Elektrotehnički fakultet
Univerzitet u Banjoj Luci

# IZVJEŠTAJ PROJEKTNOG ZADATKA

iz predmeta

# SISTEMI ZA DIGITALNU OBRADU SIGNALA

Student:
Ranko Mirosav

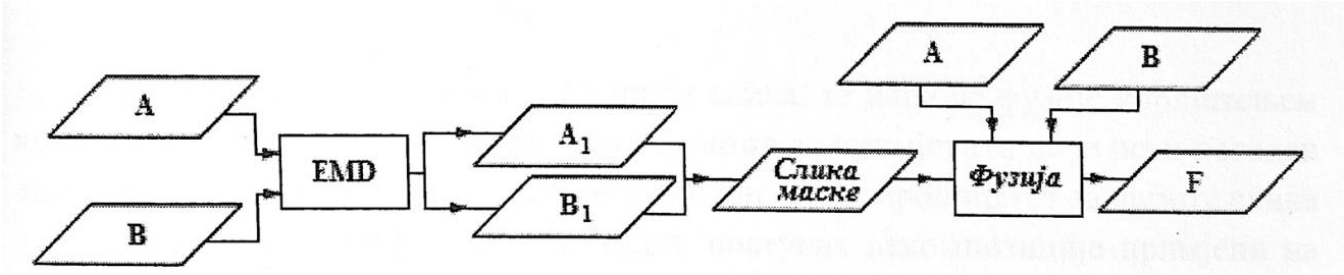
Predmetni nastavnici i asistenti:
prof. dr Mitar Simić
prof. dr Mladen Knežić
Dimitrije Obradović, dipl. inž. el
ma Vedran Jovanović

Februar 2025. godine

# 1. Opis projektnog zadatka

U sklopu projektonog zadatka je bilo potrebno realizovati sistem za generisanje empirijske vremensko-frekvencijske dekompozicije signala (Empirical Mode Decomposition – EMD) na razvojnom okruženju ADSP-21489. Osnovni dijelovi sistema su:

- **Učitavanje podataka:** Dio u kom se učitavaju dvije multifokusne slike, koje je potrebno pretvoriti u gray scale slike i od njih formirati 1D vektor čitanjem uzastopnih redova/kolona.
- **EMD dekompozicija:** Dio u kom se vrši određivanje prvog nivoa EMD tj. izdvajanje samo prve unutrašnje mod funkcije (IMF - Intristic Mode Functions). Od izdvojenih IMFs se formiraju dvije nove slike koje se prebacuju u 2D format dimenzija ulaznih slika.
- **Generisanje maske odlučivanja:** Dio u kom se algoritmom za odlučivanje dobre fokusiranosti na slikama, na osnovu prethodno izdvojenih IMF dvije ulazne slike, generiše maska odlučivanja. Algoritam radi tako što se računa lokalna varijansa piksela na svim lokacijama slika A1 i B1 na prozoru 3x3. Razlika lokalnih varijansi dvije slike se poredi sa pragom odlučivanja, na osnovu tih poređenja određuju se težinski koeficijenti za tu lokaciju. Na ovaj način se formira 2D matrica maske i ona daje informacije o mjestima dobre fokusiranosti na pojedinim slikama.
- **Formiranje slike u punom fokusu:** Nakon što se dobije maska odlučivanja, prolazeći kroz vrijednosti date maske i provjeravajući ih, vrijednosti ulaznih slika A i B se unose u izlaznu fokusiranu sliku. Ako je vrijednost piksela na maski 0, onda se piksel slike A prenosi u izlaznu sliku. Ako je piksel maske 1, onda se piksel slike B prenosi u izlaznu sliku. Ako je piksel maske 0.5 onda se u izlaznu sliku unosi srednja vrijednost piksela A i B. Blok dijagram opisanog algoritma je na slici 1.




Slika 1. Blok dijagram algoritma za fuziju multifokusnih slika A i B.

# 2. Izrada projektnog zadatka

Za učitavanje slike na razvojnu platformu je korišteno razvojno okruženje Cross Core Embedded Studio. Pošto je bilo potrebno realizovati rad sa podacima koji su u bmp i jpg formatu, shodno tome napravljena je python skripta koja uzima dvije slike koje su u paru, uzima podatke o pikselima datih slika, te njihovu širinu i visinu i upisuje ih u header fajl. Taj .h fajl se uključuje u glavni .c fajl projekta, a podaci o pikselima se smiještaju u korisnički definisanu memorijsku sekciju (slika 2). Ovim se dobijaju ulazni podaci o pikselima, smješteni u niz koji je tipa unsigned char i veličine WIDTH * HEIGHT *3 (puta 3 iz razloga jer se pikseli nalaze u RGB formatu).

Funkcija *convert_rgb_to_grayscale* konvertuje ulazne RGB nizove u gray scale. Funkcija prolazi kroz niz koji se nalazi u .h fajlu i uzima 3 podatka. Za svaki piksel uzimaju se RGB vrijednosti i uz pomoć forumule (2.1) konvertuju se u gray scale. U kodu je korištena formula (2.1.1) koja predstavlja formulu 2.1, prilagođenu za rad sa cijelobrojnim vrijednostima.

$$gray = R * 0.299 + G * 0.587 + B * 0.114 \quad (2.1)$$

$$gray\ integer\ value = (R * 77 + G * 150 + B * 29) \gg 8 \quad (2.1.1)$$

gdje je 77 = 0.299 * 256 , 150 = 0.578 * 256 , 29 = 0.114 * 256, kada pomnožimo RGB vrijednosti onda je potrebno samo još to pomjeriti udesno za 8 bita što predstavlja dijeljenje sa 256. Funkcija koristi for petlju sa dva brojača, gdje se sa jednim prolazi kroz RGB niz, a drugim popunjava gray scale niz.

Svi podaci o pikselima se smještaju u niz ili matricu za koje je kreirana posebna memorijska sekcija koja je smještena u datoteci *app.ldf*, čiji je prikaz dat na slici 2. Prikaz ulaznih nizova piksela koji su konvertovani u gray scale je dat na slici 5. Podaci se smještaju u sekciju ***seg_sdram1,*** koja je mapirana na fizičku ***mem_sram***, koja se nalazi u Blok 3 SRAM memorije, SRAM predstavlja internu statičku memoriju velikih brizina, malog kapaciteta.

Nakon učitavanja podataka o pikselima i njihovog konvertovanja u gray scale, može se započeti obrada multifokusnih slika. Prvi nivo EMD ulaznih slika, dobija se u funkciji *emd* koja prvo računa lokalne ekstreme ulaznog signala, te vrijednosti i poziciju na kojoj je pronađen taj ekstrem smiješta u niz koji je tipa *Extremum,* čija se struktura može vidjeti na slici 3. Za te lokalne ekstreme računamo gornju i donju anvelopu signala, koja će biti omotač signala. Potrebno je izvršiti interpolaciju anvelope, tj. kreirati cubic spline funkciju, čime ćemo povezati tačke lokalnih ekstrema. To se radi u funkciji *interpolate_envelope* gdje prvo popunimo početne i krajnje vrijednosti prije interpolacije, a nakon toga započinjemo interpolaciju gdje uzimamo dva susjedna ekstrema i popunjavamo vrijednosti između njih, koristeći linearnu interpolaciju između dvije

tačke (formula (2.2)), gdje promjenljiva *t* predstavlja normalizovanu poziciju između dva ekstrema i računa se prema formuli (2.3). Kompletan kod za linearnu interpolaciju možemo vidjeti na slici 4.

```
dxe_seg_sdram1
{

    INPUT_SECTIONS( $OBJECTS(seg_sdram1) )

} > mem_sram
```

Slika 2. Korisnički kreirana memorijska sekcija smještena u *app.ldf*

```
typedef struct {
    int idx_value;
    unsigned int  value;
} Extremum;
```

Slika 3. Struktura za smiještanje vrijednosti i pozicije lokalnog ekstrema

```
int start, end, j;
double start_value, end_value, t;
for (int i = 0; i < num_extrema - 1; i++) {
    start = extrema_idx[i].idx_value;
    end = extrema_idx[i + 1].idx_value;

    start_value = signal[start];
    end_value = signal[end];

    for (j = start; j <= end; j++) {
        t = (j - start) / (double)(end - start);
        envelope[j] = (unsigned int)((1 - t) * start_value + t * end_value);
    }
```

Slika 4. Prikaz koda za linearnu interpolaciju između tačaka lokalnih ekstrema

$$t = \frac{j - start}{end - start} \qquad (2.2)$$

$$envelope[j] = start\_value + t * (end\_value - start\_value) \qquad (2.3)$$

```
#pragma section("seg_sdram1")
unsigned int pixelsA[SIZE];

#pragma section("seg_sdram1")
unsigned int pixelsB[SIZE];
```

Slika 5 . Prikaz smiještanja ulaznih piksela slika u korisnički kreiranu memorijsku sekciju.

Nakon što se izračunaju gornja i donja anvelopa signala, računa se njihova srednja vrijednost, te se taj signal oduzima od originalnog signala. Rezultat razlike je prva IMF ulaznog signala. Kada se dobiju prve IMF ulaznih slika na osnovu njih se računa maska odlučivanja dobre fokusiranosti slika. Maska odlučivanja je tipa *unsigned char,* gdje jedan piksel maske može da ima vrijednost 0, 0.5 ili 1. Da bi se izbjegao rad sa floating point podacima, maska može da ima vrijednosti FIXED_POINT_SCALE. Kad vrijednost maske treba da bude 0.5 onda je to FIXED_POINT_SCALE/2 , što je 128 *unsigned char*. Računanje vrijednosti piksela maske se radi u funkciji *generate_decision_mask* koja koja prolazi kroz 2D sliku, uzima piksel po piksel i pozivom funkcije *compute_local_variance* računa lokalnu varijansu na prozoru 3x3. Za trenutne piksele računa srednju vrijednost i nakon toga računa kvadratno odstupanje od srednje vrijednosti za dati piksel slike. Varijansom nastojimo da minimiziramo gešku, što je varijansa manja, manja je i srednja vrijednost greške. Varijansu (kvadratno odstupanje od srednje vrijednosti) računamo prema formuli (2.4).

$$\sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2 \qquad (2.4)$$

Gdje je $x_i$ vrijednost piksela u prozoru 3x3, a $\mu$ srednja vrijednost piksela na datom prozoru. Vrijednost koju dobijemo dijelimo sa brojem piksela čiju srednju vrijednost smo računali. Na taj način dobijamo varijansu za pojedini piksel. Dati kod je prikazan na slici 6. U funkciji *generate_decision_mask,* nakon što izračunamo vrijednost varijanse za svaki piksel, razliku lokalnih varijansi poredimo sa pragom odlučivanja (što je prag bolji, bolji je i fokus na izlaznoj slici). Rezultat tog poređenja daje vijednost piksela maske, koji moze biti: 0 – crno , 1 – bijelo i 0.5 – sivo (vrijednost piksela nula predstavlja dobar fokus, a 1 loš fokus).

Nakon što je generisana slika maske, pristupa se generisanju slike izlaza u punom fokusu. Kroz masku se prolazi piksel po piksel i na osnovu vrijednosti piksela dodjeljuje se piksel iz ulaznih slika koji ima bolji fokus. Ako je vrijednost piksela maske 0.5 (FIXED_POINT_SCALE/2), računa se aritmetička sredina vrijednosti ulaznih piksela.

Generisanje slike izlaza u punom fokusu se radi u funkciji generate_focused_image koja prima pokazivače ulaznih slika i na osnovu vrijednosti piksela maske, određuje koji piksel ulazne slike će biti proslijeđen na izlaz. Kod ove funkcije je prikazan na slici 7.

```
    for (int i = -1; i <= 1; i++) {
        for (int j = -1; j <= 1; j++) {
             nx = x + i;
             ny = y + j;
            if (nx >= 0 && nx < WIDTH && ny >= 0 && ny < HEIGHT) {
                mean += pixels[nx][ny];
                count++;
            }
        }
    }
    mean /= count;
    for (int i = -1; i <= 1; i++) {
        for (int j = -1; j <= 1; j++) {
             nx = x + i;
             ny = y + j;
           if (nx >= 0 && nx < WIDTH && ny >= 0 && ny < HEIGHT) {
                variance += (pixels[nx][ny] - mean) * (pixels[nx][ny] - mean);
            }
        }
    }

    return variance / count;
}
```

Slika 6. Prikaz koda u kojim se računa lokalna varijansa piksela na prozoru 3x3

```
    for (int y = 0; y < HEIGHT; y++) {
        for (int x = 0; x < WIDTH; x++) {
             idx = y * WIDTH + x;

            if (mask[y][x] == 0) {
                pixelsOutput[idx] = pixA[idx];
            }
            else if (mask[y][x] == FIXED_POINT_SCALE) {
                pixelsOutput[idx] = pixB[idx];
            }
            else {
                pixelsOutput[idx] = (pixA[idx] + pixB[idx]) >> 1 ;
            }
        }
    }
```

Slika 7. Prikaz koda za generisanje izlazne fokusirane slike

Dio koda: idx = y * WIDTH + x; je dio u kom računamo indeks izlaznog niza na osnovu konverzije 2D vektora u 1D vektor, da bi olakšali i ubrzali sam upis slike u izlazni fajl.

Pozivom funkcije ***save_image_to_header*** izlaznu sliku pišemo u header fajl, gdje se kasnije uz pomoć python skripte *kon_header_in_image.py* izlazni header fajl prebacije u sliku. Potebno je dodati eksteziju za izlaznu sliku, a to se određuje na osnovu ulaznih slika. Ova python skripta čita podatke iz .h fajla i koristenjem PIL (Pillow) biblioteke konvertuje podatke u jpg ili bmp fajl.

Zaglavlje *cycle_count.h* sadrži dva makroa kojima se omogućava pristup sadržaju EMUCLK registra, koji prikazuje broj izvršenih ciklusa. Da bi se ovi makroi upotrebili potrebno je prije kompajliranja omogućiti DO_CYCLE_COUNTS, ovim se omogućuje prikaz procesorskih ciklusa. Parametri *count_start* i *count_stop* su deklarisani kao cycle_t tip koji deklarisan u pomenutom zaglavlju. Dobijeni rezultati su na osnovu JPG slike dimenzija: WIDTH 178, HEIGHT 134. Bez uključivanje bilo kakvih optimizacija rezultat potrebnih procesorskih ciklusa je sledeći:

- *Konverzija slika u gray scale*: 7 176 249
- *Računanje emd ulaznih slika*: 32 003 962
- *Konverzija 1D vektora u 2D*: 2 311 215
- *Generisanje slike maske*: 80 807 924
- *Formatiranje slike u punom fokusu*: 2 976 275
- *Pisanje izlazne slike u* header *fajl*: 34 773 333

Na osnovu prethodnih rezultata da se primjetiti da najviše procesorskih ciklusa zauzima generisanje slike maske. Razlog tome jesu neizbježne unutrašnje petlje, te provjere koje imamo u for petljama.

Ako se uključi optimizacije po vremenu, korištenjem pretprocesorske direktive:

#pragma optimize_for_speed

Dobijaju se sledeći rezultati za potreban broj procesorskih ciklusa:

- *Konverzija slika u gray scale*: 3 848 257
- *Računanje emd ulaznih slika*: 18 943 554
- *Konverzija 1D vektora u 2D*: 2 306 140
- *Generisanje slike maske*: 42 500 659
- *Formatiranje slike u punom fokusu*: 2 208 483

- *Pisanje izlazne slike u header fajl*: 34 396 708

Nakon što smo primjenili optimizaciju za brzinu vidimo da imamo značajno smanjenje broja procesorskih ciklusa za dio algoritma koju vrši konverziju u gray scale, za računanje emd ulaznih signala i najveće smanjenje je za generisanje slike maske. U ostalim dijelovima algoritma nije došlo do značajnog smanjenja broja procesorskih ciklusa.

Sledeća predprocesorsa direktiva koju uključujemo je :

#pragma no_vectorization

Vekotrizacija petlje - paralelno izvršavanje više od jedne iteracije u petlji može da uspori izvršavanje petlji sa veoma malim brojem iteracija u petlji, zbog prologa i epiloga. Primjenom ove predprocesorske direktive iznad ciljane petlje daje se naredba kompajleru da ne vektorizuje petlju, te blokira vektorizaciju svih petlji ako se napiše iznad funkcije. Direktivu smo primjenili iznad main funkcije. Dobijaju se sledeći rezultati za potreban broj procesorskih ciklusa:

- *Konverzija slika u gray scale*: 7 176 244
- *Računanje emd ulaznih slika*: 32 003 966
- *Konverzija 1D vektora u 2D*: 2 311 217
- *Generisanje slike maske*: 80 807 920
- *Formatiranje slike u punom fokusu*: 2 976 273
- *Pisanje izlazne slike u header fajl*: 34 796 708

Dobijeni rezultati ukazuju na to da ova optimizacija nije donijela nikakva poboljšanja, te da imamo gore rezultate nego što smo imali prije optimizacije.

Sledeća predprocesorska direktiva koju uključujemo je :

#pragma SIMD_for

Direktiva koja sa manjim stepenom sigurnosti govori kompajleru da se samo dvije iteracije parelelno mogu izvršavati, slična je predprocesorskoj drektivi #pragma vector_for (n) , gdje n govori koliko iteracija se može paralelno izvršavati. Primjenom SIMD direktive dobijeni su sledeci rezultati:

- *Konverzija slika u gray scale*: 7 176 574
- *Računanje emd ulaznih slika*: 32 003 986
- *Konverzija 1D vektora u 2D*: 2 311 221
- *Generisanje slike maske*: 80 807 929
- *Formatiranje slike u punom fokusu*: 2 976 265
- *Pisanje izlazne slike u header fajl*: 34 796 713

Da se primjetiti isto tako da ni ova optimizacija nije donijela poboljšanja. Po pitanju broja procesorskih ciklusa imamo reziltate slične onima bez optimizacije.

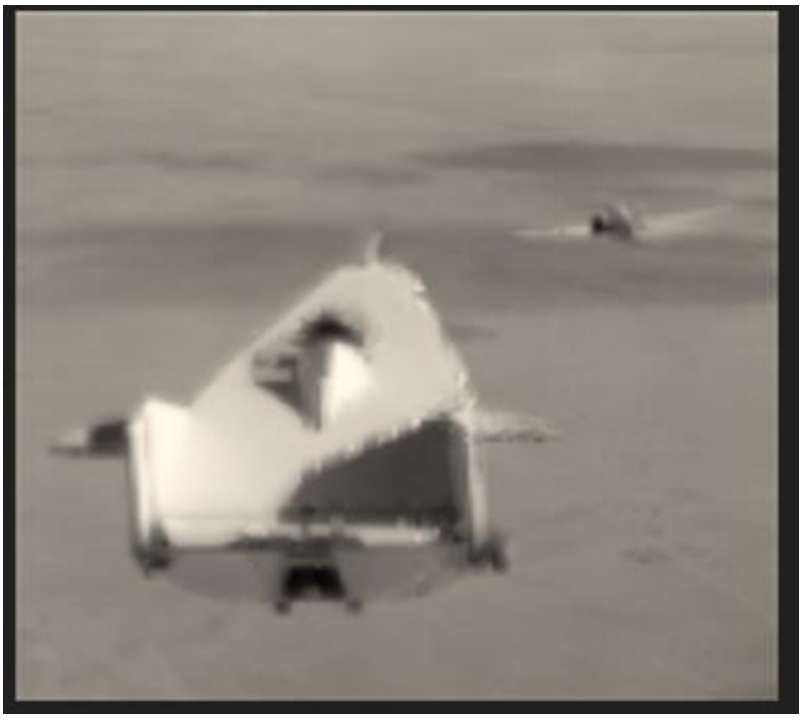
# 3. Python skripte

Za obradu ulaznih jpg i bmp fajlova koristi se python skripta *kon_image_in_header.py.* Ova scripta koristi PIL (Pillow) biblioteku da otvori sliku uz poziv funkcije *Image.open(image_path),* te da uz pomoć funkcije *img.convert("RGB")* konvertuje tu sliku u RGB ako nije u tom formatu i na kraju da iz te slike piksel po piksel izvuče RGB vrijednosti za svaki piksel i upiše ih u niz koji je tipa *unsigned char* koji zajedno sa podacima o veličini date slike upisuje u predviđen .h fajl, koji se kasnije uključuje u glavni program. Niz sa RGB vrijednostima piksela se smiješta u korisnički definisani memorijsku sekciju.

Za konverziju izlaznog .h fajla u sliku se koristi python skripta *kon_header_in_image.py*. Ova skripta takođe koristi Pillow (PIL) biblioteku. Iz biblioteke koristi funkciju za kreiranje nove slike *Image.new('RGB', (width, height)),* koja kreira sliku sa zadanim dimenzijama sa RGB bojama. Pored ove funkcije, koristi još funkciju *image.putdata(pixels)* koja postavlja listu piksela u sliku i funkciju *image.save(output_file)* koja kreiranu sliku čuva na datoj lokacij, u formatu( jpg ili bmp). Ova skripta će da otvori fajl i pročita sve linije u listu, izvući će podatke o dimenzijam slike i uz pomoć *int* funkcije prebaciti ih u cjelobrojne vrijednosti. Kada je *reading_pixels* postavljen na true počinjemo da čitamo podatke o pikselima, te svaku vrijednost konvertujemo u int i smiještamo u listu *pixel_values.* Pritom još provjerimo da li su di menzije liste piksela jednake očekivanim vrijednostima, a zatim kreiramo listu piksela u RGB foratu. Pošto očekujemo uvjek gray scale izlaz, za RGB komponete postavimo istu vrijednost. Na kraju kreiramo tu sliku, dodamo u nju listu piksela i sačuvamo je na željenoj lokaciji.

# 4. Rezultati


Slika 8. prikaz fokusirane slike od ulaza p30a i p30b.



Slika 9. Prikaz fokusirane slike od ulaza testna_slika6a i testna_slika6b

# 5. Zaključak

Jedan od nedostataka ovakve relizacije jeste ograničenost memorije razvojne platforme, koja ne može da obrađuju slike dimenzija većih od 180x180. Ovaj problem bi se mogao potencijalno riješiti blokovskim učitavanjem slika i njihovom djelimičnom obradom, obrada blok po blok generisanjem maske za taj blok, te formiranje slike izlaza i njen unos u izlazni fajl. Ovaj postupak ponavljati sve dok ima ulaznih blokova. Na ovaj način bi riješili problem memorijske ograničenosti razvojne platforme.

# 6. Literatura

1. *INFORMACIONE TEHNOLOGIJE - SADAŠNJOST I BUDUĆNOST - Božo Krstajić Aleksandra Radulović (2010)*