UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA
KATEDRA ZA ELEKTRONIKU

Projektovanje, integracija i verifikacija IP jezgra za konvoluciju matrica zasnovanog na FFT algoritmu

Završni rad

Osnovne akademske studije

kandidat
Vladimir Vincan, EE5-2015

mentor
dr Vuk Vranjković, docent

Novi Sad, Jul 2020

	UNIVERZITET U NOVOM SADU • **FAKULTET TEHNIČKIH NAUKA** 21000 Novi Sad, Trg Dositeja Obradovića 6	Broj:
	ZADATAK ZA ZAVRŠNI (BACHELOR) RAD	Datum:

(Podatke unosi predmetni nastavnik - mentor)

Vrsta studija:	Osnovne akademske studije
Studijski program:	Energetika, elektronika i telekomunikacije
Rukovodilac studijskog programa:	dr Milan Sečujski, vanredni profesor

Student:	Vladimir Vincan	Broj indeksa:	EE5-2015
Oblast:	Projektovanje složenih digitalnih sistema		
Mentor:	dr Vuk Vranjković, docent		

NA OSNOVU PODNETE PRIJAVE, PRILOŽENE DOKUMENTACIJE I ODREDBI STATUTA FAKULTETA IZDAJE SE ZADATAK ZA ZAVRŠNI (Bachelor) RAD, SA SLEDEĆIM ELEMENTIMA:

- problem – tema rada;
- način rešavanja problema i način praktične provere rezultata rada, ako je takva provera neophodna;
- literatura

NASLOV ZAVRŠNOG (BACHELOR) RADA:

Projektovanje, integracija i verifikacija IP jezgra za konvoluciju matrica zasnovanog na FFT algotirmu

TEKST ZADATKA:

1. Izvršiti projektovanje i analizu na sistemskom nivou hardversko-softverskog kodizajn rešenja konvolucije matrica, zasnovanog na brzim Furijeovim transformacijama.
2. Projektovati IP jezgro za brzu Furijeovu transformaciju matrica.
3. Verifikovati projektovano IP jezgro koristeći UVM metodologiju.
4. Razvoj Linuks drajvera za komunikaciju sa projektovanim IP jezgrom i pisanje korisničke aplikacije.

Rukovodilac studijskog programa:	Mentor rada:

Primerak za: O-studenta; O-studentsku službu fakulteta

UNIVERZITET U NOVOM SADU • **FAKULTET TEHNIČKIH NAUKA**
21000 Novi Sad, Trg Dositeja Obradovića 6

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj, **RBR:**			
Identifikacioni broj, **IBR:**			
Tip dokumentacije, **TD:**		Monografska dokumentacija	
Tip zapisa, **TZ:**		Tekstualni štampani materijal	
Vrsta rada, **VR:**		Završni rad	
Autor, **AU:**		Vladimir Vincan	
Mentor, **MN:**		dr Vuk Vranjković, docent	
Naslov rada, **NR:**		Projektovanje, integracija i verifikacija IP jezgra za konvoluciju matrica zasnovanog na FFT algotirmu	
Jezik publikacije, **JP:**		Srpski	
Jezik izvoda, **JI:**		Srpski	
Zemlja publikovanja, **ZP:**		Srbija	
Uže geografsko područje, **UGP:**		Vojvodina	
Godina, **GO:**		2020	
Izdavač, **IZ:**		Autorski reprint	
Mesto i adresa, **MA:**		21000 Novi Sad, Trg Dositeja Obradovića 6	
Fizički opis rada, **FO:** (poglavlja/strana/citata/tabela/slika/grafika/priloga)		(7/68/16/7/24/0/0)	
Naučna oblast, **NO:**		Elektronika	
Naučna disciplina, **ND:**		Embeded sistemi	
Predmetna odrednica/Ključne reči, **PO:**		FPGA, Hardverski akcelerator, Furijeova transformacija, Konvolucija	
UDK			
Čuva se, **ČU:**		Biblioteka Fakulteta tehničkih nauka 21000 Novi Sad, Trg Dositeja Obradovića 6	
Važna napomena, **VN:**		Nema	
Izvod, **IZ:**		U ovom završnom radu opisano je projektovanje, verifikacija i integracija IP jezgra za konvoluciju matrica zasnovanog na dvodimenzionalnim brzim Furijeovim transformacijama.	
Datum prihvatanja teme, **DP:**			
Datum odbrane, **DO:**		24.7.2020.	
Članovi komisije, **KO:**	Predsednik:	dr Rastislav Struharik, vanredni profesor	
	Član:	dr Staniša Dautović, vanredni profesor	**Potpis mentora**
	Član, mentor	dr Vuk Vranjković, docent	

UNIVERSITY OF NOVI SAD • **FACULTY OF TECHNICAL SCIENCES**
21000 Novi Sad, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO:**			
Identification number, **INO:**			
Document type, **DT:**		Monographic publication	
Type of record, **TR:**		Textual material, printed	
Contents code, **CC:**		Bachelor thesis	
Author, **AU:**		Vladimir Vincan	
Mentor, **MN:**		Vuk Vranjković, PhD	
Title, **TI:**		Design, Integration and Verification of an IP Core for Matrix Convolution based on the FFT Algorithm	
Language of text:, **LT:**		Serbian	
Language of abstract, **LA:**		Serbian	
Country of publication, **CP:**		Serbia	
Locality of publication, **LP:**		Vojvodina	
Publication year, **PY:**		2020	
Publisher, **PB:**		Author's reprint	
Publication place, **PP:**		21000 Novi Sad, Trg Dositeja Obradovića 6	
Physical description, **PD:** (chapters/ pages/ ref. / tables/ pictures/ graphs/ appendixes)		(7/68/16/7/24/0/0)	
Scientific field, **SF:**		Electrical engineering	
Scientific discipline, **SD:**		Embedded systems	
Subject/ Key words, **S/KW:**		FPGA, Hardware accelerator, Fourier transform, Convolution	
UC			
Holding data, **HD:**		Library of the Faculty of Technical Sciences 21000 Novi Sad, Trg Dositeja Obradovića 6	
Note, **N:**		None	
Abstract, **AB:**		This bachelor thesis describes the design, integration and verification of an IP core for matrix convolution based on the two-dimensional fast Fourier transform algorithm.	
Accepted by the Scientific Board on, **ASB:**			
Defended on, **DE:**		24.7.2020.	
Defended board, **DB:**	President:	Rastislav Struharik, PhD, associate professor	
	Member:	Staniša Dautović, PhD, associate professor	**Mentor's signature**
	Member, Mentor	Vuk Vranjković, PhD, assistant professor	

Izjava o akademskoj čestitosti

Student: **Vladimir Vincan**
Broj indeksa: **EE5-2015**
Student **osnovnih akademskih studija**
Autor rada pod nazivom: **Projektovanje, integracija i verifikacija IP jezgra za konvoluciju matrica zasnovanog na FFT algoritmu**

Potpisivanjem izjavljujem:

- da je rad isključivo rezultat mog sopstvenog istraživačkog rada;
- da sam rad i mišljenja drugih autora koje sam koristio u ovom radu naznačio ili citirao i navedeni u spisku literature/referenci koji su sastavni deo ovog rada;
- da sam dobio sve dozvole za korišćenje autorskog dela koji se u potpunosti/celosti unose u predati rad i da sam to jasno naveo;
- da sam svestan da je plagijat korišćenje tuđih radova u bilo kom obliku (kao citata, parafraza, slika, tabela, dijagrama, dizajna, planova, fotografija, filma, muzike, formula, veb sajtova, kompjuterskih programa i sl.) bez navođenja autora ili predstavljanje tuđih autorskih dela kao mojih, kažnjivo po zakonu (Zakon o autorskom i srodnim pravima, Službeni glasnik Republike Srbije, br. 104/2009, 99/2011, 119/2012), kao i drugih zakona i odgovarajućih akata Univerziteta u Novom Sadu;
- da sam svestan da plagijat uključuje i predstavljanje, upotrebu i distribuiranje rada predavača ili drugih studenata kao sopstvenih;
- da sam svestan posledica koje kod dokazanog plagijata mogu prouzrokovati na predati rad i moj status;
- da je elektronska verzija rada identična štampanom primerku i pristajem na njegovo objavljivanje pod uslovima propisanim aktima Univerziteta.

Novi Sad, 24.7.2020. Potpis studenta

_____

Sadržaj

Slike

Listing koda

Listing tabela

Glava 1

Uvod

Operacija konvolucije predstavlja bazičan algoritam u velikom

broju aplikacija, među kojima se u značajnije predstavnike ubrajaju digitalna obrada slike i konvolucione neuronske mreže. Akceleracija date operacije bi značajno ubrzala algoritme zasnovane na konvoluciji, zato što vreme izvršavanja datih algoritama dominantno zavisi od konvolucije.

Direktna implementacija konvolucije između dva diskretna aperiodična signala sa po N elemenata ima složenost $\mathcal{O}(N^2)$. Korišćenjem brze Furijeove transformacije i osobina koje povezuju konvoluciju sa brzom Furijeovom transformacijom, vremenska složenost algoritma se svodi na $\mathcal{O}(N \log N)$ [1]. Slično kao i u prethodnom slučaju, veću vremensku složenost, odnosno manju brzinu rada, ima algoritam direktne implementacije konvolucije nad matricom, u odnosu na algoritam zasnovan na brzim Furijeovim transformacijama [2].

Stoga, dati projekat ima za cilj da hardverski ubrza izvršavanje algoritma konvolucije nad matricom, korišćenjem brzih Furijeovih transformacija.

Rad se sastoji iz sledećih celina:

1. Prvog, uvodnog poglavlja.
2. U drugom poglavlju su objašnjene operacije konvolucije matrica i brze Furijeove transformacije, njihova veza, kao i osobine koje smanjuju složenost implementiranog algoritma.
3. U trećem poglavlju je opisan tok projektovanja dizajna na sistemskom nivou. To podrazumeva vremensku i prostornu analizu performansi sistema, kao i particionisanje na hardverski i softverski deo.
4. U četvrtom poglavlju su diskutovane moguće implementacije i opisan je projketovani hardverski blok.
5. U petom poglavlju je opisan postupak funkcionalne verifikacije projektovanog hardverskog bloka, zasnovanog na UVM metodologiji.
6. U šestom poglavlju su opisani implementirani Linuks drajver za projektovani hardverski blok i korisnička aplikacija.

Glava 2

Konvolucija matrica

2.1 Konvolucija u jednoj dimenziji

Posmatrajmo dva niza, $f[n]$ i $g[m]$, sa brojem elemenata N i M ($N \geq M$) redom. Pomoću njih će biti objašnjena dva osnovna tipa diskretne konvolucije – linearna i kružna.

- **Linearna konvolucija** se zasniva na proširivanju signala f i g sa nulama i sa leve i sa desne strane, do beskonačnosti, i potom izvršavanje sledeće operacije:

$$h_\infty[k] = (f * g)[k] \triangleq \sum_{i=-\infty}^{\infty} f_\infty[i] \cdot g_\infty[k-i] = (g * f)[k] \tag{1}$$

Pri čemu važi:

$$f_\infty[n] = \begin{cases} f[n], 0 \leq n \leq N \\ 0, inače \end{cases} \tag{2}$$

$$g_\infty[m] = \begin{cases} g[n], 0 \leq m \leq M \\ 0, inače \end{cases} \tag{3}$$

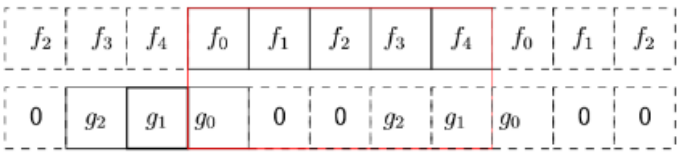
Rezultujući signal h_∞ može imati maksimalno *N+M-1* nenultih vrednosti. U zavisnosti od toga da li konvolucioni proizvod uključuje sve nenulte članove dobijenog niza h_∞, razlikujemo tri tipa konvolucionog množenja:

- **Potpuno množenje** – rezultujući niz h će uključivati sve elemente niza h_∞, koji mogu imati nenultu vrednost. Niz h će posedovati *N+M-1* elemenata.
- **Množenje iste veličine** – rezultujući niz h će uključivati samo članove niza h_∞, kod kojih je centralni element niza g pomnožen sa nekim članom neproširenog niza f. Niz h će posedovati N elemenata.
- **Validno množenje** – rezultujući niz h će uključivati samo članove niza h_∞, kod kojih su svi neproširenи elementi niza g pomnoženi sa neproširenim članovima niza f. Niz h će posedovati *N-M+1* elemenata.

Dalje u radu će se pod linearnim konvolucionim množenjem podrazumevati potpuno množenje, osim ukoliko ne bude drugačije naznačeno, pošto je algoritam za konvoluciju implementiran pomoću potpunog množenja. Matematički, formula za potpuno konvoluciono množenje se definiše na sledeći način:

$$h[k] \triangleq \sum_{i=\max(0,k-M+1)}^{\min(N-1,k)} f[i] \cdot g[k-i], \qquad \forall k \in [0, N+M-2] \tag{4}$$

- **Kružna konvolucija** se zasniva na „obmotavanju“ nizova f i g sa sopstvenim članovima, umesto što se proširuju sa nulom do beskonačnosti. Time se postiže da novodobijeni nizovi f_P i g_P budu periodični sa istim periodom P. Konačni nizovi f i g su prošireni sa nulama tek toliko, da bi se zadovoljio uslov da nizovi budu iste periodičnosti *($P = N_P = M_P \geq N \geq M$)*. Drugim rečima, za bilo koji ceo broj k, važi *f(k) = f(k mod P)* i *g(k) = g(k mod P).* Na slici 1 se nalazi primer za *P=N=5* i *M=3*.



Slika 1 Izgled nizova f i g za P=N=5 i M=3 (preuzeto iz [1])

Matematički, formula za kružno konvoluciono množenje sa modulom P se definiše na sledeći način:

$$h_P[k] = (f *_P g)[k] \triangleq \sum_{i=0}^{P-1} f_P[i] \cdot g_P[k-i] = (g *_P f)[k], \ \forall k \in \mathbb{Z} \tag{5}$$

Pri čemu važi:

$$f_P[i] = \sum_{p=-\infty}^{\infty} f_\infty[i + p \cdot P] = f_\infty[i \bmod P], \ \forall i \in \mathbb{Z} \tag{6}$$

$$g_P[i] = \sum_{p=-\infty}^{\infty} g_\infty[i + p \cdot P] = g_\infty[i \bmod P], \ \forall i \in \mathbb{Z} \tag{7}$$

U slučaju da je moduo kružne konvolucije $P \geq N + M - 1$, jednostavno se može pokazati da linearna i kružna konvolucija predstavljaju identičnu operaciju [1]:

$$(f * g)[k] = (f *_P g)[k], \quad ako\ P \geq N + M - 1 \tag{8}$$

Uloga i značaj pojma kružne konvolucije je u tome što ona predstavlja spregu između diskretne Furijeove transformacije i linearne konvolucije signala.

2.2 Konvolucija u dve dimenzije

Celokupna teorija diskutovana za konvoluciju u jednoj dimenziji može biti primenjena i na konvoluciju u više dimenzija. Ovaj rad će biti ograničen na dve dimenzije, odnosno ograničen na konvolucije nad dvodimenzionalnim nizovima, koji se zovu matrice. Linearna konvolucija nad matricama f i g se definiše na sledeći način:

$$h_\infty[k_1,k_2] \triangleq (f * g)[k_1,k_2]$$

$$= \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f_\infty[i,j] \cdot g_\infty[k_1-i,k_2-j]$$

$$= (g * f)[k_1,k_2], \qquad \forall k_1,k_2 \in \mathbb{Z} \tag{9}$$

Pri čemu važi:

$$f_\infty[i,j] = \begin{cases} f[i,j], 0 \le i \le H_1, 0 \le j \le W_1, \\ 0, inače \end{cases} \tag{10}$$

$$g_\infty[i,j] = \begin{cases} g[i,j], 0 \le i \le H_2, 0 \le j \le W_2, \\ 0, inače \end{cases} \tag{11}$$

Potpuna linearna konvolucija će biti matrica veličine (H_1+H_2-1, W_1+W_2-1). Kružno konvoluciono množenje nad matricama f i g se definiše na sledeći način:

$$h_P[k_1,k_2] \triangleq \left(f *_{P_1,P_2} g\right)[k_1,k_2]$$

$$= \sum_{i=0}^{\mathrm{P_H}-1} \sum_{j=0}^{P_W-1} f[i,j] \cdot g[k_1-i,k_2-j]$$

$$= \left(g *_{P_H,P_W} f\right)[k_1,k_2], \qquad \forall k_1,k_2 \in \mathbb{Z} \tag{12}$$

Pri čemu važi:

$$f_{P_H,P_W}[i,j] = \sum_{p_1=-\infty}^{\infty} \sum_{p_2=-\infty}^{\infty} f_\infty[i+p_1 \cdot P_H, j+p_2 \cdot P_W],$$

$$= f_\infty[i \bmod P_H, j \bmod P_W], \qquad \forall i,j \in \mathbb{Z} \tag{13}$$

$$g_{P_H,P_W}[i,j] = \sum_{p_1=-\infty}^{\infty} \sum_{p_2=-\infty}^{\infty} g_\infty[i+p_1 \cdot P_H, j+p_2 \cdot P_W],$$

$$= g_\infty[i \bmod P_H, j \bmod P_W], \qquad \forall i,j \in \mathbb{Z} \tag{14}$$

I u dvodimenzionalnom slučaju se može pokazati da kružna i linearna konvolucija predstavljaju identičnu operaciju ukoliko važi da je $P_H \geq H_1 + H_2$ -1 i $P_W \geq W_1 + W_2 - 1$.

2.3 Diskretna Furijeova transformacija u jednoj dimenziji

Diskretna Furijeova transformacija nad diskretnim signalom $f[n]$ dužine N se definiše na sledeći način:

$$F[k] = \mathcal{F}\{f\}[k] \triangleq \sum_{n=0}^{N-1} f[n] e^{-\frac{2i\pi kn}{N}} = \sum_{n=0}^{N-1} f[n] W_N^{kn} \tag{15}$$

Vrednosti kompleksne eksponencijalne funkcije sa kojima se množe odbirci diskretnog signala $f[n]$ nazivaju se rotacioni faktori (eng. *twiddle factor*). Rotacioni faktori imaju sledeće osobine, koje će biti značajni za izvođenje brze Furijeove transformacije [3]:

- Kompleksno konjugovana simetričnost

$$W_N^{k(N-n)} = W_N^{-kn} = \left(W_N^{kn}\right)^C \tag{16}$$

- Periodičnost

$$W_N^{kn} = W_N^{k(n+N)} = W_N^{(k+N)n} \tag{17}$$

- Redundantnost

$$W_N^{2kn} = W_{\frac{N}{2}}^{kn} \tag{18}$$

Inverzna diskretna Furijeova transformacija nad istim signalom se definiše na sledeći način:

$$f[n] = \mathcal{F}^{-1}\{F\}[n] \triangleq \frac{1}{N} \sum_{k=0}^{N-1} F[k] e^{\frac{2i\pi kn}{N}} = \frac{1}{N} \left(\sum_{k=0}^{N-1} F[k]^C e^{-\frac{2i\pi kn}{N}} \right)^C \tag{19}$$

Iz formule za inverznu Furijeovu transformaciju se može zaključiti da Furijeova transformacija nad konjugovanim signalom u frekvencijskom domenu predstavlja inverznu Furijeovu transformaciju. Konjugacija nad rezultujućim signalom nije neophodna, pošto su za dati algoritam značajne samo realne vrednosti inverzne Furijeove transformacije. Korišćenjem date osobine se povećava konciznost koda, i omogućava se korišćenje već realizovane funkcije za direktnu Furijeovu transformaciju, sa neznatnim usporavanjem vremena izvršavanja. U embeded aplikacijama, sa memorijskim ograničenjima, ova osobina se može pokazati veoma korisnom.

2.4 Diskretna Furijeova transformacija u dve dimenzije

Diskretna Furijeova transformacija nad diskretnim višedimenzionalnim signalom se definiše analogno kao u jednodimenzionalnom slučaju. Formula za Furijeovu transformaciju u dve dimenzije nad matricom *f* dimenzija (*N*, *M*) glasi:

$$F[k,l] = \mathcal{F}\{f\}[k,l]$$

$$\triangleq \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} f[n,m]e^{-\frac{2i\pi lm}{M}}e^{-\frac{2i\pi kn}{N}}$$

$$= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} f[n,m]W_M^{lm}W_N^{kn}$$

$$= \sum_{n=0}^{N-1}\left(\sum_{m=0}^{M-1} f[n,m]W_M^{lm}\right)W_N^{kn} \tag{20}$$

Data jednačina pokazuje kako se pomoću jednodimenzionalne diskretne Furijeove transformacije može dobiti dvodimenzionalna Furijeova transformacija. Prvo se izvrši Furijeova transformacija nad svim kolonama matrice (dvodimenzionalnog signala), a potom po svim vrstama, ili obratno, da bi se dobila željena dvodimenzionalna transformacija. Ova osobina je značajna jer pojednostavljuje implementaciju algoritma i može se primeniti na beskonačno dimenzija.

Inverzna Furijeova transformacija u dve dimenzije se može realizovati izvšavanjem dvodimenzionalne Furijeove transformacije nad konjugovanom matricom:

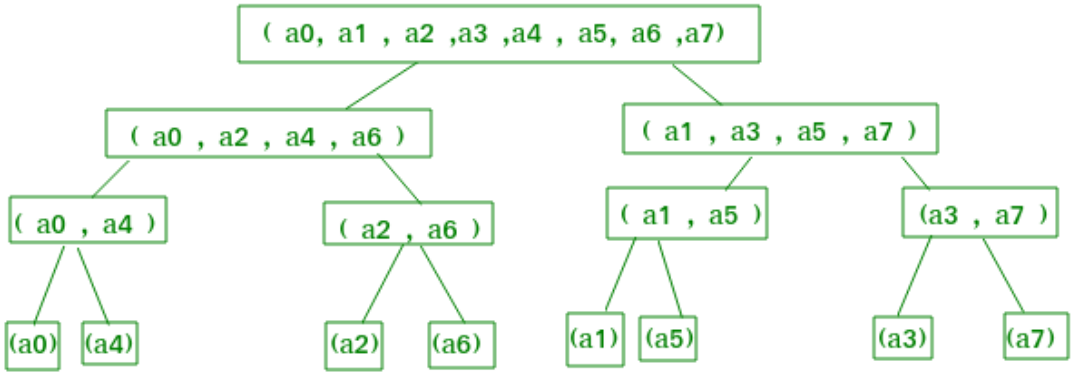
$$f[n,m] = \mathcal{F}^{-1}\{F\}[n,m] = (\mathcal{F}\{F^C\}[n,m])^C \tag{21}$$

2.5 Brza Furijeova transformacija

Postoji veliki broj algoritama koji smanjuju vreme izvršavanja jednodimenzionalne Furijeove transformacije sa $\mathcal{O}(N^2)$ na $\mathcal{O}(N\log N)$, kao što su Kuli-Tukijev Radiks 2 algoritam (eng. *Radix 2*), Prajm faktor algoritam, Raderov algoritam, Vinogradov algoritam i drugi. U ovom radu će biti implementiran Radiks 2 algoritam sa preuređivanjem u vremenu, pošto je jedan od najstarijih i najpoznatijih algoritama za brzu Furijeovu transformaciju.

Radiks 2 postiže ubrzanje u odnosu na diskretnu Furijeovu transformaciju izbegavanjem ponovnog računanja određenih izraza. Pripada klasi „zavadi pa vladaj“ algoritama (eng. *divide and conquer*) i kao preduslov neophodno je da broj članova niza bude stepen dvojke. Tokom preprocesiranja, izmenimo redosled članova niza pomoću algoritma za obrtanje redosleda bita [4]. Potom, rekurzivnim ponavljanjem, podelimo članove na parne i neparne, izračunamo brzu Furijeovu transformaciju za oba

novodobijena niza i spojimo rezultate, koristeći osobine korena jedinice (rotacionih faktora) u polju kompleksnih brojeva (slika 2).



Slika 2 Rekurzivno izvršavanje brze Furijeove transformacije (preuzeto iz [4])

U narednih nekoliko koraka će biti izvedene formule za Radiks 2 algoritam brze Furijeove transformacije:

$$F[k] = \mathcal{F}\{f\}[k] = \sum_{n=0}^{N-1} f[n] W_N^{nk}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} f[2n] W_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} f[2n+1] W_N^{(2n+1)k}$$

$$= \sum_{n=0}^{\frac{N}{2}-1} f[2n] W_N^{2nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} f[2n+1] W_N^{2nk},$$

$$k \in [0, \dots, N-1] \qquad (22)$$

Na osnovu osobine redundantnosti, a potom i periodičnosti rotacionih faktora, formula za Furijeovu transformaciju postaje:

$$F[k] = \sum_{n=0}^{\frac{N}{2}-1} f[2n] W_{N/2}^{nk} + W_N^k \sum_{n=0}^{\frac{N}{2}-1} f[2n+1] W_{N/2}^{nk}$$

$$= F_p[k] + W_N^k F_n[k], \qquad k \in \left[0, \dots, \frac{N}{2} - 1\right] \qquad (23)$$

Na osnovu osobine kompleksno konjugovane simetričnosti, redundantnosti i periodičnosti, može se jednostavno, na osnovu već izračunatih vrednost, dobiti vrednost člana $F[k+N/2]$:

$$F\left[k+\frac{N}{2}\right] = F_p\left[k+\frac{N}{2}\right] + W_N^{k+\frac{N}{2}} F_n\left[k+\frac{N}{2}\right]$$

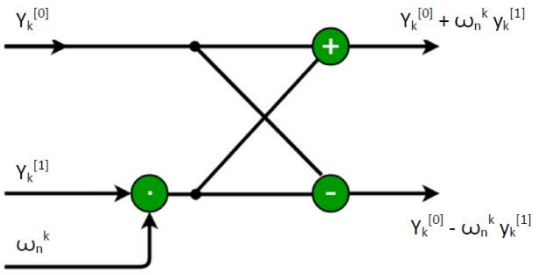
$$= F_p[k] - W_N^k F_n[k], \qquad k \in \left[0, \ldots, \frac{N}{2}-1\right] \tag{24}$$

Iz poslednjih jednačina možemo primetiti rekurzivnu prirodu Furijeove transformacije i konačan izgled Radiks 2 algoritma:

$$F[k] = F_p[k] + W_N^k F_n[k] \tag{25}$$

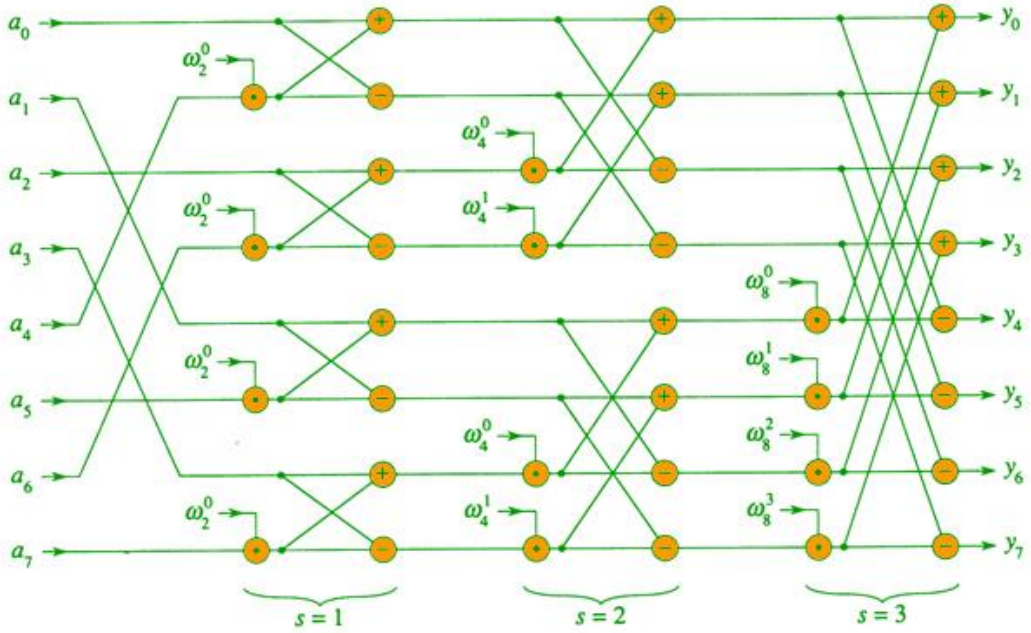
$$F\left[k+\frac{N}{2}\right] = F_p[k] - W_N^k F_n[k] \tag{26}$$

One nam omogućavaju da značajno uštedimo broj upotrebljenih operacija. Operacije računanja vrednosti $F[k]$ i $F[k+N/2]$, odnosno množenje neparnog člana sa rotacionim faktorom, sabiranje i oduzimanje sa parnim članom se naziva leptir (eng. *butterfly*) operacija, slika 3.



Slika 3 Leptir operacija (preuzeto iz [4])

Na slici 4 je prikazan postupak dobijanja Furijeove transformacije niza sa osam članova pomoću Radiks 2 algoritma.



Slika 4 Primer brze Furijeove transformacije za signal sa osam elemenata (preuzeto iz [4])

2.6 Odnos konvolucije i Furijeove transformacije

Neka su *f* i g dva niza veličine *N* i *M*, redom, i neka je $P \geq N + M - 1$ moduo kružne konvolucije. Tada formula koja spaja kružnu konvoluciju sa diskretnom Furijeovom transformacijom glasi:

$$(f *_P g)[k] = \mathcal{F}^{-1}\{\mathcal{F}\{f_P\} \cdot \mathcal{F}\{g_P\}\}[k] \tag{27}$$

Formula se može dokazati računanjem Furijeove transformacije kružne konvolucije signala *f* i g.

$$\mathcal{F}\{(f *_P g)\}[k] = \sum_{i=0}^{P-1} \left((f *_P g)[i] W_P^{ki}\right)$$

$$= \sum_{i=0}^{P-1} \sum_{j=0}^{P-1} f_P[j] \cdot g_P[i-j] W_P^{ki}$$

$$= \sum_{j=0}^{P-1} f_P[j] W_P^{kj} \sum_{i=0}^{P-1} g_P[i-j] W_P^{k(i-j)}$$

$$= \sum_{j=0}^{P-1} f_P[j] W_P^{kj} \sum_{i=0}^{P-1} g_P[i] W_P^{ki}, (zbog\ periodičnosti\ g_P)$$

$$= \mathcal{F}\{f_P\}[k] \cdot \mathcal{F}\{g_P\}[k] \tag{28}$$

Povezivanjem date formule sa odnosom između potpune linearne i kružne konvolucije, dobijemo formulu za računanje potpune linearne konvolucije korišćenjem Furijeovih transformacija:

$$h[k] = \sum_{i=\max(0,k-M+1)}^{\min(N-1,k)} f[i] \cdot g[k-i] = (f *_P g)[k]$$

$$= \mathcal{F}^{-1}\{\mathcal{F}\{f_P\}\mathcal{F}\{g_P\}\}[k], \qquad \forall k \in [0, N+M-2] \tag{29}$$

Glava 3

Projektovanje aplikacije na sistemskom nivou

3.1 ESL Metodologija

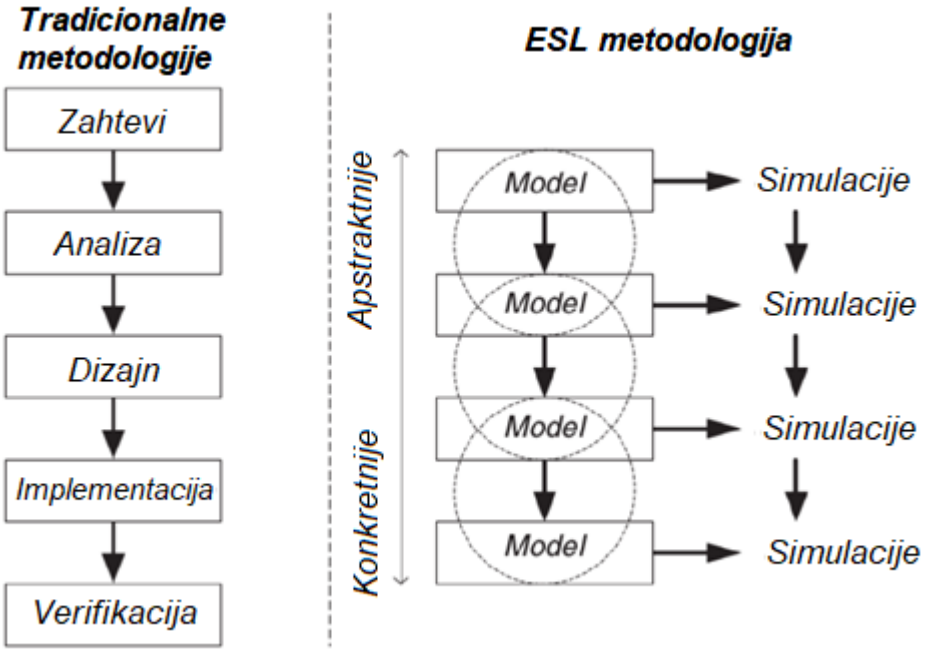
ESL (eng. *Electronic System Level*) metodologija predstavlja dizajn metodologiju za razvoj elektronskih uređaja. Zasnovana je na projektovanju modela uređaja na visokom nivou apstrakcije (odnosno na zahtevima poslodavca ili tržišta) i njegovim postepenim profinjavanjem do prostornog rasporeda svih hardverskih komponenti u uređaju i signala koji ih povezuju.

Korišćenje ESL metodologije je motivisano činjenicom da tradicionalne metodologije za razvoj elektronskih uređaja ne uspevaju da ispune zadate zahteve prilikom realizacije složenih elektronskih uređaja [5]:

- Preko 70% dizajniranih elektronskih uređaja ima barem 30% niže performanse od zadatih.
- Preko 30% dizajniranih elektronskih uređaja nema korektno realizovano barem 50% zadate funkcionalnosti.
- Oko 54% projekata zakasni sa rokom realizacije, pri čemu je prosek kašnjenja četiri meseca.
- Oko 13% započetih projekata se prekine pre njihovog završetka.

Među glavnim razlozima za neuspeh realizacije dizajniranih elektronskih uređaja pomoću tradicionalnih metodologija su ograničena mogućnost preglednosti dizajna i kontrole njegovog izvršavanja. Da bi se postiglo povećanje kontrolabilnosti i preglednosti, moraju se meriti karakteristike individualnih signala i logičkih kapija, sa preciznošću reda veličine jedne nanosekunde. Datim nivoima apstrakcije se dobijaju ogromne količine neupotrebljivih informacija.

ESL metodologija omogućava razvoj elektronskih uređaja na sistemskom nivou, odnosno celokupno izvršavanje dizajna se apstrahuje u formi procesorskih instrukcija, poziva funkcija, pristupa memoriji i slanja paketa podataka. Na datom nivou se najjednostavnije i najbrže particioniše dizajn na hardverske i softverske komponente i biraju IP blokovi. Prednosti ESL metodologije su povećani kvalitet i pouzdanost, realizacija optimalnih dizajnova elektronskih uređaja, smanjenje vremena do plasiranja rešenja na tržište (eng. *time to market*) i mogućnost konkurentnog razvoja softvera i hardvera.



Slika 5 Odnos između tradicionalnih i ESL metodologije razvoja elektronskih uređaja (preuzeto iz [5])

ESL metodologija se sastoji iz sedam koraka:

1. **Pisanje specifikacije i modelovanje** – predstavlja razvoj dokumenta koji opisuje sistem ili funkcionalnost koju uređaj treba da ispuni, zajedno sa njegovim ograničenjima. Specifikacija na najvišem nivou je često napisana prirodnim jezikom, čime se ostavlja dovoljno prostora da model na sledećem, nižem nivou apstrakcije zadovolji zahtevane uslove bez da bude ograničen na specifičnu implementaciju. Za razliku od specifikacije napisane prirodnim jezikom, specifikacije koje se izvršavaju iskazuju željeno ponašanje na nedvosmislen način iz perspektive crne kutije. S druge strane, implementacije koje se izvršavaju reflektuju mikroarhitekturne odluke. Postoji veliki broj programskih jezika koji se koriste za ESL specifikaciju, u koje se ubrajaju *MATLAB M-Code*, *SystemC* i drugi.
2. **Analiza pre particionisanja** – predstavlja proces upoređivanja karakteristika različitih implementacija. U ključne karakteristike se ubrajaju:
    - Vreme izvršavanja – algoritam može biti serijski ili konkurentno realizovan
    - Memorijski zahtevi
    - Potrošnja - najviše utiču spekulativno izvršavanje, keširanje podataka i izbor algoritma
    - Vreme potrebno da bi se plasirao proizvod na tržište – kašnjenje može da uzrokuje veoma velike finansijske gubitke

- Složenost – pravljenje kompromisa između performansi i mogućnosti verifikacije rada

Da bi se mogle uporediti karakteristike mogućih implementacija, mogu se vršiti različite analize, kao što su statička, dinamička, algoritamska, i druge. U ovom radu će se vršiti profajliranje koda, na osnovu kojeg će se izvršiti particionisanje hardvera i softvera.

3. **Particionisanje** – predstavlja proces raspodele algoritama (ili njihovih delova), definisanih u specifikaciji, na one koji će biti realizovani u softverskim (procesori), odnosno hardverskim komponentama. Pravi se arhitekturni opis jedinica za procesiranje, memoriju i komunikaciju. Hardverska particija donosi izbore vezane za broj korišćenih jezgara (distribuirani sistem) i da li će jezgra biti upravljana aperiodičnim događajima ili periodičnim signalima (klokom). Softversko particionisanje se odnosi na izbor operativnog sistema, biblioteka, tipa procesora (simetrični ili heterogeni), i na nižem procesorskom nivou na niti i procese. Konačno, u slučaju distribuiranog sistema, mora biti izabran algoritam za planiranje (eng. *scheduler*).
4. **Analiza i debagovanje nakon particionisanja** – predstavlja proces ponovne analize karakteristika i upoređivanja sa karakteristikama dobijenim u analizi pre particionisanja. Algoritmi koji će biti izvršavani na procesoru su napisani u nekom programskom jeziku (na primer *C*, ili *C++*), dok su algoritmi koji će biti izvršavani u hardveru realizovani u nekom HDL (eng. *Hardware Description Language*) jeziku. U ovom koraku često može doći do ponovnih iteracija između particionisanja i analize nakon particionisanja, sa ciljem optimizacije performansi.
5. **Verifikacija nakon particionisanja** – predstavlja proces ispitivanja rada dizajna, sa ciljem da se utvrdi da li je željena funkcionalnost odgovarajućih komponenti pre particionisanja sačuvana nakon particionisanja dizajna. Tipičan tok verifikacije predstavlja planiranje verifikacije, implementacija verifikacionog okruženja i na kraju analiza verifikacionih rezultata. Uz pomoć formalne analize i simulacije, istražuje se prostor stanja u kojem se dizajn može naći. Verifikaciono okruženje se implementira pomoću HLVL (eng. *High Level Verification Languages*) jezika. Ovo poglavlje će detaljnije biti razjašnjeno u glavi 5.
6. **Hardversko-softverska implementacija** – Hardverska implementacija predstavlja proces kreiranja modela koji se mogu sintetizovati do nivoa kapija, i obično se pišu na RT nivou. Postoji pet mogućih hardverskih implementacija, a to su ekstenzibilni procesori (eng. *extensible processors*), DSP (eng. *Digital Signal Processing*) koprocesori, VLIW (eng. *Very Long Instruction Word*) koprocesori, namenski (eng. *application specific*) koprocesori, i ASIC ili FPGA. Softverska implementacija se može klasifikovati prema vidljivosti sistemskih komponenti (klok, uređaj za

enkripciju i slično), vremenskim intervalima između susednih događaja, platforme na kojoj se izvršava i vremenskih performansi.

7. **Verifikacija implementacije** – predstavlja verifikaciju ispravnog rada celokupnog sistema. Može se postići simulacijom mešovitih modela, hardverskom emulacijom, ili realizacijom prototipa pomoću FPGA. Radi lakšeg debagovanja hardvera, često se ugrađuju komponente kao što su JTAG (eng. *Joint Test Action Group*) i ILA (eng. *Integrated Logic Analyzer*) komponente.

3.2 Specifikacija i modelovanje

Potpuna linearna konvolucija u dve dimenzije je definisana jednačinom (9). Radi ilustracije, data matematička operacija će biti izvršena nad dve proizvoljne matrice A i B veličine 3x3. Rezultujuća matrica C ima veličinu 5x5 (tabela 1).

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} \\ a_{1,0} & a_{1,1} & a_{1,2} \\ a_{2,0} & a_{2,1} & a_{2,2} \end{bmatrix} \quad B = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} \\ b_{1,0} & b_{1,1} & b_{1,2} \\ b_{2,0} & b_{2,1} & b_{2,2} \end{bmatrix} \quad C = \begin{bmatrix} c_{0,0} & \cdots & c_{0,4} \\ \vdots & \ddots & \vdots \\ c_{4,0} & \cdots & c_{4,4} \end{bmatrix}$$

$c_{0,0} = a_{0,0} \cdot b_{0,0}$	$c_{0,1} = a_{0,0} \cdot b_{0,1} + a_{0,1} \cdot b_{0,0}$	$c_{0,2} = a_{0,0} \cdot b_{0,2} + a_{0,1} \cdot b_{0,1} + a_{0,2} \cdot b_{0,0}$	$c_{0,3} = a_{0,1} \cdot b_{0,2} + a_{0,2} \cdot b_{0,1}$	$c_{0,4} = a_{0,2} \cdot b_{0,2}$
$c_{1,0} = a_{0,0} \cdot b_{1,0} + a_{1,0} \cdot b_{0,0}$	$c_{1,1} = a_{0,0} \cdot b_{1,1} + a_{0,1} \cdot b_{1,0} + a_{1,0} \cdot b_{0,1} + a_{1,1} \cdot b_{0,0}$	$c_{1,2} = a_{0,0} \cdot b_{1,2} + a_{0,1} \cdot b_{1,1} + a_{0,2} \cdot b_{1,0} + a_{1,0} \cdot b_{0,2} + a_{1,1} \cdot b_{0,1} + a_{1,2} \cdot b_{0,0}$	$c_{1,3} = a_{0,1} \cdot b_{1,2} + a_{0,2} \cdot b_{1,1} + a_{1,1} \cdot b_{0,2} + a_{1,2} \cdot b_{0,1}$	$c_{1,4} = a_{0,2} \cdot b_{1,2} + a_{1,2} \cdot b_{0,2}$
$c_{2,0} = a_{0,0} \cdot b_{2,0} + a_{1,0} \cdot b_{1,0} + a_{2,0} \cdot b_{0,0}$	$c_{2,1} = a_{0,0} \cdot b_{2,1} + a_{0,1} \cdot b_{2,0} + a_{1,0} \cdot b_{1,1} + a_{1,1} \cdot b_{1,0} + a_{2,0} \cdot b_{0,1} + a_{2,1} \cdot b_{0,0}$	$c_{2,2} = a_{0,0} \cdot b_{2,2} + a_{0,1} \cdot b_{2,1} + a_{0,2} \cdot b_{2,0} + a_{1,0} \cdot b_{1,2} + a_{1,1} \cdot b_{1,1} + a_{1,2} \cdot b_{1,0} + a_{2,0} \cdot b_{0,2} + a_{2,1} \cdot b_{0,1} + a_{2,2} \cdot b_{0,0}$	$c_{2,3} = a_{0,1} \cdot b_{2,2} + a_{0,2} \cdot b_{2,1} + a_{1,1} \cdot b_{1,2} + a_{1,2} \cdot b_{1,1} + a_{2,1} \cdot b_{0,2} + a_{2,2} \cdot b_{0,1}$	$c_{2,4} = a_{0,2} \cdot b_{2,2} + a_{1,2} \cdot b_{1,2} + a_{2,2} \cdot b_{0,2}$

$c_{3,0}$ $= a_{1,0} \cdot b_{2,0}$ $+ a_{2,0} \cdot b_{1,0}$	$c_{3,1}$ $= a_{1,0} \cdot b_{2,1}$ $+ a_{1,1} \cdot b_{2,0}$ $+ a_{2,0} \cdot b_{1,1}$ $+ a_{2,1} \cdot b_{1,0}$	$c_{3,2}$ $= a_{1,0} \cdot b_{2,2}$ $+ a_{1,1} \cdot b_{2,1}$ $+ a_{1,2} \cdot b_{2,0}$ $+ a_{2,0} \cdot b_{1,2}$ $+ a_{2,1} \cdot b_{1,1}$ $+ a_{2,2} \cdot b_{1,0}$	$c_{3,3}$ $= a_{1,1} \cdot b_{2,2}$ $+ a_{1,2} \cdot b_{2,1}$ $+ a_{2,1} \cdot b_{1,2}$ $+ a_{2,2} \cdot b_{1,1}$	$c_{3,4}$ $= a_{1,2} \cdot b_{2,2}$ $+ a_{2,2} \cdot b_{2,1}$
$c_{4,0}$ $= a_{2,0} \cdot b_{2,0}$	$c_{4,1}$ $= a_{2,0} \cdot b_{2,1}$ $+ a_{2,1} \cdot b_{2,0}$	$c_{4,2}$ $= a_{2,0} \cdot b_{2,2}$ $+ a_{2,1} \cdot b_{2,1}$ $+ a_{2,2} \cdot b_{2,0}$	$c_{4,3}$ $= a_{2,1} \cdot b_{2,2}$ $+ a_{2,2} \cdot b_{2,1}$	$c_{4,4}$ $= a_{2,2} \cdot b_{2,2}$

Tabela 1 Ilustracija dvodimenzionalne konvolucije

3.3 Analiza pre particionisanja

U ovom koraku je izvršeno upoređivanje performansi različitih algoritama i delova algoritma. Algoritam koji računa dvodimenzionalnu konvoluciju po definiciji, ima mnogo veću vremensku složenost nego algoritam zasnovan na brzim Furijeovim transformacijama [2]. S druge strane, korišćenje brzih Furijeovih transformacija zahteva preko duplo više memorijskih lokacija, koje su zadužene za skladištenje vrednosti imaginarnih članova matrica i rotacionih faktora. Zbog značajne uštede vremena izvršavanja, izabrana je implementacija zasnovana na brzoj Furijeovoj transformaciji.

Profajliranjem izvornog koda napisanog u C++ jeziku dobijeni su sledeći rezultati odnosa vremena izvršavanja konvolucije matrica i brze Furijeove transformacije za date matrice:

Veličina matrica	Relativno vreme izvršavanja ostatka algoritma konvolucije [%]	Relativno vreme izvršavanja brze Furijeove transformacije [%]
8x8	31.02	68.98
128x128	6.76	93.24

Tabela 2 Rezultati profajliranja

Za još veće matrice, odnos vremena izvršavanja brze Furijeove transformacije u dve dimenzije i konvolucije će biti jos veći.

3.4 Particionisanje

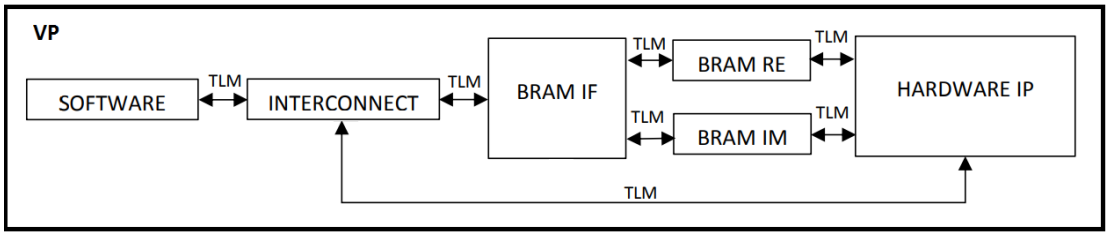
Dvodimenzionalna Furijeova transformacija je vremenski najsporiji deo algoritma konvolucije. Zbog toga je odlučeno da se Furijeova transformacija, kao i inverzna Furijeova transformacija, izvršava na hardveru, dok će se ostatak algoritma konvolucije („omotač“ oko Furijeove transformacije) izvršavati u softveru. Realni brojevi u hardveru će biti predstavljeni sa fiksnom tačkom.

Ciljana platforma u ovom sistemu će biti *Zybo* ploča iz familije *Zynq-7010*. *Zybo* predstavlja sistem na čipu koji se sastoji iz programabilne logike (FPGA) i procesirajućeg dvojezgrarnog sistema ARM *Cortex A9*. Pošto nije zamišljeno da se ubrzava konvolucija sa paralelizacijom softverske particije, koristiće se samo jedno jezgro procesora. Operativni sistem na ploči će biti *Xilinx Linux Kernel 4.10*.

Matrice koje se obrađuju će biti skladištene u BRAM-u. Direktna komunikacija između softverskog i hardverskog dela će se obavljati preko AXI *Lite* magistrale. Particije će razmenjivati informacije o veličini matrica preko date magistrale, kao i signale za početak i završetak obrade matrica. Softverska particija komunicira sa BRAM-om putem AXI-BRAM kontrolera, dok je hardverska particija direktno povezana putem BRAM interfejsa sa memorijom.

3.5 Implementacija

Model je implementiran u programskom jeziku *SystemC*. Komponente su instancirane u vidu klasa, i mogu se podeliti na komponente zadužene za skladištenje memorije (BRAM RE i BRAM IM), softversku i hardversku particija (*Hard* i *Soft*) i komponente koje povezuju date module (*Interconnect* i *BRAM Controller*).



Slika 6 Šematski prikaz virtuelne platforme

Sve komponente komuniciraju na nivou transakcija putem TLM (eng. *Transaction Level Modelling*) mehanizma [6]. Komunikacija na nivou transakcija omogućava razdvajanje implementacije protokola od funkcionalnosti koju treba da obavljaju komunikacioni kanali. Na osnovu toga se jednostavno mogu menjati protokoli u skladu sa zahtevima dizajna. Transakcija koja se prenosi putem TLM mehanizma sadrži nekoliko parametara, od kojih su najznačajniji dužina niza koji se šalje (svi podaci su osmobitni), pokazivač na niz i komanda koja je oblika „čitaj“ ili „upiši“.

Komponenta *Soft* vrši implementaciju konvolucije, odnosno šalje zahteve hardverskom modulu da izvrši dvodimenzionalne Furijeove transformacije nad matricama, obavlja zamenu redosleda matrica u memoriji, kao i jednostavne operacije nad elementima matrica. Pseudo kod je napisan u listingu 1.

1. Slanje zahteva hardverskoj particiji za izvršavanje dvodimenzione Furijeove transformacije nad ulaznom realnom matricom A
2. Zamena redosleda obrađene matrice A i matrice B
3. Slanje zahteva hardverskoj particiji za izvršavanje dvodimenzione Furijeove transformacije nad ulaznom realnom matricom B
4. Množenje elemenata novih matrica A i B i skladištenje na lokaciji matrice B (zove se matrica C) i invertovanje vrednosti imaginarnog dela matrice C
5. Slanje zahteva hardverskoj particiji za izvršavanje dvodimenzione Furijeove transformacije nad ulaznom realnom matricom C
6. Deljenje svakog elementa sa proizvodom visine i širine matrice C

Listing 1 Pseudo-kod algoritma koji izvršava *Soft* komponenta

Hard komponenta izvršava dvodimenzionalnu Furijeovu transformaciju, koja je opisana u poglavljima 2.4 i 2.5.

Interconnect komponenta služi za rutiranje podataka koji šalje komponenta *Soft*, i na osnovu prvih osam bita tridesetdvobitne adrese utvrđuje da li da transakciju prosledi memoriji ili direktno u hardverski blok. BRAM kontroler prima transakciju iz *Interconnect* komponente. Transakcija koja stiže u BRAM kontroler sadrži niz dvostruke dužine, pri čemu prva polovina niza predstavlja lokaciju za podatke koji treba da se učitaju ili upišu u memoriju sa realnim podacima, dok druga polovina niza je rezervisana za memoriju sa imaginarnim podacima. Stoga, može se reći da BRAM kontroler jednu transakciju deli na dva dela, i ponovo sastavlja transakcije nakon uspešnog pisanja ili čitanja.

Memorijske komponente BRAM RE i BRAM IM skladište vrednosti dve kompleksne matrice. Prva polovina memorijskih lokacija je rezervisana za jednu matricu (u početku se to odnosi na matricu A, do zamene redosleda), dok je druga polovina rezervisana za drugu matricu. Da bi se omogućilo istrovremeno upisivanje realnog i imaginarnog dela (i time uštelo na vremenu), realni i imaginarni delovi matrica su razdeljeni u dva BRAM-a.

Ceo sistem se zove virtuelna platforma [7]. Klasa VP (eng. *Virtual Platform*) instancira sve komponente, i spaja odgovarajuće interfejse jedne sa drugima.

3.6 Analiza modela posle particionisanja

Realne vrednosti matrica su reprezentovane sa fiksnom tačkom. Da bi se utvrdila optimalna širina celog dela i dela iza fiksne tačke, neophodno je izvršiti analizu širine. Napisana su dva koda sa identičnim algoritmom, u *C++* i *SystemC* programskim jezicima, pri čemu se u prvom slučaju koristi reprezentacija realnih brojeva sa pokretnom tačkom, dok u drugom slučaju se koristi reprezentacija sa fiksnom tačkom. Vršena je analiza nad realnim matricama veličine 128x128 proizvoljnih realnih vrednosti i mereno je maksimalno odstupanje između odgovarajućih elemenata u rezultujućim matricama.

Širina <ceolobrojni deo, realni deo>	Odstupanje
<32,12>	12288
<32,13>	8192.02
<32,14>	16384
<32,15>	0.0273132
<32,16>	0.0311432
<32,17>	0.0534661
<32,18>	0.118255
<48,18>	0.0233966
<48,19>	0.023397
<48,20>	0.023398
<48,21>	0.0234

Tabela 3 Analiza širine

Po datim rezultatima, širina od 48 bita, sa 18 bita koji se nalaze iza fiksne tačke, predstavlja minimalnu širinu realnih brojeva za matrice veličine 128x128. Može se primetiti da ukoliko je premali broj bita iza fiksne tačke, usled odsecanja realnog dela (eng. *truncation*) dolazi do nastanka velike greške. Sa druge strane, povećavanjem broja bita iza fiksne tačke, neznatno se povećava odstupanje. Za reprezentaciju realnih brojeva, u ovom projektu je izabrana širina od 32 bita, sa širinom realnog dela od 16 bita, jer nema veliku grešku i zato što vrednosti predstavljaju stepen broja 2, što olakšava rad sa Linuks drajverom. Za veće matrice od 128x128 bi širina realnih brojeva morala biti veća.

Glava 4

Projektovanje hardverskog IP jezgra

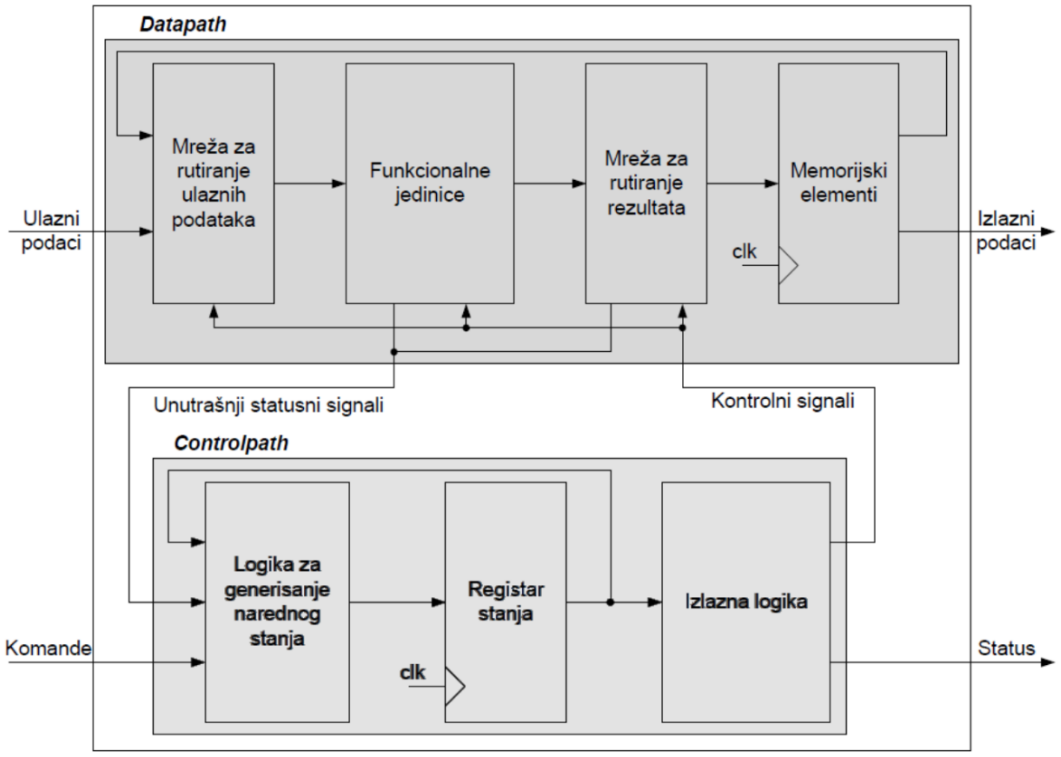
4.1 Uvod

4.1.1 RT metodologija

Prevođenje algoritma, odnosno detaljne sekvence akcija ili koraka neophodnih da se reši problem, u digitalno elektronsko kolo, se vrši pomoću RT metodologije. **RT** (eng. *Register Transfer*) **metodologija** [8] predstavlja formalizovani postupak pomoću kojega se vrši mapiranje varijabli u registre, i sekvenci koraka nad varijablama u manipulacije nad registrima i signalima (RT operacije). RT metodologija podrazumeva projektovanje dva podsistema:

- **Podsistem za obradu podataka** (eng. *Data path*) je modul koji sadrži sve hardverske resurse neophodne da bi se izvršili svi koraci definisani algoritmom i koji će biti izvršeni u datom IP jezgru, svih memorijskih resursa neophodnih da bi se skladištile varijable algoritma, kao i mreže za rutiranje koja na željeni način povezuje date hardverske komponente. Podsistem za obradu podataka se sastoji iz:
    - Mreže za rutiranje ulaznih podataka – dato kombinaciono digitalno elektronsko kolo rutira ulazne signale i vrednosti memorijskih elemenata sa odgovarajućim funkcionalnim jedinicama. Uglavnom je realizovano pomoću multipleksera.
    - Mreže za rutiranje rezultata – dato kombinaciono digitalno elektronsko kolo rutira izlaze funkcionalnih jedinica sa odgovarajućim memorijskim elementima. Uglavnom se realizuje pomoću multipleksera.
    - Funkcionalnih jedinica – sastoje se iz svih aritmetičkih, logičkih i relacionih operatora implementiranih u obliku kombinacionih mreža.
    - Memorijskih elemenata koji skladište sve podatke koji se koriste u toku izvršavanja algoritma. Najčešće su implementirani pomoću registara, mada mogu se koristiti registarske banke, jednopristupne i višepristupne memorije, FIFO (eng. *first in first out*) baferi i drugi elementi.
- **Upravljački podsistem** (eng. *Control path*) je modul čija je uloga da upravlja redosledom operacija koje se izvršavaju unutar podsistema za obradu podataka i signalima koji se u datom trenutku skladište u memoriji, u skladu sa koracima definisanim algoritmom. Upravljački podsistem se stoga najčešće realizuje kao konačni automat stanja.

Konačni automat stanja (eng. *Finite State Machine*, *FSM*), predstavlja digitalno elektronsko kolo sa unutrašnjim stanjima. Tokom vremena FSM menja stanje u kom se nalazi, u zavisnosti od trenutnog stanja i ulaznih signala. Sastoji se iz logike za generisanje narednog stanja, registra stanja i izlazne logike. Grafička reprezentacija konačnog automata stanja se može vršiti pomoću ASM (eng *Algorithmic State Machine*) dijagrama.



Slika 7 Šematski prikaz podsistema za obradu podataka i upravljačkog podsistema

Projektovanje digitalnog sistema koji treba da izvršava željeni algoritam, odnosno podsistema za obradu podataka i upravljačkog podsistema, pomoću RT metodologije, se sastoji iz pet koraka:

1. Eliminacija naredbi ponavljanja – sve naredbe ponavljanja unutar algoritma se moraju zameniti sa odgovarajućim *if* i *goto* naredbama.
2. Definisanje interfejsa digitalnog sistema koji se projektuje – analizom algoritma neophodno je utvrditi ulazne i izlazne promenljive (signale), kao i širine datih signala.
3. Projektovanje upravljačkog podsistema – na osnovu algoritma je neophodno kreirati odgovarajući ASM dijagram, kao i razmotriti potencijalne optimizacije.
4. Projektovanje podsistema za obradu podataka – sastoji se iz četiri koraka:
    a. Identifikacija svih RT operacija koje postoje unutar ASM dijagrama, i dimenzionisanje registara koji će biti pridruženi datim operacijama.
    b. Grupisanje RT operacija prema odredišnim registrima.
    c. Za svaku grupu operacija, izvršavaju se tri koraka:

i. Formira se odredišni registar.
ii. Formiraju se kombinacione mreže koje implementiraju sve funkcionalne transformacije signala u datoj grupi.
iii. Dodaju se multiplekserska i rutirajuća kola ispred odredišnog registra ukoliko je on asociran sa većim brojem RT operacija.

d. Dodaju se kombinacione mreže koje formiraju izlazne statusne signale.

5. Pisanje HDL modela – na osnovu ASM dijagrama dobijenog u trećem koraku može se napisati odgovarajući HDL model upravljačkog podsistema, a na osnovu blok dijagrama formiranog u četvrtom koraku može se napisati HDL model podsistema za obradu podataka. Konačno, objedinjavanjem datih modela, dobija se HDL model kompletnog digitalnog elektronskog kola koje implementira željeni algoritam. U ovom radu korišćen je programski jezik VHDL [9].

4.1.2 IP jezgro

Hardverska IP (eng. *Intellectual Property*) **jezgra** su digitalni sistemi projektovani korišćenjem RT metodologije, koji enkapsuliraju neku jasno definisanu funkcionalnost. Uglavnom predstavljaju gradivne blokove većih i složenijih sistema na čipu. Da bi se olakšala ponovna upotrebljivost jezgara, za razmenu informacija se koriste standardni interfejsi kao što je AXI (eng. *Advanced eXtensible Interface*), PLB, OPB, LocalLink, i drugi. Na osnovu funkcionalnosti koju IP moduli obavljaju, dele se na:

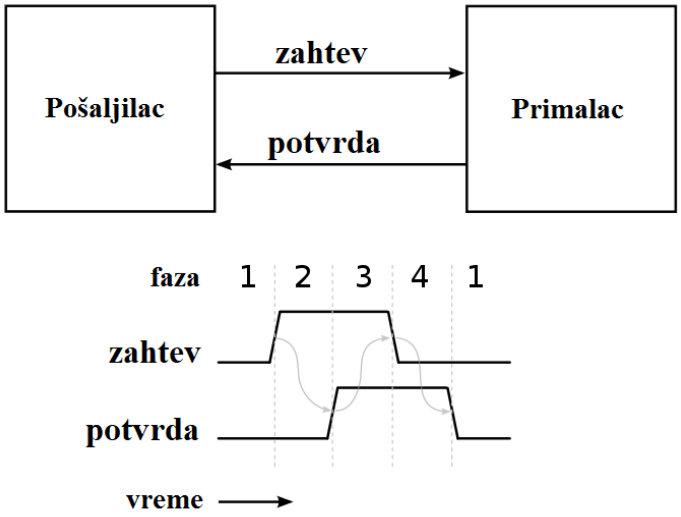
- Module za skladištenje podataka – u ovu grupu spadaju IP jezgra koja implementiraju mehanizme za skladištenje podataka, kao što su SRAM, DRAM, ROM, Flash, FIFO, i drugi.
- Module za prenos podataka - u ovu grupu spadaju IP jezgra koja implementiraju i međusobno prevode različite komunikacione protokole, kao što su AXI, PLB, PCIe, Ethernet, I^2C, SPI, i drugi.
- Module za obradu podataka - u ovu grupu spadaju IP jezgra koja vrše obradu podataka, odnosno algoritme kao što su filtriranje (FIR i IIR filtri), unitarne transformacije (FFT, DCT, DWT...), kompresiju, enkripciju i zaštitno kodovanje podataka, i druge često korišćene specifične algoritme.
- Procesorske module - u ovu grupu spadaju IP jezgra koja takođe vrše obradu podataka, i zapravo pripadaju prethodnoj grupi, ali su složeniji i moguće je implementirati proizvoljni algoritam pomoću njih, pa se stoga uglavnom posmatraju odvojeno. Osnovni tipovi procesorskih modula su skalarni, superskalarni, VLIW, vektorski, višejezgarni i konfigurabilni procesori.

4.1.3 Četvorofazni hendšejk protokol

Četvorofazni hendšejk protokol (eng. *Four-Phase Handshake Protocol*) [8] se koristi za sinhronu i asinhronu razmenu informacija između različitih modula. Protokol se sastoji iz pošaljioca i primaoca (eng. *talker* i *listener*) koji komuniciraju putem signala za zahtev i potvrdu (eng. *request* i *acknowledge*).

Redosled operacija datog protokola se može podeliti na četiri koraka (otuda naziv četvorofazni):

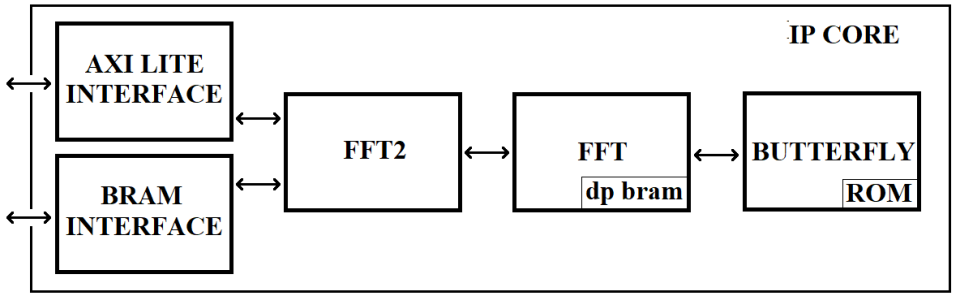
1. Ukoliko su signali za zahtev i potvrdu na niskom logičkom nivou, pošaljilac može da uspostavi komunikaciju, odnosno aktivira signal za zahtev.
2. Kada primalac primeti da je signal za zahtev na visokom logičkom nivou, aktivira signal za potvrdu.
3. Kada pošaljilac primeti da je signal za potvrdu na visokom logičkom nivou, deaktivira signal za zahtev.
4. Kada primalac primeti da je signal za zahtev na niskom logičkom nivou, deaktivira signal za potvrdu.



Slika 8 Šematski prikaz četvorofaznog hendšejk protokola

4.2 Realizovana implementacija i diskusija mogućih implementacija

Algoritam za dvodimenzionalnu Furijeovu transformaciju matrica je ekvivalentan izvršavanju algoritma za jednodionu Furijeovu transformaciju prvo nad svim redovima, a potom nad svim kolonama matrice. IP jezgro komunicira sa dva spoljašnja BRAM-a (eng. *Block Random Access Memory*) u koje su smešteni realni i imaginarni deo matrice, kao i sa procesorom pomoću AXI *Lite* interfejsa, koji mu šalje veličinu matrice i zahtev za početak rada. Unutar FFT modula se nalazi unutrašnji dvopristupni BRAM (jedan pristup za čitanje i jedan pristup za pisanje) u koji se skladišti trenutni red ili kolona, nad kojim se vrši jednodimenzionalna Furijeova transformacija. Moduli FFT2, FFT i BUTTERFLY komuniciraju preko četvorofaznog hendšejk protokola. U ovakvoj konfiguraciji, modul FFT2 ima jedinu ulogu kao konektor između BRAM-a i FFT modula, pri čemu mora da vodi računa o trenutnom redu ili koloni nad kojim se vrši obrada. FFT modul u toku izvršavanja više puta poziva BUTTERFLY modul, koji izvšava leptir operaciju nad dva kompleksna broja. Unutar BUTTERFLY modula se nalazi ROM (eng. *Read Only Memory*) iz kojih se čitaju vrednosti rotacionih faktora.



Slika 9 Šematski prikaz implementiranog IP jezgra za algoritam *fft2*

Postoji nekoliko mogućih razlika u implementaciji koji su razmatrani:

- Korišćenje AXI *Full* interfejsa umesto **BRAM interfejsa** za transport članova realnog i imaginarnog dela kompleksne matrice – izabran je BRAM isključivo zbog jednostavnosti implementacije. AXI *Full* interfejs bi mogao da obavlja daleko kompleksniju komunikaciju između komponenti koja za potrebe ovog projekta nisu potrebne.
- Korišćenje **unutrašnje memorije** za skladištenje međurezultata – umesto čuvanja samo jedne vrste ili kolone matrice, bilo je moguće uopšte ne implementirati unutrašnju memoriju, ili skladištiti i realni i imaginarni deo matrice unutar IP jezgra (dve matrice iste veličine). U prvom slučaju, obavljanje komunikacije sa spoljašnjim BRAM-om bi značajno usporilo vreme izvršavanja. U drugom slučaju, zauzimanje dodatne memorije bi veoma uticalo na ukupnu veličinu matrice koja se može obrađivati, jer su resursi na ploči ograničeni. Izabrana je realizacija sistema u kojem se čuva vrednost samo jedne vrste odnosno kolone matrice, kao optimalan odnos između vremena izvršavanja i korišćenja hardverskih resursa.
- Korišćenje dvofaznog umesto **četvorofaznog hendšejk protokola** za *start* i *ready* signale između različitih modula i za pristup memoriji – iako je dvofazni hendšejk protokol jednostavniji za realizaciju i neznatno brži, izabran je četvorofazni protokol zbog jasnijeg praćenja rada algoritma i eventualnog otklanjanja grešaka u implementaciji celog algoritma. Dodatno, korišćenje protokola za pristup memoriji povećava sigurnost rada, jer dobijamo povratnu informaciju da se vrednost u memoriju zaista i upisala. U slučaju paralelizacije i korišćenja većeg broja istih modula (na primer više *fft* ili *butterfly* modula), može se jednostavnije realizovati modul za arbitražu i raspoređivanje (eng. *scheduling*) zahteva za upis vrednosti u memoriju.
- Računanje **rotacionih faktora** umesto čitanja iz ROM-a – računanje rotacionih faktora podrazumeva implementaciju modula za računanje sinusa i kosinusa, ili skladištenje vrednosti sinusa i kosinusa za različite vrednosti ulaznog signala. U oba slučaja bi se vreme izvšavanja celog algoritma značajno povećalo (u prvom pogotovo), dok bi se u drugom slučaju svakako morale skladištiti vrednosti u memoriju, čime nikakva dobit ne bi bila postignuta.

- **Pajplajnovanje leptir operacije** – pajplajnovanjem izvršavanja date operacije povećavamo broj stanja kroz koja algoritam mora proći (samim tim i ukupan broj taktova), ali smanjujemo trajanje jednog takta, čime se efektivno postiže ubrzanje izvršavanja algoritma. U ovom radu su međusobno zavisne operacije maksimalno pajplajnovane, dok se nezavisne operacije izvršavaju u istom taktu.
- **Računanje logaritma veličine niza (matrice)** – koristi se za određivanje koliko leptir operacija treba da se izvrši (pogledati sliku 4, vertikalne stupce označene sa „s=1“, „s=2“ i „s=3“). U najjednostavnijoj hardverskoj realizaciji bi se zahtevalo korišćenje memorije velike veličine zarad jednog registra koji se neće menjati tokom izvršavanja algoritma. Izabrano je da se logaritam računa u softveru i da se šalje u IP jezgro putem AXI *Lite* interfejsa.

4.3 Implementacija leptir modula (BUTTERFLY)

Kao što je već opisano, uloga leptir modula je da izvrši leptir operaciju:

$$F[k] = F_p[k] + W_{size}^{k} F_n[k] \tag{30}$$

Prethodna jednačina je predstavljena u kompleksnom obliku. Raščlanjivanjem na realni i imaginarni deo dobijemo sledeće formule:

$$topRE_o = topRE_i + botRE_i * wCOS[k] - botIM_i * wSIN[k] \tag{31}$$

$$topIM_o = topIM_i + botRE_i * wSIN[k] + botIM_i * wCOS[k] \tag{32}$$

$$botRE_o = topRE_i - botRE_i * wCOS[k] + botIM_i * wSIN[k] \tag{33}$$

$$botIM_o = topRE_i - botRE_i * wSIN[k] - botIM_i * wCOS[k] \tag{34}$$

Pri čemu važi:

$$wCOS[k] = cos\left(-2\pi \frac{k}{size}\right) \tag{35}$$

$$wSIN[k] = sin\left(-2\pi \frac{k}{size}\right) \tag{36}$$

Rotacioni faktori *wCOS* i *wSIN* su realizovani kao ROM memorija unutar leptir bloka, i potrebno ih je ukupno *N/2*, pri čemu *N* predstavlja broj elemenata niza, zbog osobine simetričnosti.

Radi povećavanja maksimalne frekvencije rada algoritma na FPGA ploči, pajplajnovano je izvršavanje leptir operacije, pa se vrednosti sabiranja i množenja izvršavaju u različitim taktovima (prvo množenja, pa suma proizvoda, i potom dodavanje vrednosti $topRE_i$ i $topIM_i$).

4.3.1 Definisanje interfejsa

- Ulazni interfejs

  topRE_i – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja gornji realni ulaz unutar leptir bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  topIM_i – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja gornji imaginarni ulaz unutar leptir bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  botRE_i – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja donji realni ulaz unutar leptir bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  botIM_i – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja gornji realni ulaz unutar leptir bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  k – tipa STD_LOGIC_VECTOR (log2c(FFT_SIZE/2)-1 downto 0) – označava trenutnu vrednost rotacionog faktora $W_{2^{size}}^{k}$.

  size – tipa STD_LOGIC_VECTOR (log2c(log2c(FFT_SIZE/2)-1 downto 0)) – predstavlja logaritam veličine niza za koji se trenutno računa Furijeova transformacija. Zajedno sa veličinom k u potpunosti opisuje trenutnu vrednost rotacionog faktora $W_{2^{size}}^{k}$.

- Izlazni interfejs

  topRE_o – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja gornji realni izlaz unutar leptir bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  topIM_o – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja gornji imaginarni izlaz unutar leptir bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  botRE_o – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja donji realni izlaz unutar leptir bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  botIM_o – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja gornji realni izlaz unutar leptir bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

- Komandni interfejs

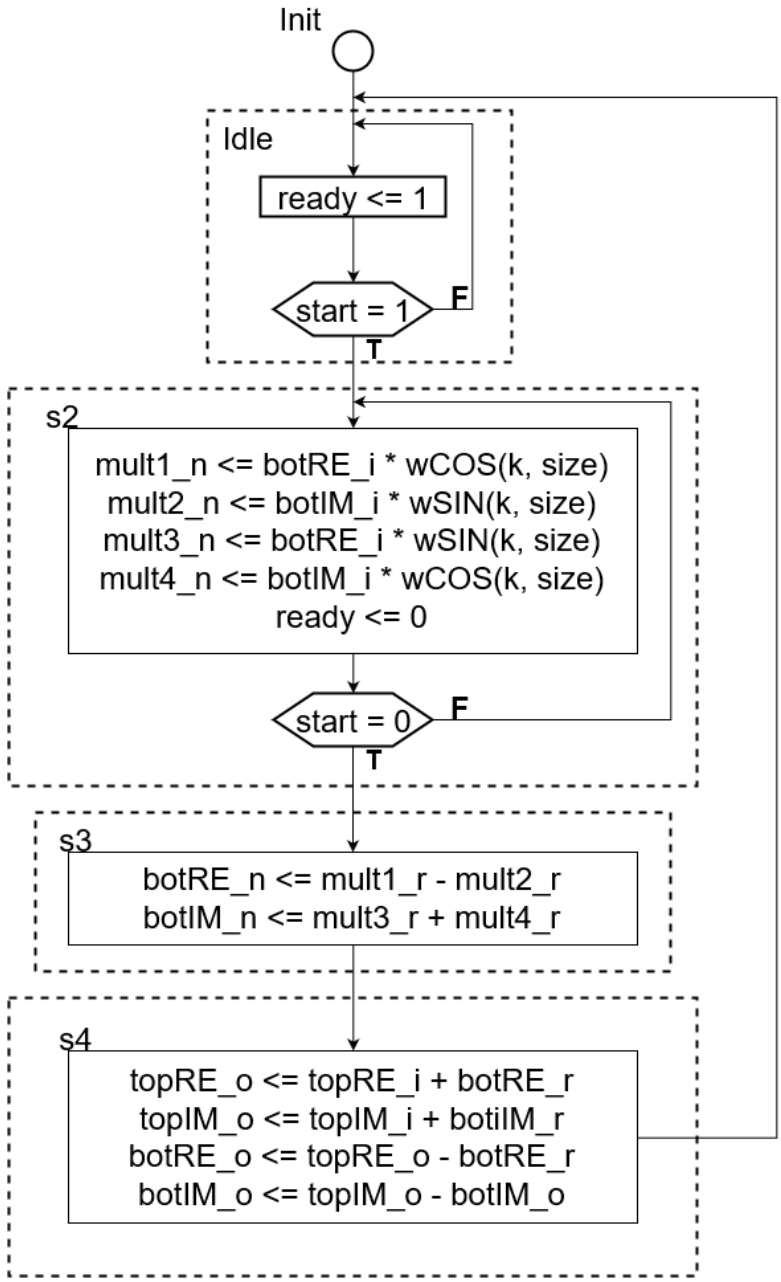
  start – tipa STD_LOGIC – kontroliše početak rada leptir bloka

- Statusni interfejs

  ready – tipa STD_LOGIC - ukazuje da li je leptir blok trenutno aktivan

4.3.2 Projektovanje upravljačkog modula

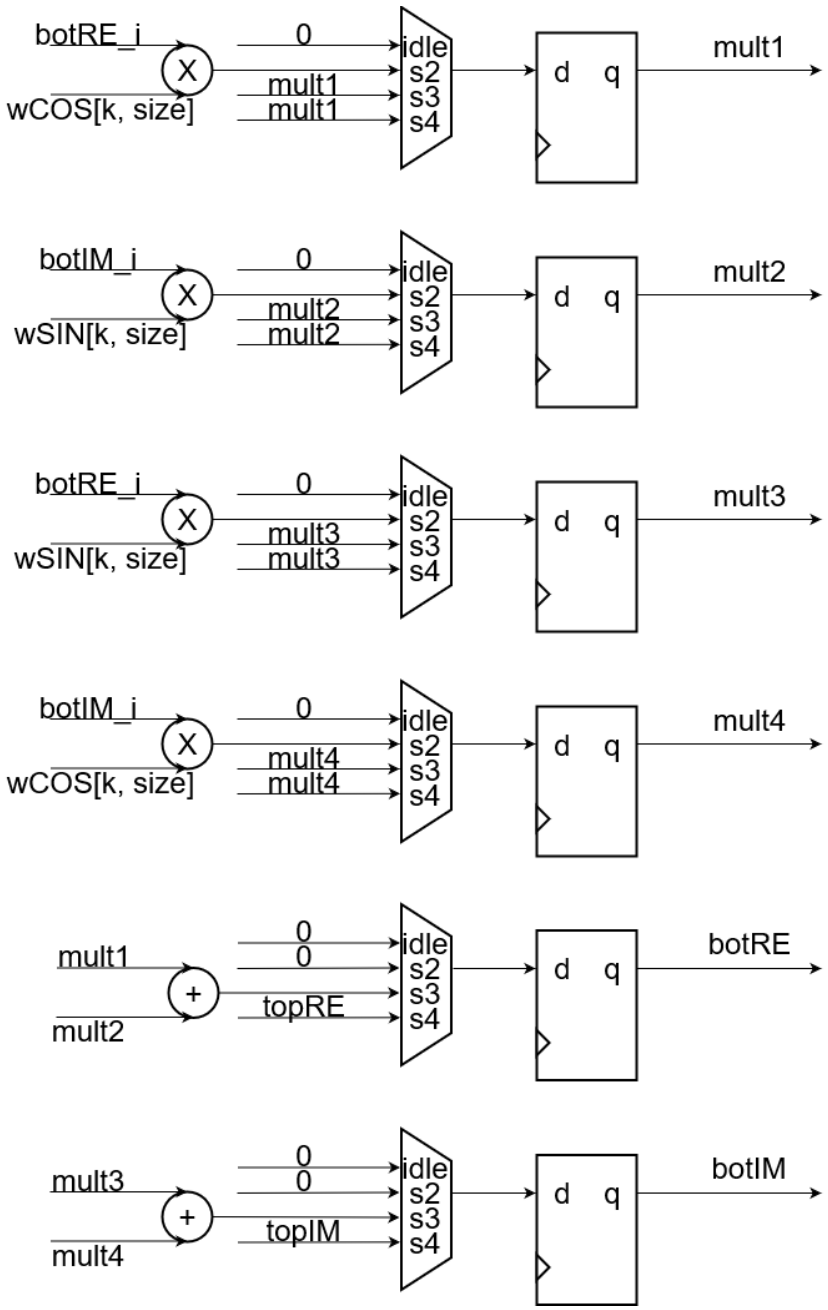
ASM dijagram ima četiri stanja. U prvom (*idle*) stanju se čeka na instrukciju za početak obrade ulaznih signala (*start*), signal *ready* je na visokom logičkom nivou. U drugom (*s2*) stanju se vrši množenje rotacionih faktora sa donjim ulaznim signalima leptir bloka, i spušta se *ready* signal na nisku logičku vrednost. U trećem (*s3*) se stanju vrši sabiranje pomnoženih signala. U četvrtom (*s4*) stanju se sabiraju vredosti gornjih ulaznih signala sa rezultujućim donjim signalima. Konačno, ponovo se ulazi u početno (*idle*) stanje i podiže se *ready* signal na visok logički nivo.



Slika 10 ASM dijagram leptir bloka

4.3.3 Projektovanje modula za obradu podataka

U datom sistemu postoji šest registara – četiri što skladište vrednosti množenja, i dva što skladište vrednosti zbira proizvoda. Množači se aktiviraju u drugom stanju, nakon što se detektuje da je aktivan start signal. Nakon što se izvrši množenje, odnosno u trećem stanju, vrši se sabiranje proizvoda i njegovo memorisanje u registre *botRE* i *botIM*. U četvrtom stanju se vrši sabiranje vrednosti ulaznih signala *topRE_i* i *topIM_i* sa registrima *botRE* i *botIM* i prosleđuje na izlaz. Dakle, ovaj modul je realizovan kao Murov automat i zahteva se konstantna vrednost ulaznih signala tokom trajanja izvršavanja datog modula.



Slika 11 Modul za obradu podataka unutar leptir bloka

4.4 Implementacija bloka za brzu Furijeovu transformaciju u jednoj dimenziji (FFT)

4.4.1 Definisanje interfejsa

- Ulazni interfejs

  data_i_addr_o – tipa STD_LOGIC_VECTOR (ld(FFT_SIZE)-1 downto 0) - predstavlja adresu člana niza dužine *size* koja se trenutno učitava.

  dataRE_i – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja realni ulaz unutar *fft* bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  dataIM_i – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja imaginarni ulaz unutar *fft* bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  data_rd_o – tipa STD_LOGIC – označava da li *fft* blok zahteva učitavanje narednog člana niza.

  data_rd_i – tipa STD_LOGIC – označava da li su vrednosti dataRE_i i dataIM_i stabilne i spremne da budu učitane u unutrašnju memoriju.

  log2s – tipa STD_LOGIC_VECTOR (ld(ld(FFT_SIZE))-1 downto 0), predstavlja celobrojnu vrednost logaritma veličine niza koji se obrađuje.

  size – tipa STD_LOGIC VECTOR (ld(FFT_SIZE)-1 downto 0), predstavlja veličinu niza koji se obrađuje.

- Izlazni interfejs

  data_o_addr_o – tipa STD_LOGIC_VECTOR (ld(FFT_SIZE)-1 downto 0) - predstavlja adresu člana niza dužine *size* koja se trenutno učitava.

  dataRE_o – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja realni izlaz iz *fft* bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  dataIM_o – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja imaginarni izlaz iz *fft* bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  data_wr_o – tipa STD_LOGIC – označava da li *fft* blok zahteva upis člana niza u spoljašnju memoriju i da li su vrednosti dataRE_o i dataIM_o stabilne

  data_wr_i – tipa STD_LOGIC – označava da li su vrednosti dataRE_i i dataIM_i učitane u spoljašnju matričnu memoriju.

- Komandni interfejs

  start – tipa STD_LOGIC – kontroliše početak rada *fft* bloka

- Statusni interfejs

  ready – tipa STD_LOGIC - ukazuje da li je *fft* blok trenutno aktivan

4.4.2 Projektovanje upravljačkog modula

Blok za jednodimenzionalnu Furijeovu transformaciju se sastoji iz dvadeset stanja, koji se mogu podeliti na dve celine: deo za permutovanje članova ulaznog niza korišćenjem algoritma za obrtanje redosleda bita (eng. *bit reversal*) i učitavanje niza – prvih šest stanja, kao i deo za rekurzivnu podelu niza na parne i neparne članove i izvršavanje leptir operacija nad članovima – preostalih trinaest stanja (*idle* je dvadeseto stanje).

U **neaktivnom (*idle*) stanju** se kontinualno u registre skladište ulazne vrednosti dužine i logaritma dužine niza, dok god je *start* signal na niskom nivou. Kad *start* signal postane aktivan, učitane vrednosti dužine niza ostaju nepromenjene do kraja izvršavanja Furijeove transformacije.

Algoritam za **obrtanje redosleda bita** (pomenut u poglavlju 2.5) funkcioniše na sledeći način. Pretpostavimo da je širina adresnog porta 12 bita, a da za trenutni niz koji se obrađuje se koristi 9 bita. U tom slučaju će se poslednjih sedam bita obrnuti, dok će prva tri ostati nepromenjena, odnosno biće na niskoj logičnoj vrednosti (na primer: *000*1 1100 1100 →*000*0 0110 0111). Podaci sa obrnute adrese se učitavaju u memoriju.

```
for (int jj = 0; jj < size; ++jj){
  int reversed = 0;
  int temp = jj;
  for (int kk = 0; kk < log2s-1; ++kk){
    reversed = reversed << 1;
    reversed = reversed | (temp&1);
    temp = temp >> 1;
  }
  dataRE[jj] = dataRE_i[reversed];
  dataIM[jj] = dataIM_i[reversed];
}
```

Listing 2 Algoritam za obrtanje redosleda bita

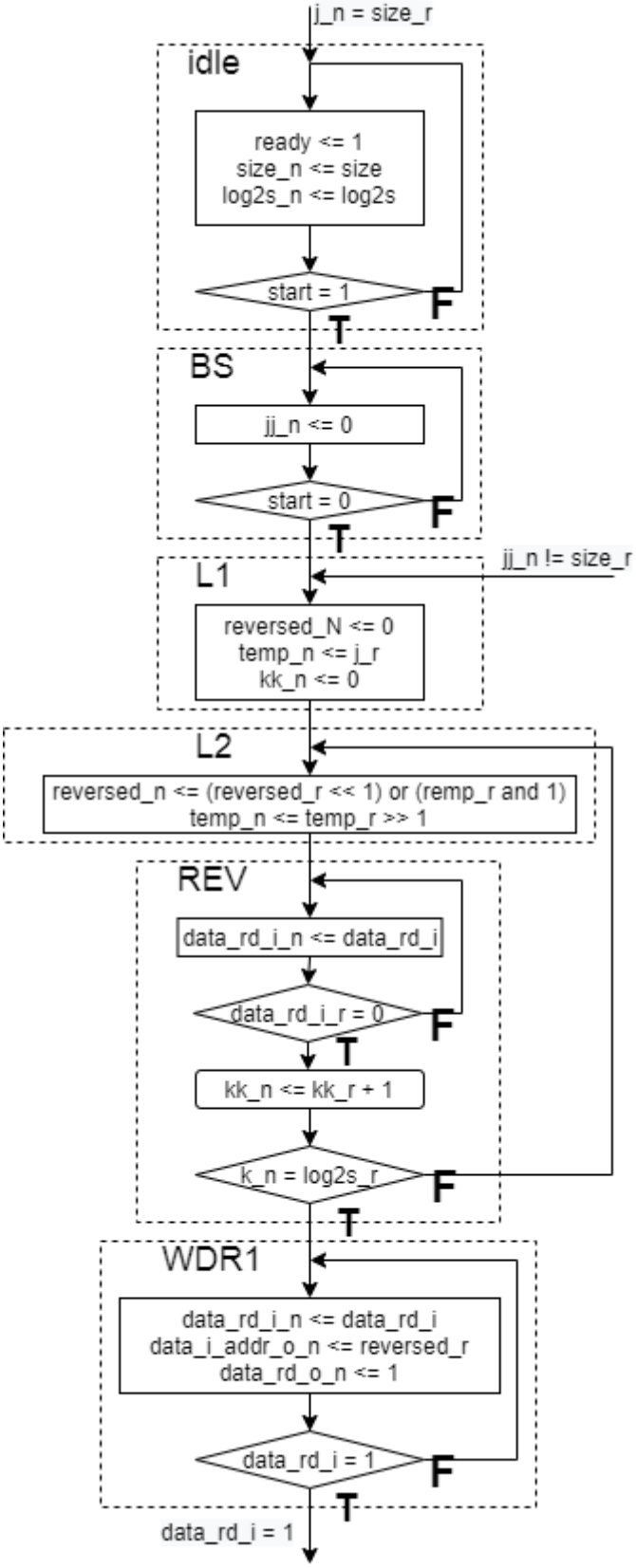
Upravljački modul algoritma za obrtanje redosleda bita se sastoji iz stanja BS (*bit reversal*), L1, L2, REV (*reverse*), WDR1 (*wait data read 1*) i RD (*read*). U stanju **BS** se inicijalizuje promenljiva *jj* na nulu, i čeka da se spusti *start* signal na nisku logičku vrednost. U stanju **L1** se inicijalizuju promenljive *reversed*, *temp* i *kk* (linije 2, 3 i 4 listinga), u stanju **L2** se ažuriraju vrednosti promenljivih *reversed* i *temp* (linije 5, 6 i 7 listinga), a u stanju **REV** se utvrđuje da li je cela adresa, odnosno promenljiva obrnula redosled bita. Ukoliko jeste, nastavlja se sa učitavanjem realnih i imaginarnih članova niza. U stanju **WDR1** se šalje zahtev za čitanje iz memorije sve dok se ne dobije potvrdan odgovor da su stigli podaci na ulazne portove. U stanju **RD** se upisuju vrednosti sa ulaza u unutrašnju memoriju (linije 9 i 10 listinga), i ukoliko su svi podaci upisani nastavlja se sa glavnim delom Furijeove transformacije, inače se vraća na početak (stanje BS) i inkrementuje vrednost registra *jj*.

Iterativni algoritam za **rekurzivnu podelu niza** na parne i neparne članove i **pozivanje leptir operacije** nad tekućim elementima je prikazan u listingu 3 i izvršava operacije kao na slici 4. Napisan je u iterativnom obliku, zato što RT metodologija ne omogućava mapiranje rekurzivnih funkcija na hardver.

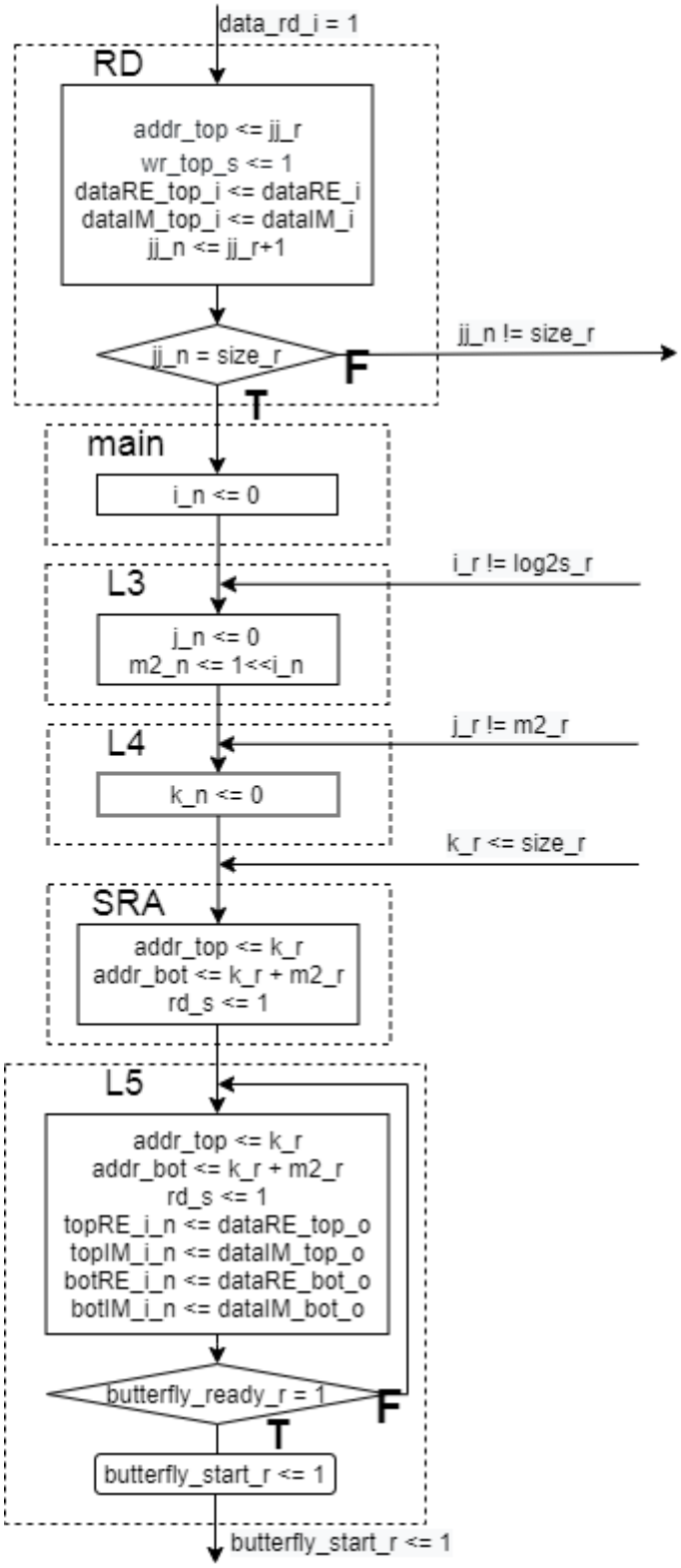
```
for (int i=0; i < log2size-1; ++i){
  int m2 = 1 << i;
  for (int j=0; j < m2; ++j){
    for (int k=j; k < size; k+=m2<<1){
      double topRE = dataRE[k];
      double topIM = dataIM[k];
      double bottomRE = dataRE[k+m2];
      double bottomIM = dataIM[k+m2];
      butterfly(
        topRE,
        topIM,
        bottomRE,
        bottomIM,
        dataRE+k,
        dataIM+k,
        dataRE+k+m2,
        dataIM+k+m2,
        j << (log2size-i-2)
      );
    }
  }
}
```

Listing 3 Iterativni algoritam za rekurzivnu podelu niza i pozivanje leptir operacije

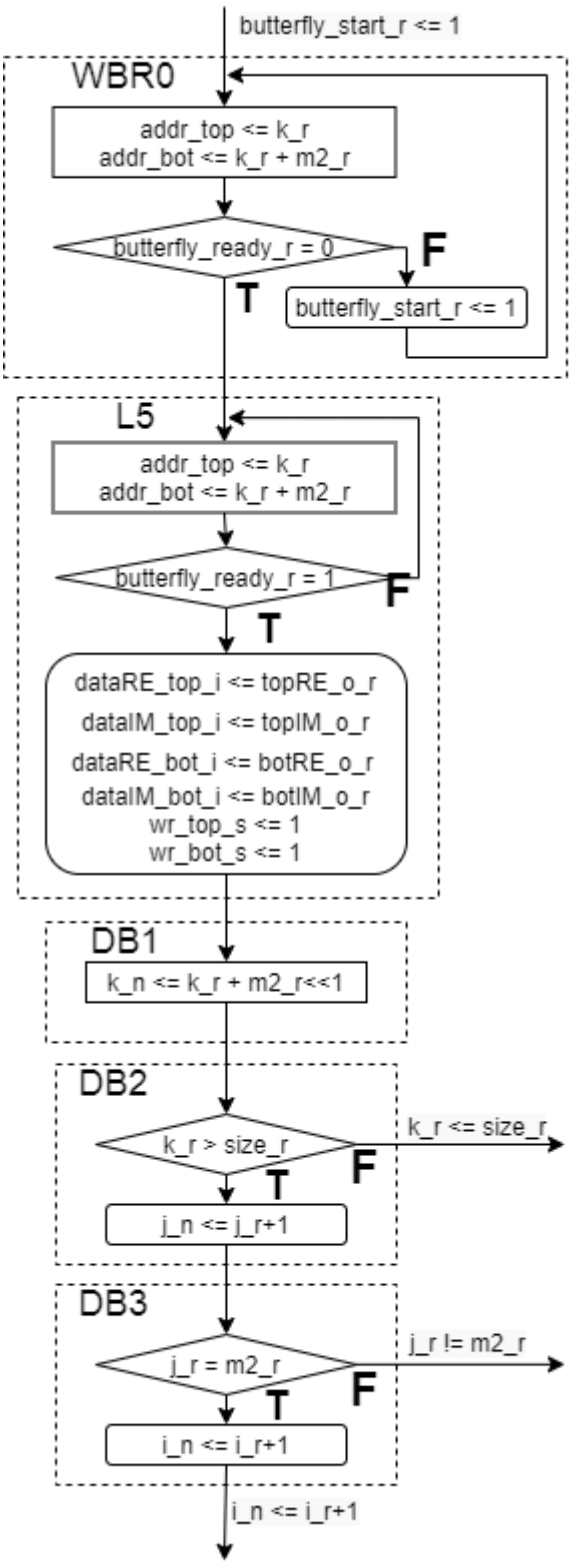
Upravljački modul algoritma za rekurzivnu podelu niza se sastoji iz stanja MAIN, L3, L4, SRA (*set read address*), L5, WBR0 (*wait butterfly ready 0*), WBR1 (*wait butterfly ready 1*), DB1, DB2, DB3, DB4 (*do butterfly*), WDW0 (*wait data write 0*) i WDW1 (*wait data write 1*). U stanju **MAIN** se inicijalizuje vrednost *i* registra. U stanju **L3** se inicijalizuju vrednosti *j* i *m2* registra. U stanju **L4** se inicijalizuje vrednost *k* registra. U stanju **SRA** se postavljaju adrese za učitavanje vrednosti iz memorije koji treba da se obrađuju u leptir operaciji, a potom se u stanju **L5** učitane vrednosti prosleđuju portovima leptir modula i drži se *start* signal na visokom logičkom nivou, sve dok leptir modul ne potvrdi da je primio naredbu. U stanju **WBR0** se čeka potvrda od leptir bloka da je primio *start* signal, a u **WBR1** se čeka potvrda da je leptir blok završio obradu podataka koji se potom upisuju u unutrašnju memoriju. U stanjima **DB1**, **DB2**, **DB3** i **DB4** se utvrđuje da li su registri *i*, *j* ili *k* došli do maksimalnih dozvoljenih vrednosti unutar petlji. Ukoliko je registar *i* došao do maksimalne vrednosti, podaci se upisuju u matricu pomoću stanja **WDW0** i **WDW1**.



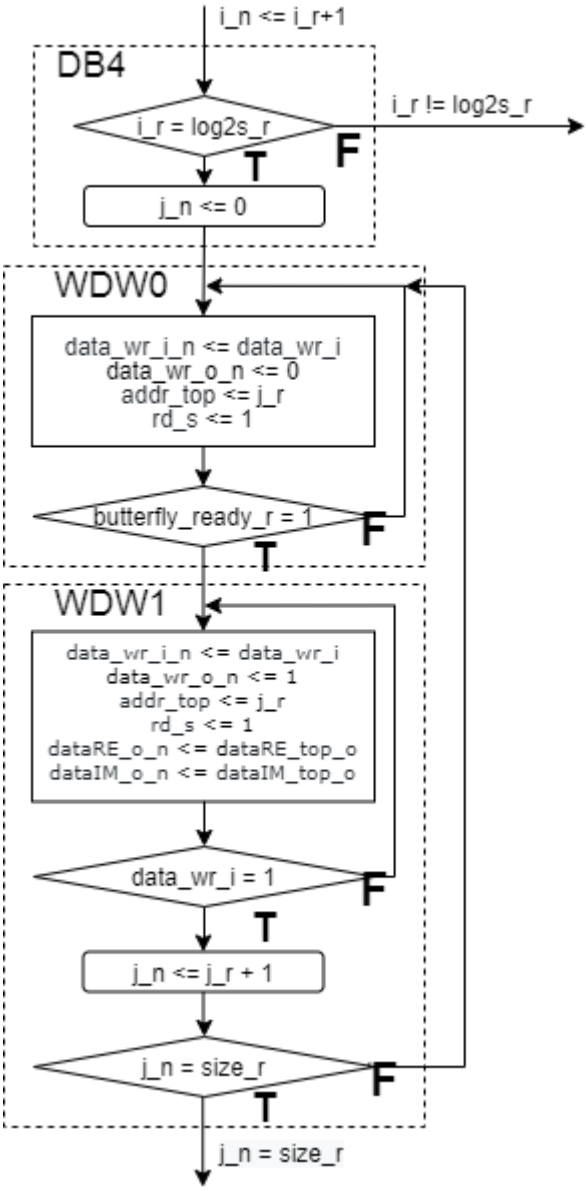
Slika 12 ASM dijagram *fft* bloka (prvi deo)



Slika 13 ASM dijagram *fft* bloka (drugi deo)



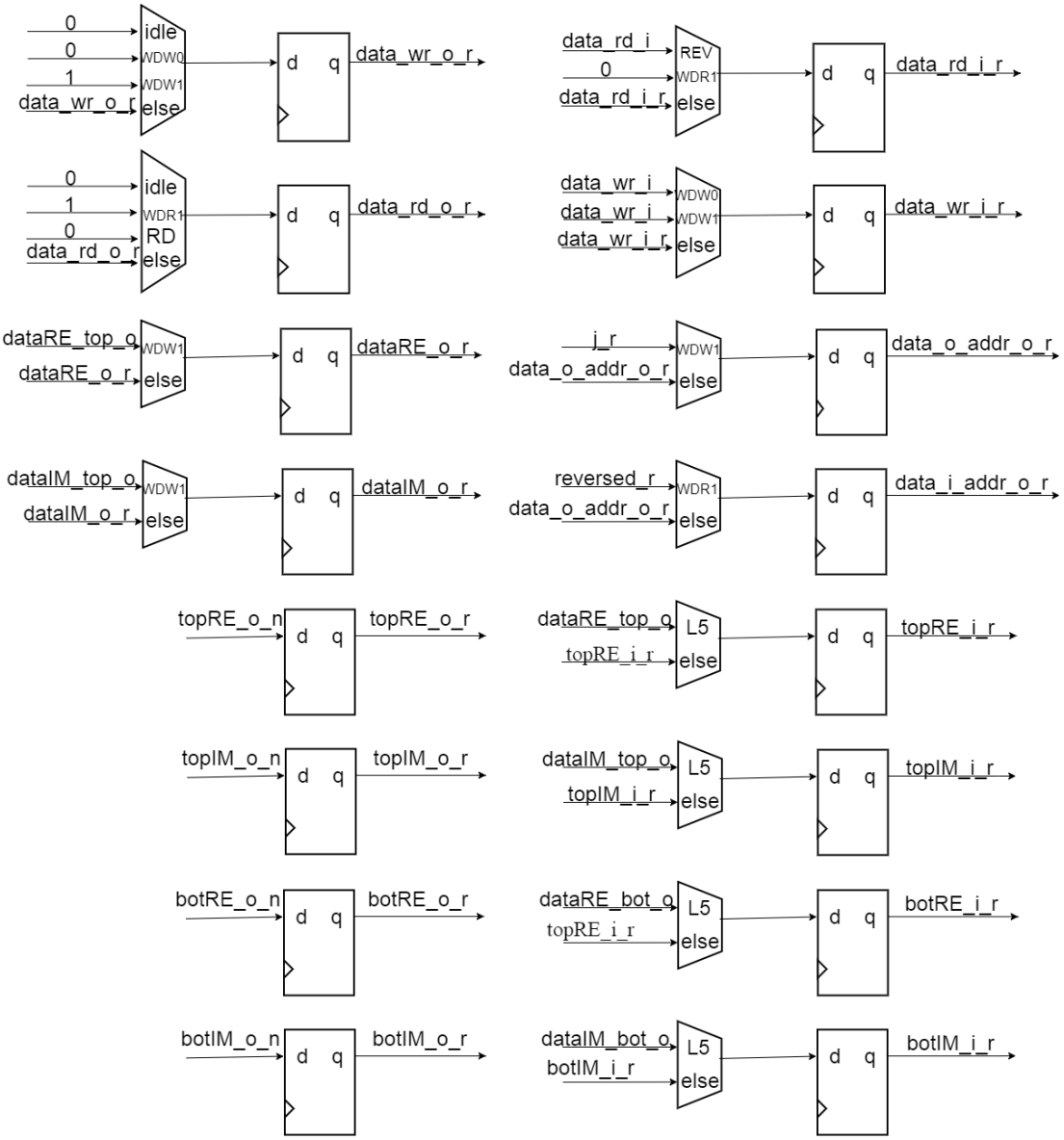
Slika 14 ASM dijagram *fft* bloka (treći deo)



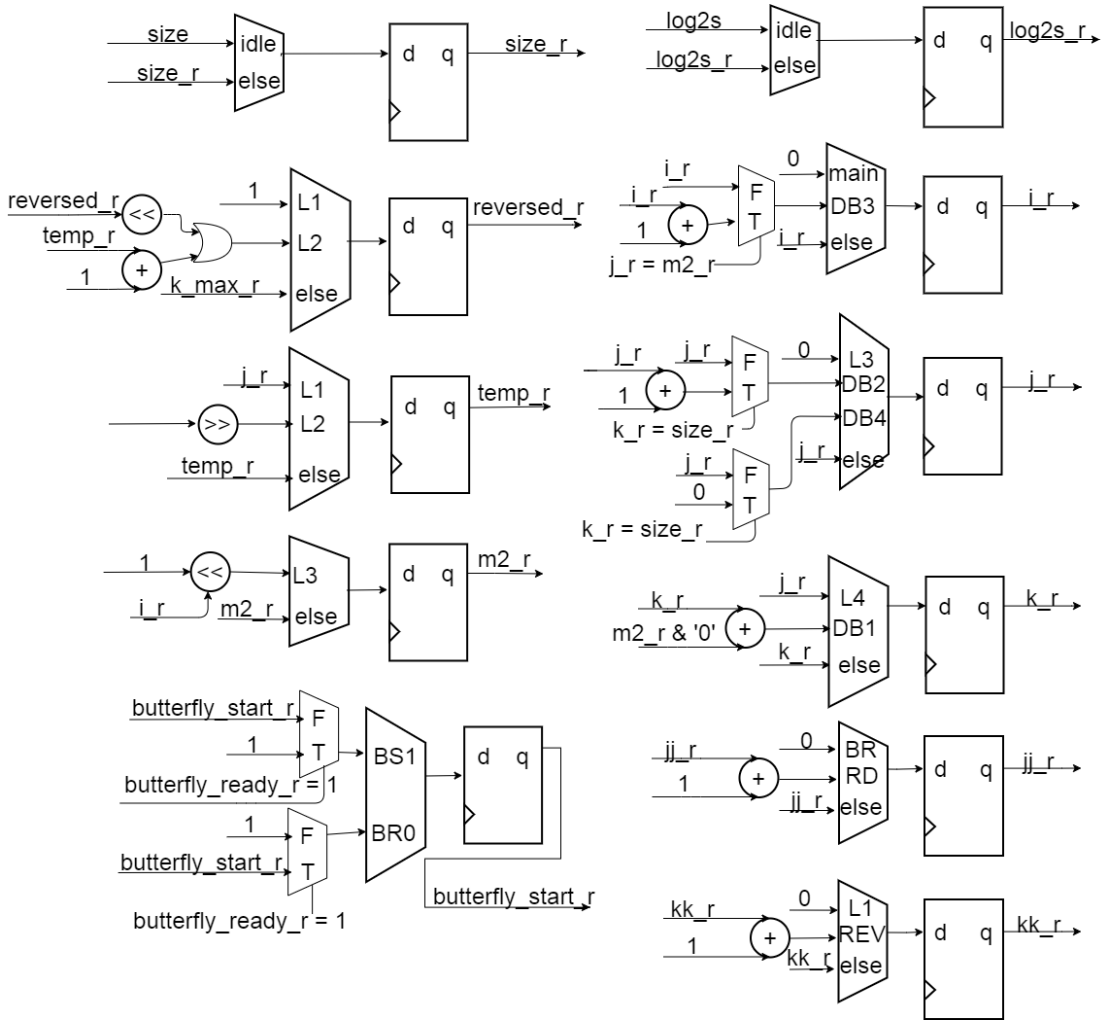
Slika 15 ASM dijagram *fft* bloka (četvrti deo)

4.4.3 Projektovanje modula za obradu podataka

U *fft* bloku postoji dvadeset tri registra, ne računajući unutrašnju memoriju u kojoj se skladište sve vrednosti imaginarnog i realnog niza koji se obrađuje. Signali *dataRE_o_r*, *dataIM_o_r*, *data_o_addr_r*, *data_i_addr_r*, *data_wr_i* i *data_rd_o* su diretkno povezani sa odgovarajućim izlaznim portovima. Veličine registara direktno zavise od širine ulaznih signala u registre, i veličine mogu biti DATA_WIDTH, ld(FFT_SIZE), ili ld(ld(FFT_SIZE)).



Slika 16 Izgled modula za obradu podataka *fft* bloka (prvi deo)



Slika 17 Izgled modula za obradu podataka *fft* bloka (drugi deo)

4.5 Implementacija bloka za brzu Furijeovu transformaciju u dve dimenzije (FFT2)

4.5.1 Definisanje interfejsa

- Ulazni interfejs

  Data_i_addr_o – tipa STD_LOGIC_VECTOR (ld(FFT_SIZE)-1 downto 0) - predstavlja adresu člana niza dužine *size* koja se trenutno učitava.

  dataRE_i – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja realni ulaz unutar *fft2* bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

dataIM_i – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja imaginarni ulaz unutar *fft2* bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

data_rd_o – tipa STD_LOGIC – označava da li *fft2* blok zahteva učitavanje narednog člana niza.

data_rd_i – tipa STD_LOGIC – označava da li su vrednosti dataRE_i i dataIM_i stabilne i spremne da budu učitane u spoljašnju memoriju matrice.

log2w – tipa STD_LOGIC_VECTOR (ld(ld(FFT_SIZE))-1 downto 0), predstavlja celobrojnu vrednost logaritma širine matrice koja se obrađuje zaokruženu na gore.

log2h – tipa STD_LOGIC_VECTOR (ld(ld(FFT_SIZE))-1 downto 0), predstavlja celobrojnu vrednost logaritma visine matrice koja se obrađuje zaokruženu na gore.

width – tipa STD_LOGIC VECTOR (ld(FFT_SIZE)-1 downto 0), predstavlja širinu matrice koji se obrađuje.

height – tipa STD_LOGIC VECTOR (ld(FFT_SIZE)-1 downto 0), predstavlja visinu matrice koji se obrađuje.

- Izlazni interfejs

  data_o_addr_o – tipa STD_LOGIC_VECTOR (ld(FFT_SIZE)-1 downto 0) - predstavlja adresu člana niza dužine *size* koja se trenutno učitava.

  dataRE_o – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja realni izlaz iz *fft2* bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  dataIM_o – tipa STD_LOGIC_VECTOR (WIDTH-1 downto 0) – predstavlja imaginarni izlaz iz *fft2* bloka. Vrednost je predstavljena sa fiksnom tačkom širine FIXED_POINT_WIDTH.

  data_wr_o – tipa STD_LOGIC – označava da li *fft2* blok zahteva upis člana niza u spoljašnju memoriju i da li su vrednosti dataRE_o i dataIM_o stabilne na izlazu.

  data_wr_i – tipa STD_LOGIC – označava da li su vrednosti dataRE_i i dataIM_i učitane u spoljašnju memoriju matrice.

- Komandni interfejs

  start – tipa STD_LOGIC – kontroliše početak rada *fft2* bloka

- Statusni interfejs

  ready – tipa STD_LOGIC - ukazuje da li je *fft2* blok trenutno aktivan

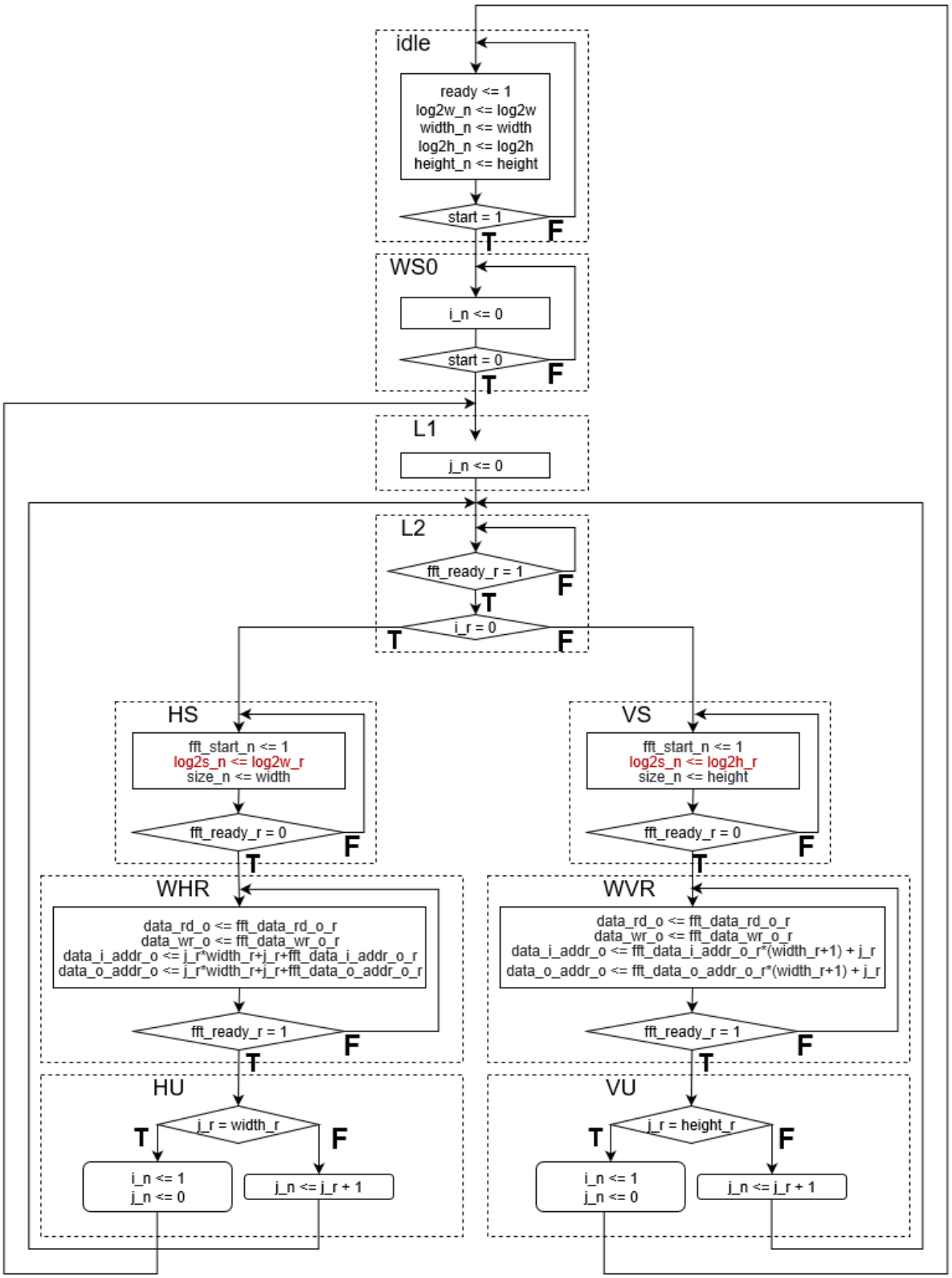
4.5.2 Projektovanje upravljačkog modula

Blok za dvodimenzionalnu Furijeovu transformaciju ima ulogu da bude most između memorija u kojima su uskladištene imaginarna i realna matrica, i bloka za jednodimenzionalnu Furijeovu transformaciju. Drugim rečima, *fft2* blok mora da pamti i vodi računa koji se red, odnosno kolona u matricama trenutno obrađuje, kao i da omogući komunikaciju između memorija i *fft* bloka.

Upravljački modul se sastoji iz deset stanja – IDLE, WSO (*wait start 0*), L1, L2, HS (*horizontal start*), WHR (*wait horizontal ready*), HU (*horizontal update*), VS (*vertical start*), WVR (*wait vertical ready*), VU (*vertical ready*).

U **neaktivnom (*idle*) stanju**, slično kao i u slučaju sa jednodimenzionalnom Furijeovom transformacijom, kontinualno se u registre skladište ulazne vrednosti visine i širine matrica, i njihovih logaritama, dok god je *start* signal na niskom logičkom nivou. Kad *start* signal postane aktivan, učitane vrednosti dužine niza ostaju nepromenjene do kraja izvršavanja dvodimenzionalne Furijeove transformacije. U stanju **WS0** se čeka na spuštanje *start* signala, da bi se započelo obrađivanje podataka (u skladu sa četvorofaznim hendšejk protokolom), i podešava se signal registar *i_r* na nisku logičku vrednost, što označava da se trenutno obrađuju redovi matrica (*i_r* ukazuje da li se kolone ili redovi obrađuju).

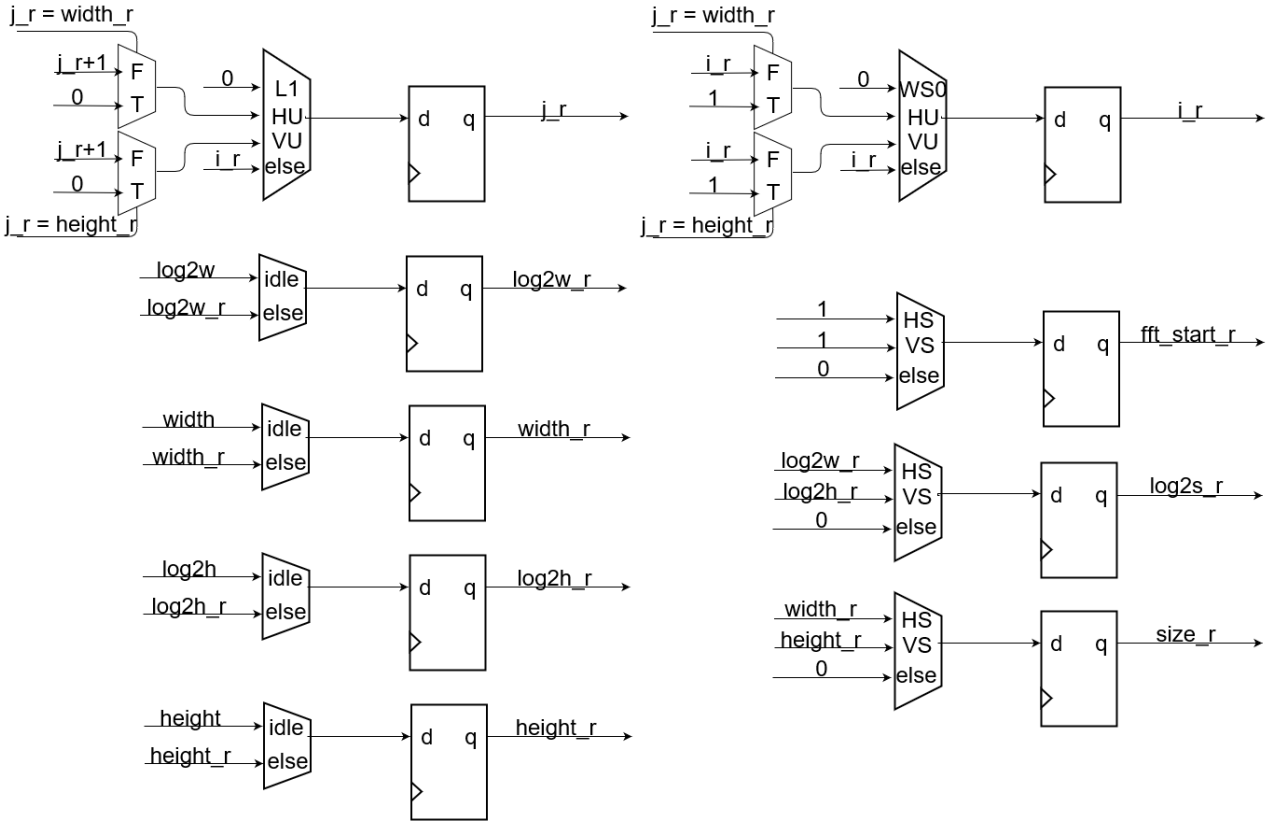
U stanju **L1** se inicijalizuje vrednost *j* registra, koji označava broj reda ili kolone koji se trenutno obrađuje u *fft* bloku. U stanju **L2** se proverava da li je *fft* blok spreman da obrađuje sledeći niz. Ukoliko jeste, podaci o veličini i start signal se šalju u **HS** (**VS**) stanju. U stanju **WHR** (**WVR**) se vrši transformacija adrese sa jednodimenzionalnog niza na matricu, kao i obostrano prosleđivanje podataka. U stanja **HU** i **VU** se ulazi tek nakon što je obrađen čitav niz, i tu se proverava da li su svi nizovi (horizontalni ili veritikalni) obrađeni. Ukoliko jesu, nakon HU stanja se podiže signal *i_r* na logički visok nivo, kao indikator da treba da se krene sa obradom vertikalnih nizova. Nakon VU stanja se završava za obradom matrice i vraća u *idle* stanje.



Slika 18 ASM dijagram *fft2* modula

4.5.3 Projektovanje modula za obradu podataka

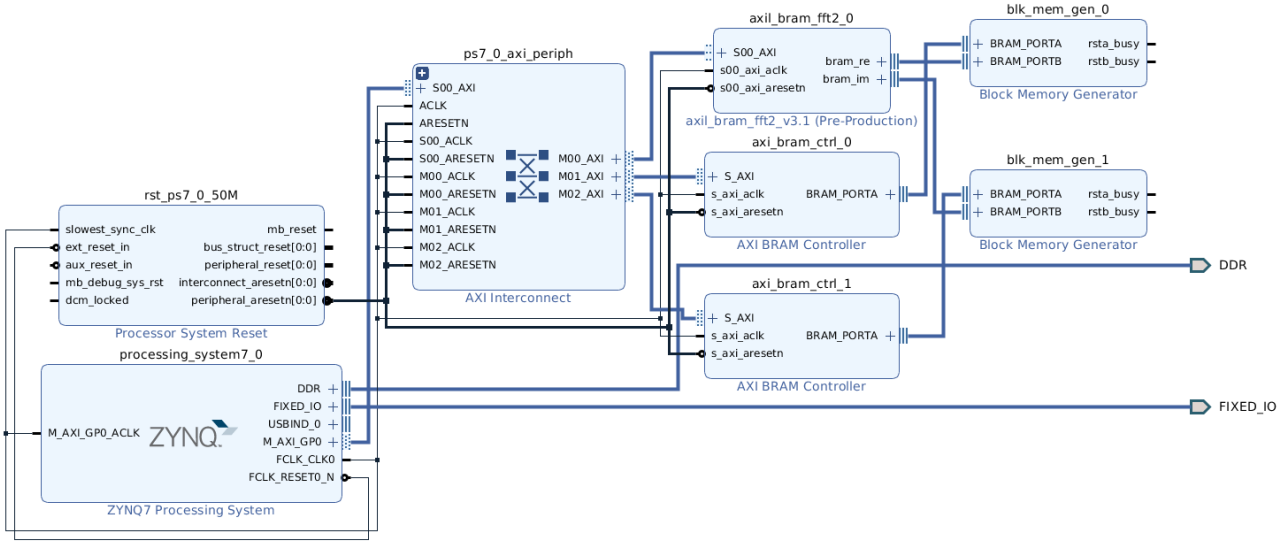
Modul za obradu podataka *fft2* bloka se sastoji iz devet registara. Četiri registra skladište ulazne podatke, odnosno širinu i visinu matrica, kao i njihove logaritme. Veličine tih registara su ld(FFT_SIZE * FFT_SIZE) i ld(ld(FFT_SIZE * FFT_SIZE)). Dva registra služe da skladište veličinu niza koji će biti obrađivan u bloku *fft*, kao i njegov logaritam, i veličina tih registara je ld(FFT_SIZE) i ld(ld(FFT_SIZE)). Indikator *i_r* je jednobitni registar, dok je iterator *j_r* veličine ld(FFT_SIZE).



Slika 19 Modul za obradu podataka *fft2* bloka

4.6 Integracija u *Zynq* platformu

Nakon završetka projektovanja IP jezgra, mogu se povezati sve komponente u jedinstven sistem, prikazano na slici 20, i pristupiti generisanju binarnog koda na osnovu kojeg će se programirati FPGA. Sistem se sastoji iz *Zynq7* procesora , dva *true dual-port* BRAM-a povezana preko dvopristupnog BRAM interkonekta sa procesorom, i IP jezgra za dvodimenzionalnu Furijeovu transformaciju. IP jezgro je direktno povezano sa memorijom i *Zynq7* procesorom pomoću BRAM i AXI *Lite* interfejsa redom, pri čemu IP jezgro koristi port B od memorija, dok BRAM interkonekt koristi port A. BRAM interkonekt je takođe povezan sa procesorom preko AXI *Lite* interfejsa.



Slika 20 Blok dijagram celokupnog sistema

4.7 Potrošnja hardverskih resursa i frekvencija rada

Nakon obavljanja vremenske (eng. *timing*) analize, dobijamo sledeće rezultate:

Setup time (Worst Slack):	6.143ns
Hold time (Worst Slack):	0.026ns
Pulse Width (Worst Slack):	9.020ns

Tabela 4 Vremenska analiza

Drugim rečima, kad se saberu prethodne vrednosti vremena, minimalno trajanje jedne periode klok signala iznosi 15.189ns, odnosno maksimalna frekvencija rada ploče iznosi 65.837MHz.

Potrošeni hardverski resursi su prikazani u tabelama 5, 6 i 7. Na njima se može primetiti da je najveća iskorišćenost LUT-ova (preko trećine), dok je najmanja iskorišćenost blok memorije (6.67%). Na slici 21 je prikazan prostorni raspored iskorišćenih resursa.

Site Type	Used	Fixed	Available	Util%
Slice LUTs	6047	0	17600	34.36
LUT as Logic	5992	0	17600	34.05
LUT as Memory	55	0	6000	0.92
LUT as Distributed RAM	0	0		
LUT as Shift Register	55	0		
Slice Registers	4683	0	35200	13.30
Register as Flip Flop	4683	0	35200	13.30
Register as Latch	0	0	35200	0.00
F7 Muxes	330	0	8800	3.75
F8 Muxes	0	0	4400	0.00

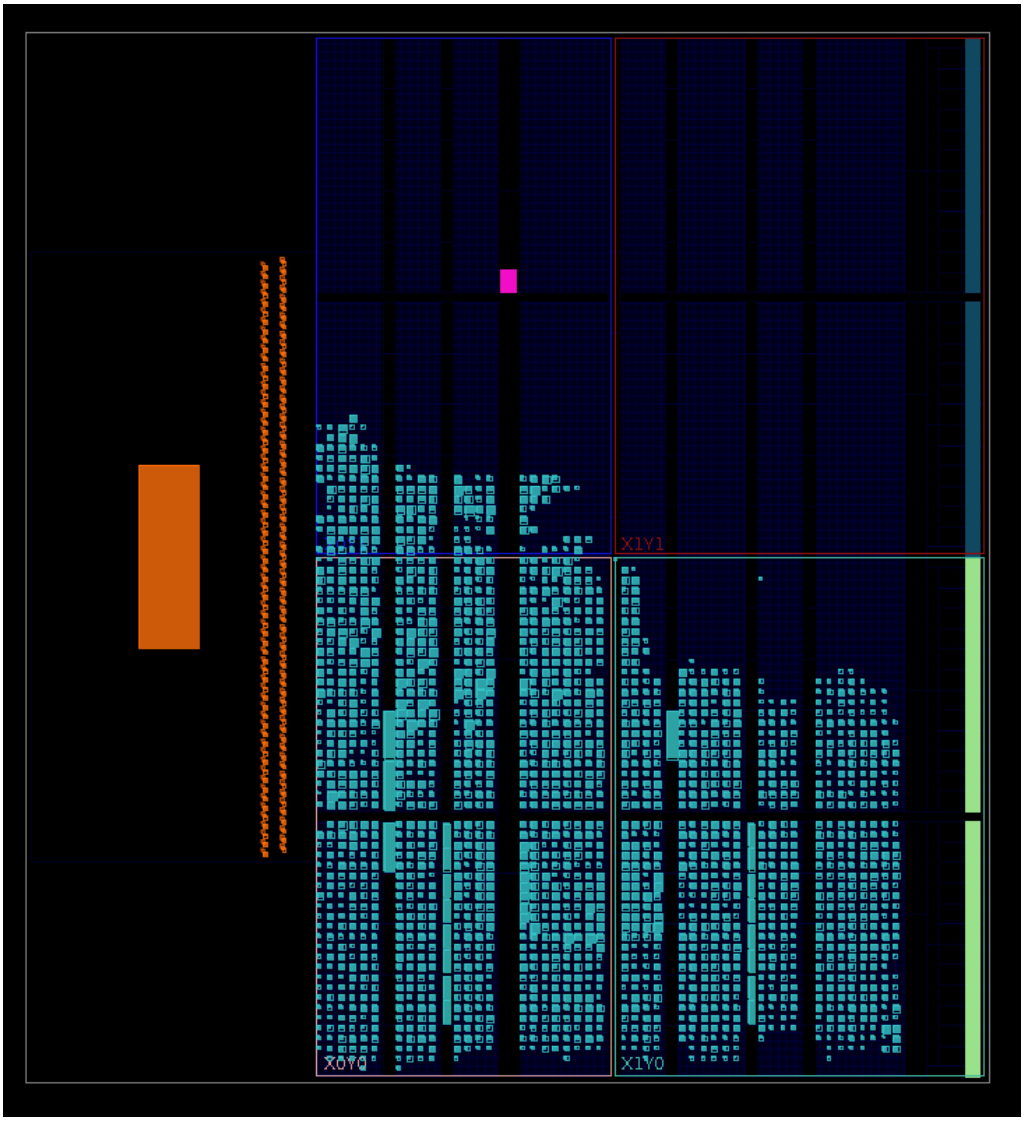
Tabela 5 Iskorišćenost logičkih kola

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	4	0	60	6.67
RAMB36/FIFO*	4	0	60	6.67
RAMB36E1 only	4			
RAMB18	0	0	120	0.00

Tabela 6 Iskorišćenost blok memorije

Site Type	Used	Fixed	Available	Util%
DSPs	16	0	80	20.00
DSP48E1 only	16			

Tabela 7 Iskorišćenost DSP-jeva



Slika 21 Prostorna raspodela iskorišćenih resursa

Glava 5

Funkcionalna verifikacija projektovanog IP bloka

5.1 Uvod

Savremena IP jezgra predstavljaju veoma složene uređaje, i sastoje se iz ogromnog broja komponenti, u koje se ubrajaju registri, memorijski blokovi, i kombinaciona logika. Da bi se u potpunosti utvrdio ispravan rad takvih IP jezgara, neophodno je proveriti da li će sva moguća unutrašnja stanja i ulazni podaci uzrokovati generisanje ispravnog rešenja. Usled ograničene moći računara, takve provere su najčešće neisplative. Neophodno je uskladiti proces verifikacije sa finansijskim i vremenskim zahtevima, da bi se dobio proizvod traženog kvaliteta. Kao posledica, većina savremenih digitalnih uređaja poseduje greške koje nije lako pronaći.

Funkcionalna verifikacija je proces koji ima za cilj utvrđivanje da li će dizajn raditi u skladu sa zadatim funkcionalnim specifikacijama, pre nego što zamišljeni uređaj bude fabrikovan. Cena otklanjanja greške je višestruko manja ukoliko je greška otkrivena tokom verifikacije, nego prilikom testiranja proizvedenih uređaja. Proces funkcionalne verifikacije počinje od napisane funkcionalne specifikacije dizajna, i sastoji se iz šest koraka koji se zajedno zovu verifikacioni ciklus:

1. **Kreiranje verifikacionog plana** – on opisuje šta i na koji način treba da bude verifikovano. Verifikacioni plan se sastoji iz više poglavlja koja detaljno odgovaraju na pitanja kao što su: kakvi će se testovi i metode verifikacije koristiti, koji su softverski i hardverski resursi neophodni, koliko je verifikacionih inžinjera potrebno, kako raspodeliti njihov posao tokom vremena, koje funkcionalnosti treba da budu verifikovane, koji su uslovi završetka verifikacije, i tako dalje [10].
2. **Razvoj verifikacionog okruženja** – predstavlja softverski kod koji emulira neposredno okruženje dizajna, i koje ima mogućnost provere ispravnosti rezultata. Razni alati olakšavaju proveru ispravnosti rada dizajna, sa već implementiranim funkcionalnostima kao što je analiza strukturne pokrivenosti. Verifikaciono okruženje se može razviti tako da omogućava različite metode verifikacije hardvera, u koje se ubrajaju okruženja zasnovana na zlatnim vektorima, referentnim modelima, deterministički, formalni, i drugi.
3. **Debagovanje dizajna i verifikacionog okruženja** – u toku izvršavanja testova se pronalaze razlike između očekivanih rezultata verifikacionog okruženja i stvarnih rezultata dizajna, sa ciljem da se utvrde i uklone uzroci tih razlika.
4. **Izvršavanje regresionih testova** – predstavlja ponovno izvršavanje testova nakon ispravljanja otkrivenih grešaka, zbog mogućeg nastanka novih problema.

5. **Debagovanje proizvedenog hardvera** – nakon fabrikacije hardvera, proces debagovanja je mnogo učinkovitiji zbog razlike u brzini između pravog hardvera i njegovog simulatora. Moguće je pronaći greške u ovom koraku, međutim primarni cilj je otkrivanje greški pre fabrikacije s obzirom na veliku cenu otklanjanja takvih greški.
6. **Izvršavanje eskejp analize** (eng. *escape analysis*) – ukoliko su uočene greške nakon fabrikacije, neophodno je utvrditi zbog čega nisu detektovane tokom simulacije, sa ciljem da bi se takvi propusti otklonili u budućim projektima.

Slika 22 Verifikacioni ciklus (preuzeto iz [11])

U ovom projektu nije posebno opisan korak debagovanja dizajna i verifikacionog okruženja zbog male složenosti projekta. Takođe, nisu rađeni koraci debagovanja proizvedenog hardvera i izvršavanja eskejp analize jer nisu uključeni u projektne zahteve.

5.2 Kreiranje verifikacionog plana

5.2.1 Opis rada dizajna

U ovom projektu je verifikovana funkcionalnost modula za dvodimenzionalnu Furijeovu transformaciju bez enkapsuliranih interfejsa za AXI komunikaciju. Interfejs verifikovanog modula je prikazan u listingu 4. Vrednosti ulaznih parametara *width* i *height* moraju biti stepeni dvojke. Parametri *log2w* i *log2h* su zavisini i predstavljaju logaritme *width* i *height*. Pokretanje uređaja počinje sa četvorofaznim hendšejk protokolom između signala *start* i *ready* (pogledati poglavlje 4.1.3), dok su prethodno pomenuta četiri parametra aktivna i ne menjaju vrednost. Sve vrednosti datih parametara treba da budu umanjene za jedan – na primer, ako je matrica veličine 4x8, treba da budu prosleđene vrednosti 3 i 7, dok prosleđene vrednosti logaritama treba da budu 1 ($\log_2 4=2$) i 2($\log_2 8=3$).

Pomoću prvih pet signala u listingu 4 (od *data_i_addr_o* do *data_rd_i*) se vrši upis vrednosti u unutrašnju memoriju dizajna iz spoljašnje memorije (BRAM-ova), dok se sa preostalih pet signala (od *data_o_addr_o* do *data_wr_i*) vrši upisivanje u spoljašnju memoriju.

```
data_i_addr_o : out STD_LOGIC_VECTOR (log2c(FFT_SIZE*FFT_SIZE)-1 downto 0);
dataRE_i : in STD_LOGIC_VECTOR (DATA_WIDTH-1 downto 0);
dataIM_i : in STD_LOGIC_VECTOR (DATA_WIDTH-1 downto 0);
data_rd_o : out STD_LOGIC;
data_rd_i : in STD_LOGIC;

data_o_addr_o : out STD_LOGIC_VECTOR (log2c(FFT_SIZE*FFT_SIZE)-1 downto 0);
dataRE_o : out STD_LOGIC_VECTOR (DATA_WIDTH-1 downto 0);
dataIM_o : out STD_LOGIC_VECTOR (DATA_WIDTH-1 downto 0);
data_wr_o : out STD_LOGIC;
data_wr_i : in STD_LOGIC;

log2w : in STD_LOGIC_VECTOR (log2c(log2c(FFT_SIZE))-1 downto 0);
width : in STD_LOGIC_VECTOR (log2c(FFT_SIZE)-1 downto 0);

log2h : in STD_LOGIC_VECTOR (log2c(log2c(FFT_SIZE))-1 downto 0);
height : in STD_LOGIC_VECTOR (log2c(FFT_SIZE)-1 downto 0);

start : in STD_LOGIC;
ready : out STD_LOGIC);
```

Listing 4 Inferfejs DUT-a

DUT (eng. *Design Under Test*) obavlja jednodimenzionalnu Furijeovu transformaciju prvo nad svim vrstama matrice, a potom kolonama. Redosled slanja podataka iz matrice će biti opisan u sledećem primeru matrice veličine 4x4, bez gubljenja opštosti za matrice proizvoljne veličine:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

Verifikaciono okruženje mora prvo da pošalje prvi red (1, 2, 3, 4), pa nakon što primi Furijeovu transformaciju datog reda (na primer 21, 22, 23, 24 - pogledati matricu ispod), šalje drugi red (5, 6, 7, 8), i tako dalje do poslednjeg reda (13, 14, 15, 16). Dobija se sledeća matrica koja mora da se obradi po kolonama:

$$\begin{bmatrix} 21 & 22 & 23 & 24 \\ 25 & 26 & 27 & 28 \\ 29 & 30 & 31 & 32 \\ 33 & 34 & 35 & 36 \end{bmatrix}$$

Na isti način, sad se šalju vrednosti (21, 25, 29, 33), pa (22, 26, 30, 34), sve do poslednje kolone. Vrednosti koje se na kraju dobiju od IP jezgra, predstavljaju dvodimenzionalnu Furijeovu transformaciju matrice.

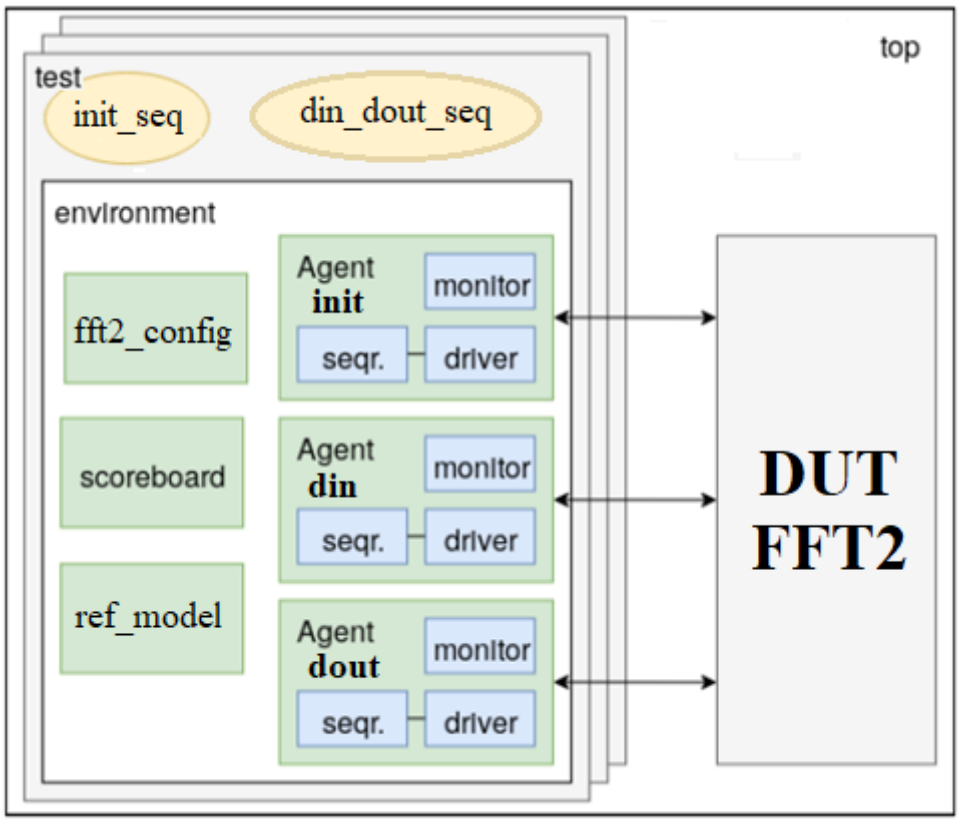
5.2.2 Opis nivoa verifikacije

Projektovani *fft2* modul je jednostavan i zahteva verifikaciju samo na najvišem nivou hijerarhije. Dodatno, intermedijerni rezultati automatski predstavljaju rezultate modula na nižim nivoima hijerarhije (jednodimenzionalne Furijeove transformacije – modula *fft*). Stoga, zamišljeno je da se vrši isključivo provera ispravnosti rada modula *fft2* sa različitim vrednostima veličine matrica, od 2x2 do 64x64 (širina i visina se mogu razlikovati), pri čemu će vrednosti elemenata matrica imati potpuno proizvoljne vrednosti.

5.3 Razvoj verifikacionog okruženja

Verifikaciono okruženje u ovom projektu je zasnovano na UVM (eng. *Universal Verification Methodology*). **UVM** predstavlja standardizovanu verifikacionu metodologiju sa razvijenim softverskim bibliotekama u *SystemVerilog* [12] jeziku. Svako verifikaciono okruženje razvijeno pomoću UVM se sastoji iz nekoliko međusobno povezanih komponenti sa jasno definisanom ulogom. Komponente su međusobno povezane preko TLM komunikacije.

Šematski prikaz realizovanog verifikacionog okruženja je prikazano na slici 23. Sastoji se iz sledećih komponenti: test, okruženje (eng. *environment*), sekvenci, sekvencera, paketa (eng. *sequence item*), monitora, drajvera, agenata, skorborda, referentnog modela i konfiguracije.



Slika 23 Šematski prikaz verifikacionog okruženja

5.3.1 Projektovanje sekvenci, paketa i sekvencera

Sekvence predstavljaju objekte u kojima se nalazi celokupna informacija o logičkim pobudama koje će biti prosleđene testiranom dizajnu (DUT-u). Sekvencer prosleđuje sekvence drajveru i kontroliše prosleđivanje jedne ili više sekvenci ka drajveru. Informacija koja se u jednom trenutku šalje naziva se paket.

Analizom interfejsa modula *fft2*, prikazanom u listingu 4, konstruisana su dva formata paketa, nazvana *init_transaction* (listing 5) i *dout_din_transaction* (listing 6). Paket *init_transaction* služi da prosleđuje veličinu matrice, start i ready signale, dok *dout_din_transaction* služi da prosleđuju vrednosti i adrese podataka.

```
class fft2_dout_din_transaction extends uvm_sequence_item;

    // fields
    rand logic [$clog2(FFT_SIZE*FFT_SIZE) - 1 : 0] data_i_addr_o;
    rand logic [$clog2(FFT_SIZE*FFT_SIZE) - 1 : 0] data_o_addr_o;
    rand logic [DATA_WIDTH - 1 : 0] dataRE_i;
    rand logic [DATA_WIDTH - 1 : 0] dataRE_o;
    rand logic [DATA_WIDTH - 1 : 0] dataIM_i;
    rand logic [DATA_WIDTH - 1 : 0] dataIM_o;
    rand logic data_rd_i;
    rand logic data_wr_i;
```

Listing 5 Format *dout_din* paketa

```
class fft2_init_transaction extends uvm_sequence_item;

    // fields
    rand logic [$clog2($clog2(FFT_SIZE)) - 1 : 0] log2w;
    rand logic [$clog2(FFT_SIZE) - 1 : 0] width;

    rand logic [$clog2($clog2(FFT_SIZE)) - 1 : 0] log2h;
    rand logic [$clog2(FFT_SIZE) - 1 : 0] height;

    rand logic start;
```

Listing 6 Format *init* paketa

Početak testa počinje slanjem proizvoljnih vrednosti širine i visine matrice (*width* i *height*), i.njihovih logaritama (*log2w* i *log2h*). Potom se šalju potpuno proizvoljne vrednosti podataka iz *din* sekvencera u *dout* drajver. Nakon što je *fft2* modul horizontalno obradio matricu, povratne vrednosti se skladište u privremeni fajl preko *dout* monitora. Potom, podatke iz privremenog fajla čita *din* sekvencer, i započinje vertikalnu obradu matrice.

5.3.2 Projektovanje drajvera

Uloga drajvera je da emulira signale koji će biti poslati uređaju. Direktno je povezan na interfejs od dizajna. Za razliku od sekvencera, koji se bavi slanjem podataka na visokom nivou, drajver treba da dobijene podatke na visokom nivou prosledi dizajnu koristeći postojeći interfejs. To uglavnom podrazumeva softversku emulaciju protokola, odnosno u slučaju *fft2* modula implementiran je četvorofazni hendšejk protokol. *Din* drajver emulira čitanje iz memorije (BRAM-ova), *dout* drajver emulira pisanje u memoriju, a *init* drajver emulira slanje veličine matrice i signala za početak obrade (*start*).

5.3.3 Projektovanje monitora

Monitor je pasivna komponenta, uloga mu je da osluškuje i nadgleda signale. Skupljene informacije o signalima prosleđuje ostalim komponentama. Dodatno, moguće je implementirati da se u njima vrši skupljanje pokrivenosti i provera vrednosti. U ovom projektu se u monitorima vrši skupljanje pokrivenosti [13].

Din monitor skuplja ulazne podatke dizajna. Referentnom modelu prosleđuje ulazne podatke za horizontalnu Furijeovu transformaciju, dok skorbordu prosleđuje signal data_i_addr_o.

Dout monitor skuplja izlazne podatke dizajna. Skorbordu prosleđuje sve pristigle informacije. Rezultate horizontalne Furijeove transformacije upisuje u privremeni fajl koji će kasnije biti ponovo prosleđeni dizajnu preko *din* sekvenci.

Init monitor skuplja podatke o visini, širini i njihovim logaritmima, i prosleđuje ih referentnom modelu.

U *din* i *dout* monitorima se skuplja pokrivenost ulaznih i izlaznih dataRE i dataIM signala (listinzi 7 i 8), sa ciljem da se utvrdi nad kolikim rasponom vrednosti je dizajn testiran. Nije se proveravala pokrivenost adresa (signali *data_i_addr_o* i *data_o_addr_o*), iz razloga što su ti podaci zavisni. Vrednosti adresa treba da pristižu okruženju po tačno određenom algoritmu (algoritam za obrtanje redosleda bita – pomenuto u poglavljima 2.5 i 4.4.2), čija se ispravnost proverava u skorbordu.

U *init* monitoru se proverava pokrivenost logaritma širine i visine matrice (matrice mogu biti isključivo veličine 2^n). Pošto nije planirano pokretanje IP jezgra nad matricama većim od 32x32, izabrano je da se meri pokrivenost za veličine matrica od 2x2 (n=1) do 32x32 (n=5). Dodatno, proverava kao i njihov *cross coverage*, pošto matrice ne moraju biti kvadratne, već proizvoljne veličine u datom opsegu (listing 9).

```
covergroup cg_fft2_din;
    option.per_instance = 1;

    cp_dataRE_i : coverpoint tr_collected.dataRE_i;
    cp_dataIM_i : coverpoint tr_collected.dataIM_i;

endgroup : cg_fft2_din;
```

Listing 7 Pokrivenost ulaznih signala

```
covergroup cg_fft2_dout;
    option.per_instance = 1;

    cp_dataRE_o : coverpoint tr_collected.dataRE_o;
    cp_dataIM_o : coverpoint tr_collected.dataIM_o;

endgroup : cg_fft2_dout;
```

Listing 8 Pokrivenost izlaznih signala

```
covergroup cg_fft2_init;
    option.per_instance = 1;

    cp_log2w : coverpoint tr_collected.log2w {
        bins range[5] = {[1:5]};
    }
    cp_log2h : coverpoint tr_collected.log2h{
        bins range[5] = {[1:5]};
    }
    cx_log : cross cp_log2h, cp_log2w;

endgroup : cg_fft2_init;
```

Listing 9 Pokrivenost veličine matrice

5.3.4 Projektovanje agenta i konfiguracije

Agent je klasa koja enkapsulira drajver, monitor i sekvencer. Omogućava apstraktniji prisitup i olakšava ponovno korišćenje datih komponenti.

Konfiguracioni objekat sadrži sve podatke neophodne da bi se konfigurisao testbenč. Pomoću konfiguracije je odrediti broj korišćenih agenata, koji će agenti biti pasivni (instancira se samo monitor) ili aktivni (sve tri komponente se instanciraju), da li će biti *master* ili *slave* uređaji, i tako dalje.

U ovom projektu su realizovani aktivni agenti *din*, *dout*, i *init* (kao što je objašnjeno u poglavlju 5.3.2). Od podešavanja moguće je konfigurisati da li će se skupljati pokrivenost.

5.3.5 Projektovanje referentnog modela

Referentni model predstavlja model verifikovanog sistema (DUT-a) koji je napisan na visokom nivou apstrakcije. Treba da obavlja istu funkcionalnost kao i verifikovani sistem. Pošto je referentni model napisan na visokom nivou apstrakcije, manja je šansa da nastane greška u kodu. Referentni model se koristi kao referenca (po tome je dobio i naziv) koja služi za poređenje odziva dobijenih iz DUT-a. Ukoliko postoji razlika u odzivu između DUT-a i referentnog modela, to znači da u nekom modelu postoji greška.

U ovom projektu je realizovan referentni model na nivou transakcija. Algoritam koji je opisan u poglavlju 2 i realizovan u poglavlju 3 u programskim jezicima *C++* i *SystemC*, je ponovo implementiran, ali u jeziku *SystemVerilog*. Sastoji se iz tri bitne celine: *butterfly*, *fft* i *run*.

- Funkcija *butterfly* vrši operacije kompleksnog sabiranja, oduzimanja i množenja po sledećem algoritmu (pogledati poglavlje 2.5), gde top i bot predstavljaju promenljive kompleksne matrice, a W_N^k rotacioni faktor.

$$top_{next} = top + W_N^k bot \tag{37}$$

$$bot_{next} = top - W_N^k bot \tag{38}$$

- Funkcija *fft* obavlja jednodimenzionalnu Furijeovu transformaciju. Drugim rečima, ona vrši iteraciju kroz jednu kolonu ili vrstu kompleksne matrice, i odabira elemente po šablonu prikazanom na slici 4 u drugom poglavlju.
- Funkcija *run* obavlja dvodimenzionalnu Furijeovu transformaciju. To podrazumeva iterativno pozivanje jednodimenzionalne Furijeove transformacije prvo nad svim vrstama pa svim kolonama matrice.

5.3.6 Projektovanje skorborda

Skorbord vrši proveru vrednosti dobijenih iz monitora i referentnog modela.

Implementirano je da se upoređuju vrednosti četiri signala – dataRE_o i dataIM_o (izlazni podaci iz DUT-a), kao i data_i_addr_o i data_o_addr_o (adrese sa kojih se učitavaju, odnosno na kojim se upisuju podaci u spoljašnjem BRAM-u). Signali dataRE_o, dataIM_o i data_o_addr_o se prosleđuju iz *dout* monitora, dok se signal data_i_addr_o prosleđuje iz *din* monitora. Iz tog razloga referentni model poseduje dva porta za TLM komunikaciju (listing 10).

```
forever begin
        repeat (width*height) begin
                din_expected_fifo.get(din_expected_tr);
                din_detected_fifo.get(din_detected_tr);
                compare_din();
        end
        repeat (width*height) begin
                dout_expected_fifo.get(dout_expected_tr);
                dout_detected_fifo.get(dout_detected_tr);
                compare_dout();
        end
```

Listing 10 Skorbord

5.3.7 Projektovanje okruženja

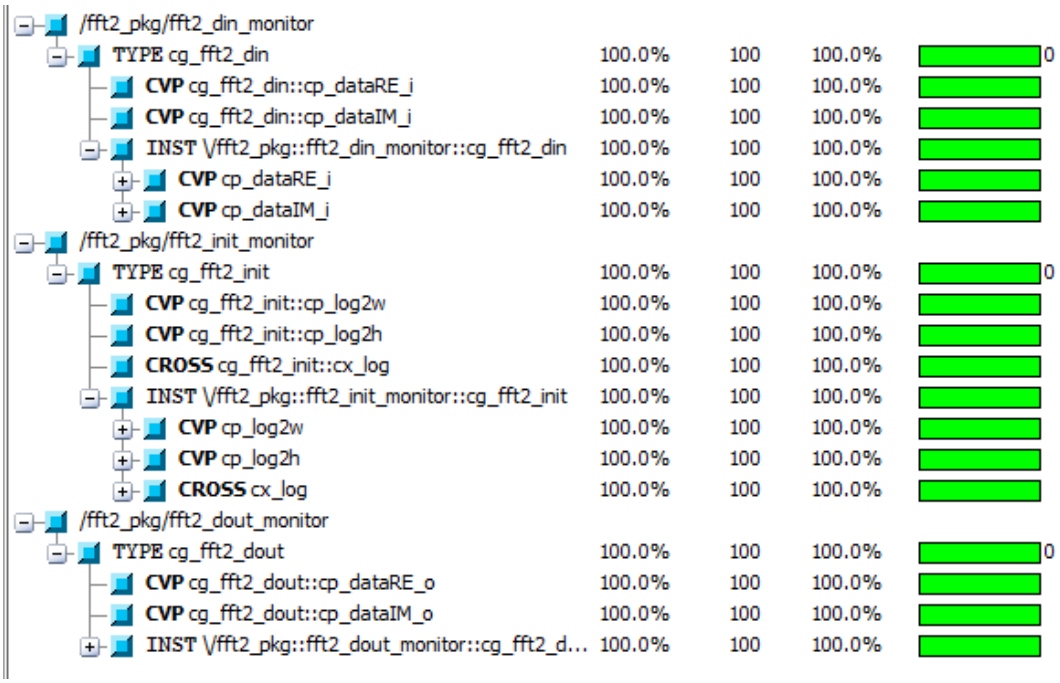
Klasa *environment* instancira i enkapsulira sve prethodno opisane komponente, za razliku od agenta koji samo enkapsulira drjver, monitor i sekvencer. Uloga okruženja i agenta je isključivo organizaciona. Verifikaciono okruženje je prikazano na slici 23.

5.4 Izvršavanje regresionih testova i skupljanje pokrivenosti

U jednom testu je moguće dobiti samo jednu vrednost za visinu i širinu i to je usko grlo za pokrivenost. Stoga, prilikom obavljanja testova najviše je potrebno voditi računa o pokrivenosti visine i širine. Eksperimentalno je potvrđeno da je 100 testova dovoljno da se dobije stopostotna pokrivenost za sve upite. U listingu 11 je prikazan kod za pokretanje regresionih testova, dok je na slici 24 prikazana funkcionalna pokrivenost nakon obavljenih testova.

```
# regression
for {set i 0} {$i < $num_seeds} {incr i 1} {
    vsim -c fft2_test_top -novopt -coverage \
        +UVM_VERBOSITY=UVM_LOW \
        +UVM_TESTNAME=fft2_test_simple \
        -sv_seed random
    # coverage save -assert -directive -cvg \
        -codeAll fft2-coverage-$i.ucdb
    # coverage save -assert -directive -cvg \
        -codeAll -onexit fft2-coverage-$i.ucdb
    onbreak {resume}
    run -all
    onbreak {resume}
    coverage save -assert -directive -cvg \
        -codeAll fft2-coverage-$i.ucdb
}
```

Listing 11 Regresija



Slika 24 Funkcionalna pokrivenost

Glava 6

Drajver za platformski uređaj

6.1 Uvod

Linuks (eng. *Linux*) je jedan od najpoznatijih i najviše korišćenih računarskih operativnih sistema današnjice. Veliku popularnost su mu donele činjenice da je besplatan, nije u posedstvu nijedne kompanije, razvijen je u skladu sa principima slobodnog softvera (eng. *Open Source*), što znači da svako ko poseduje kod može da ga menja i distribuira, i konstantno je održavan i unapređivan od strane velikog broja programera širom sveta.

Drajveri (eng. *Driver*) [14, 15] imaju specijalnu ulogu u Linuksovom jezgru (kernelu). Oni predstavljaju softverski sloj koji leži između aplikacije i hardvera. Potpuno kriju detalje funkcionisanja hardvera, i enkapsuliraju ga sa dobro definisanim interfejsom. Drugim rečima, drajveri predstavljaju spregu između hardvera i softvera (aplikacije), i olakšavaju komunikaciju između njih. S obzirom da postoji veliki broj različitih vrsta hardverskih uređaja, broj drajvera je takođe veoma veliki. Tempo pojavljivanja novih hardverskih uređaja i zastarevanja postojećih garantuje nepresušnu potrebu za drajverima. Osobina modularnosti omogućava da Linuksovo jezgro na datom računarskom ili embeded sistemu poseduje isključivo drajvere koji su mu potrebni.

Danas, povezivanje hardverskih komponenti sa procesorom se uglavnom vrši pomoću magistrala koje imaju mogućnost automatskog detektovanja periferijskih uređaja (na primer grafičkih i mrežnih karti). Kernel, prilikom podizanja sistema, komunicira sa uređajima na magistrali, uzima sve neophodne informacije i enumeriše ih. Sa druge strane, postoji veliki broj uređaja koji nisu povezani sa procesorom pomoću magistrala, pa ih procesor ne može automatski otkriti. Takvi uređaji se nazivaju **platformski uređaji**. Platformski drajveri, dakle, predstavljaju drajvere napisane za platformske uređaje.

Ukoliko u sistemu postoje platformski uređaji, neophodno je kernelu dati sve neophodne informacije o njima, što se može postići na više načina, međutim u ovom projektu je korišćena metoda sa stablom uređaja. **Stablo uređaja** (eng. *Device Tree*) predstavlja strukturu podataka u kojoj se nalaze sve neohpodne informacije o hardverskim resursima i koja se prosleđuje kernelu prilikom podizanja sistema. Razdvajanjem informacija o uređajima od kernela, izbegnuti su problemi kao što su zahtevi sa ponovnim kompajliranjem kernela prilikom nastanka bilo kakvih hardverskih promena i opisivanje uređaja unutar koda kernela [16].

U ovom projektu je razvijen drajver, kao i korisnička aplikacija, za sistem (platformski uređaj) prikazan na slici 20.

6.2 Razvoj platformskog drajvera

Implementirani hardverski sistem koji obavlja dvodimenzionalnu Furijeovu transformaciju je opisan u poglavlju 3. Da bi aplikacija mogla da koristi dati sistem, realizovan je modul koji ima pristup realnoj i imaginarnoj blok memoriji, i samom IP jezgru. Celokupan drajverski modul se sastoji iz sledećih osam funkcija i strukture *platform_driver*.

6.2.1 Struktura *platform_driver*

Struktura *platform_driver* (prikazana u listingu 12) ima ulogu u inicijalizaciji hardverskog sistema. Sastoji se iz podstrukture „driver“, i pokazivača na funkcije *fft2_probe* i *fft2_remove*, koje će biti kasnije objašnjene. Struktura *driver* se deli na sledeća polja:

- *name* - predstavlja ime drajvera, u ovom projektu makro *DRIVER_NAME* ima vrednost *FFT2*.
- *owner* - predstavlja vlasnika drajvera. Najčešće, kao i u ovom projektu, ima vrednost predefinisanog makroa *THIS_MODULE*.
- *of_match_table* - koja pokazuje na strukturu tipa *of_device_id* (prikazanu takođe u listingu 13).

Struktura *of_device_id* je niz koji se sastoji iz *compatible* polja, i njena uloga je da imenuje sve hardverske resurse koje koristi drajver. Imena upisana u strukturu *of_device_id*, treba da se nalaze i u stablu uređaja, na osnovu kojih kernel ima mogućnost da utvrdi fizičke adrese korišćenih resursa i poveže ih sa drajverom. Kao što se može videti, navedena su imena *compatible* polja od realizovanog IP jezgra za Furijeovu transformaciju (*xlnx,fft2*), realne i imaginarne blok memorije (*xlnx,bram_re* i *xlnx,bram_im*).

```
static struct of_device_id device_of_match[] = {
  { .compatible = "xlnx,fft2", },
  { .compatible = "xlnx,bram_re", },
  { .compatible = "xlnx,bram_im", },
  { /* end of list */ }
};

MODULE_DEVICE_TABLE(of, device_of_match);

static struct platform_driver my_driver = {
  .driver = {
    .name = DRIVER_NAME,
    .owner = THIS_MODULE,
    .of_match_table = device_of_match,
  },
  .probe = fft2_probe,
  .remove   = fft2_remove,
};
```

Listing 12 Strukture *platform_driver* i *of_device_id*

6.2.2 Funkcije *fft2_init* i *fft2_exit*

Funkcija *fft2_init* se poziva prilikom povezivanja modula sa kernelom pomoću *insmod* komande, dok se *fft2_exit* funkcija poziva isključivanjem modula iz kernela pomoću „rmmod“ komande. Drugim rečima, to su prva i poslednja funkcija koje će biti pozvane tokom života modula u kernelu.

Uloge *fft2_init* funkcije su sledeće:

- Dinamički da alocira prve slobodne upravljačke brojeve pomoću funkcije *alloc_chrdev_region*.
- Da napravi *inode* fajl (eng. *Index Node File*) u */dev* direktorijumu za svaki hardverski resurs (kojih u ovom slučaju ima tri, odnosno koliko ima i *compatible* polja – dva za blok memorije i jedan za IP jezgro) pomoću naredbe *device_create*. Ukoliko je pronađena odgovarajuća *compatible* vrednost u stablu uređaja, nakon *fft2_init* funkcije će biti pozvana *fft2_probe* funkcija.
- Da poveže kreirane *inode* fajlove sa funkcionalnostima koje treba da se izvrše prilikom pozivanja neke od funkcija za rad sa fajlovima. To se postiže pomoću c_dev strukture.
- Ukoliko neka od naredbi ne bude uspešna, sve već izvšene naredbe treba da se poništе.

Uloga *fft2_exit* funkcije je suprotna od *fft2_init* funkcije. Njena uloga je da oslobodi upravljačke brojeve i da obriše kreirane *inode* fajlove.

6.2.3 Funkcije *fft2_probe* i *fft2_remove*

Funkcija *fft2_probe* služi da se izvrši inicijalizacija i alociranje korišćenih hardverskih resursa. Suprotno tome, funkcija *fft2_remove* služi da se zaustavi rad i oslobode korišćeni hardverski resursi.

Prilikom poziva funkcije *fft2_probe*, obavljaju se redom sledeće naredbe:

- Pribavljaju se podaci o adresnom prostoru uređaja.
- Rezerviše se dati opseg adresa.
- Mapiraju se fizičke adrese na virtuelne.
- Ukoliko je neka naredba bila neuspešna, sve prethodne naredbe se poništavaju.

Pošto se funkcija *fft2_probe* poziva onoliko puta koliko ima hardverskih resursa, definisana je statička promenljiva *device_fsm* koja predstavlja konačan automat, odnosno vodi računa o aktuelnom hardverskom resursu treba koji se inicijalizuje.

6.2.4 Funkcije *fft2_open* i *fft2_close*

Iako postoji mogućnost da date funkcije obavljaju određenu funkcionalnost, u ovom projektu su korišćene isključivo radi debagovanje napisanog koda.

6.2.5 Funkcije *fft2_write* i *fft2_read*

Drajver omogućava da se upisuju dve matrice u memoriju (nazovimo ih A i B), i da se sa jednom komandom izvrše Furijeove transformacije nad obe matrice. Jedna matrica se upisuje na pozicije od 0 do 1023, dok se druga matrica upisuje na pozicije od 1024 do 2047. Upisivanje vrednosti matrica se vrši u oba *inode* fajla za blok memorije, pomoću komandi formata „pozicija, vrednost“. Pozivom komande formata „w_a, h_a, w_b, h_b“, gde su w_a i w_b širine matrica, a h_a i h_b visine matrica, nad *inode* fajlom za IP jezgro izvršava se sledeće:

1. Vrši se Furijeova transformacija nad matricom A.
2. Zamenjuju se mesta matrice A i B (IP jezgro koje obavlja Furijeovu transformaciju uvek počinje obradu od pozicije 0).
3. Vrši se Furijeova transformacija nad matricom B.
4. Ponovo se zamenjuju mesta matrice A i B.

Ukoliko se unese komanda formata „w_a, h_a, 0, 0“, izvršiće se Furijeova transformacija samo jedne matrice, i neće doći do zamene mesta u memoriji.

Čitanje se vrši preko funkcije *fft2_read*, i ispisivaće redom sve vrednosti matrica koje se nalaze u memoriji, u skladu sa zadatim veličinama w_a, h_a, w_b, h_b.

6.3 Razvoj aplikacije

Kao što je već napomenuto, krajnji cilj ovog rada je da se implementira konvolucija matrica korišćenjem brzih Furijeovih transformacija. Aplikacija izvršava sledeće korake:

1. Određuje veličinu rezultujuće matrice C po algoritmu *setBitNumber* (najmanji stepen dvojke koji je veći od zbira dva data broja), i proširuje matrice A i B do date veličine sa nulama.
2. Upisuje matrice A i B.
3. Šalje komandu za početak obrade „w_a, h_a, w_b, h_b“.
4. Iščitava ih, vrši kompleksno množenje obrađenih matrica i računa konjugaciju njihovog proizvoda.
5. Upisuje konjugovanu matricu.
6. Šalje komandu za početak obrade matrice C „w_c, h_c, *0, 0*“.
7. Iščitava konjugovanu matricu i deli sve vrednosti sa $(w_c \cdot h_c)$

Prilikom slanja i i čitanja vrednosti iz drajvera, neophodna je konverzija realnih vrednosti matrica (double) i 32-bitnih podataka sa fiksnom tačkom. U programibilnoj logici je prvih 16 bita rezervisano za ceo deo, dok je drugih 16 bita rezervisano za decimalne vrednosti. Funkcije koje vrše konverziju se zovu *int2double* i *double2int32*. Dodatno, vrednosti w_a, w_b, i w_c, kao i h_a, h_b, i h_c su jednake nakon proširenja matrica (korak 1). Pošto je maksimalna veličina matrice koja se može obraditi u drajveru veličine 32x32, ulazne matrice A i B ne smeju biti veće od 16x16.

6.4 Testiranje na *Zynq* razvojnoj ploči

U ovom projektu je korišćeno razvojno okruženje Petalinuks 2019.2 (eng. *PetaLinux*) za podizanje Linuks operativnog sistema na *Zybo* razvojnu ploču. Pomoću Petalinuksa su generisane četiri osnovne komponente neophodne za ispravan rad Linuksa: Linuks kernel, stablo uređaja, fajl sistem i butlouder (eng. *Bootloader* – zadužen je da kopira stablo uređaja na poznatu adresu u RAM-u). Date komponente su nakon generisanja prebačene na memorijsku karticu u *Zybo* ploči, kao i kodovi za drajver i korisničku aplikaciju.

Celokupan sistem je više puta testiran za matrice različitih veličina. Isprva su izabrane interesantne ulazne matrice veličine 4x4, čiji je rezultat konvolucije lako proveriti. Kasnije su korišćene matrice različitih veličina (do 16x16) sa nasumičnim vrednostima elemenata, i nijednom nije pronađena greška u rezultatu.

Glava 7

Zaključak

U ovom radu je izvršena akceleracija algoritma dvodimenzionalne konvolucije matrica. Softverski i hardverski deo su paticionisani preko ESL metodologije u programskom jeziku *SystemC*. Hardverski blok je uspešno realizovan u VHDL-u, korišćenjem RTL metodologije. Komunikacija sa memorijom je izvršena preko BRAM interfejsa. Ispravna funkcionalnost datog hardverskog modula je verifikovana preko UVM metodologije u *SystemVerilog* jeziku. Linuks drajver je implementiran i testiran na *Zynq* sistemu. Korisnička aplikacija komunicira sa hardverskim modulom preko prethodno pomenutog drajvera, i izvršava softverski deo celokupnog algoritma.

Dalji rad na ovom projektu bi mogao ići u pravcu implementacije hardverskog modula zasnovanog na brojevima sa pokretnom tačkom umesto na brojevima sa fiksnom tačkom, čime se može postići ispravan rad datog modula za veći raspon vrednosti. U tom slučaju je neophodno izmeniti samo blok koji izvršava leptir operaciju. Takođe, moguće je ukloniti četvorofazni hendšejk protokol, čime se broj stanja kroz koja algoritam mora da prođe smanjuje (automatski se smanjuje i vreme izvršavanja). Umesto BRAM interfejsa moguće je koristiti AXI *Stream* ili AXI *Full* interfejs, čime se može smanjiti vreme komunikacije između softverskog, hardverskog i memorijskog podsistema. Dodatna mogućnost za ubrzanje izvršavanja konvolucije predstavlja paralelizacija celokupnog algoritma, i softverskog i hardverskog dela, što u ovom projektu nije razmatrano.

Literatura

[1] Jeremy Fix, „Efficient convolution using the Fast Fourier Transform, Application in C++“,
2001

[2] Steven W. Smith, „The Scientist and Engineer's Guide to Digital Signal Processing”,
2001

[3] http://alwayslearn.com/DFT%20and%20FFT%20Tutorial/DFTandFFT_FFT_TwiddleFactor.html, pogledano dana 19.9.2019.

[4] https://www.geeksforgeeks.org/iterative-fast-fourier-transformation-polynomial-multiplication, pogledano dana 19.9.2019.

[5] Grant Martin, Brian Bailey, Andrew Piziali, "ESL Design and Verification: A Prescription for Electronic System Level Methodology (Systems on Silicon)",
Morgan Kaufmann, 2007

[6] Daniel Gajski, Samar Abdi, Andreas Gerstlauer, Gunar Schirner, "Embedded System Design: Modeling, Synthesis and Verification",
Springer, 2009

[7] Brian Bailey, Grant Martin, "ESL Models and their Application: Electronic System Level Design and Verification in Practice",
Springer, 2009

[8] Pong P. Chu, „RTL Hardware Design Using VHDL“,
Wiley-Interscience, 2006

[9] Peter J. Ashended, „The Designer’s Guide to VHDL“,
Morgan Kaufmann, 1996

[10] Andreas Meyer, „Principles of Functional Verification“,
Newnes, 2003

[11] https://www.researchgate.net/figure/verification-cycle_fig35_303994627, pogledano dana 17.7.2020.

[12] Chris Spear, Greg Tumbush, „SystemVerilog for Verification“,
Springer, 2012

[13] Andrew Piziali, „Functional Verification Coverage Measurement and Analysis“,
Springer, 2004

[14] Jonathan Corbet, Alessandro Rubini, Greg Kroah-Hartman, „Linux Device Drivers, Third Edition“,
O’Reilly, 2005

[15] Predrag Teodorović, Veljko Malbaša, „Mikroračunarski sistemi za rad u realnom vremenu“,
2010

[16] https://community.arm.com/developer/tools-software/oss-platforms/w/docs/268/device-tree, pogledano dana 22.10.2019.