguru.blogspot.com/2011/01/what-is-pipelining-explanation-with.html