UNIVERZITET U BANJOJ LUCI
ELEKTROTEHNIČKI FAKULTET
RAČUNARSTVO I INFORMATIKA

SEGMENTACIJE SLIKE UPOTREBOM ALGORITMA ZA DETEKCIJU IVICA NA ADSP-21489 RAZVOJNOJ PLATFORMI

PROJEKTNI ZADATAK IZ PREDMETA SISTEMI ZA DIGITALNU OBRADU SIGNALA

Student
Jovana Arežina 1105/15

Predmetni nastavnik
prof. dr. Mladen Knežić
prof. dr. Mitar Simić

Banja Luka, februar, 2023.

SADRŽAJ

1. SPECIFIKACIJA PROJEKTNOG ZADATKA

U sklopu projektnog zadatka je potrebno realizovati sistem za manipulaciju slikom na razvojnom okruženju ADSP-21489. Sistem se sastoji iz 3 osnovna dijela:

● Učitavanje slike na ADSP-21489 razvojnu platformu. Slika koja se obrađuje će biti dostupna u *.bmp* formatu. Potrebno je iz slike u tom formatu izvući pojedinačne informacije o boji piksela i tako učitanu sliku pretvoriti u *gray scale* sliku.

● Detekcija ivica na učitanoj slici. Algoritam za detekciju ivica i njegove parametre izabrati na optimalan način i obrazložiti u izvještaju zašto je baš to urađeno. Algoritam treba uspješno da detektuje ivice na svim priloženim testnim slikama. Izlaz iz algoritma za detekciju ivica rezultuje novom slikom na kojoj je moguće identifikovati zatvorene konture (objekte).

● Bojenje objekata detektovanih u prethodnom koraku. Izvršiti kodovanje svakog piksela unutar detektovanih zatvorenih kontura tako da pikselima koji pripadaju istom objektu (konturi) bude dodijeljena ista vrijednost. Sliku sa identifikovanim ivicama i kodovanu sliku sačuvati *.bmp* formatu (čuvaju se dvije slike).

Svi navedeni koraci treba da se izvršavaju na ADSP ploči. Izvršiti optimizaciju svih koraka kroz koje navedeni sistem prolazi. Voditi računa o zauzeću memorijskih resursa i brzini izvršavanja. U izvještaju priložiti rezultate za različite pristupe optimizaciji koda. Obrazložiti dobijene rezultate.

2. REALIZACIJA POTREBNIH FUNKCIONALNOSTI

2.1. Korišteni hardverski i softverski reusrsi

Hardverski resursi korišteni pri izradi projektnog zadatka su: ADSP-21489 razvojna platforma i personalni računar (x86-64 procesor sa *Windows 10 OS*).

Korištena ***softverska okruženja*** su: *Cross Core Embedded Studio* (*CCES* u nastavku teksta) i *Visual Studio (Code).* Sve funkcionalne cjeline implementirane su u *C* programskom jeziku.

2.2. Realizacija funkcionalnosti

Realizacija funkcionalnosti može da se podijeli u 6 funkcionalnih cjelina i to su:

2.2.1. Učitavanje slike

Učitavanje slike u *bmp* formatu realizovano je u *Visual Studio CODE*-u korištnjem funkcije

reda_image () dostupne na linku datom u tekstu projektnog zadatka. Iz *bmp* slike su najprije pročitani podaci o dimenzijama (širini, visini i broju bajtova po pikselu), a zatim i informacije o vrijednosti pojedinačnih piksela. Zatim su te informacije smještene u *C header* fajl. Ulazna slika je slika u boji, pa se svaki piksel sastoji od 3 bajta za smiještanje informacija o R(*red*), G(*green*) i B(*blue*) komponentama piksela. Na Slici 2.1. prikazan je kod za upis ogdovarajućeg *.h* fajla. Taj *header* fajl se zatim uvozi u *CCES* i vrši se čitanje piksela i smiještanje u 1-D niz *pixels[]* veličine *WIDTH*HEIGHT*BYTESPERPIXEL*.

```
// Read first image
read_image("test_img1.bmp", &pixels, &width, &height,&bytesPerPixel);
if((fpl=fopen("test_img1.h", "w"))==NULL) {
   printf("Cannot open file.\n");
   return 1;
}
// Place the first image in the first header file
fprintf(fpl, "#pragma section(\"seg_hp2\")\n");
fprintf(fpl, "unsigned char pix[] = {");

for (int i = 0; i < width*height*bytesPerPixel; i++) {
    fprintf(fpl, "%u", pixels[i]);
    if (i < width*height*bytesPerPixel - 1) {
        fprintf(fpl, ", ");
    }
    else
       fprintf(fpl, "};\n");
}
```

Slika 2.1. Upis u *header* fajl

2.2.2. Pretvaranje slike u boji u gray scale sliku

Algoritam za pretvaranje slike u *gray scale* implementiran je množenjem R komponente svakog piksela ulazne slike sa vrijednosti 0.3, G komponente sa vrijednsoti 0.59 i B komponente sa vrijednosti 0.11. Zatim se te 3 vrijednosti sabiraju i dodjeljuju svakom bajtu trenutno obrađivanog piksela. Funkcija *grayscale()* koja implementira gore-navedeno realizovana je u *CCES* i izvršava se direktno na ADSP-21489 razvojnoj platformi. Na Slici 2.2. prikazan je kod funkcije *grayscale()*.

```
void grayscale(byte *pixels, int width, int height) {

    int j=0;
    for (int i = 0; i < width * height; i++) {
        int gray = (pixels[i * 3]*0.3 + pixels[i * 3 + 1]*0.59 + pixels[i * 3 + 2]*0.11);
        pixels[i * 3] = gray;
        pixels[i * 3 + 1] = gray;
        pixels[i * 3 + 2] = gray;
    }
}
```

Slika 2.2. Kod funkcije *grayscale()*

2.2.3. Detekcija ivica

Algoritam za detekciju ivica je takođe implementiran u *CCES* i izvršava se direktno na DSP procesoru. Algoritam je realizovan operacijom konvolucije između piksela slike dobijene kao izlaz prethodnog koraka (i konverzije iz 1-D u 2-D format) i odgovarajućih maski za ***Kirsch***, ***Prewitt***, ***Sobel*** i tzv. ***quick*** detektor ivica. To znači da su implementirana ukupno 4 detektora ivica koji koriste različite maske. Pri čemu su prva 3 detektora realizovana tako da koriste 8 različitih maski kako bi obavili operaciju konvolucije osam puta (jednom za svaki pravac) da bi detektovali sve ivice, dok *quick* detektor koristi samo 1 masku i operaciju konvolucije

obavlja samo jednom, zato i nosi naziv *quick*. Na Slici 2.3. prikazane su maske koje koristi svaki od pomenutih detektora ivica.

Kirsch	Prewitt	Sobel	Quick mask
5 5 5	1 1 1	1 2 1	-1 0 -1
-3 0 -3	1 -2 1	0 0 0	0 4 0
-3 -3 -3	-1 -1 -1	-1 -2 -1	-1 0 -1
-3 5 5	1 1 1	2 1 0	
-3 0 5	1 -2 -1	1 0 -1	
-3 -3 -3	1 -1 -1	0 -2 -2	
-3 -3 5	1 1 -1	1 0 -1	
-3 0 5	1 -2 -1	2 0 -2	
-3 -3 5	1 1 -1	1 0 -1	
-3 -3 -3	1 -1 -1	0 -1 -2	
-3 0 5	1 -2 -1	1 0 -1	
-3 5 5	1 1 1	2 1 0	
-3 -3 -3	-1 -1 -1	-1 -2 -1	
-3 0 -3	1 -2 1	0 0 0	
5 5 5	1 1 1	1 2 1	
-3 -3 -3	-1 -1 1	-2 -1 0	
5 0 -3	-1 -2 1	-1 0 1	
5 5 -3	1 1 1	0 1 2	
5 -3 -3	-1 1 1	-1 0 1	
5 0 -3	-1 -2 1	-2 0 2	
5 -3 -3	-1 1 1	-1 0 1	
5 5 -3	1 1 1	0 1 2	
5 0 -3	-1 -2 1	-1 0 1	
-3 -3 -3	-1 -1 1	-2 -1 0	

Slika 2.3. Maske korištene za detekciju ivica

U *CCES* algoritmi za detekciju ivica implementirani su sljedećim funkcijama:

- `setup_masks(int detect_type, int mask_0[3][3], int mask_1[3][3], int mask_2[3][3], int mask_3[3][3], int mask_4[3][3], int mask_5[3][3], int mask_6[3][3], int mask_7[3][3])`

  Ova funkcija postavlja vrijednosti 8 maski proslijeđenih kao parametri funkcije: *mask_0* do *mask_7* na odgovarajuće vrijednosti maski u zavisnosti od parametra *detect_type* koji određuje o kojem detektoru se radi (*1-KIRSCH, 2-PREWITT, 3-SOBEL*).

- `perform_convolution(int detect_type, int rows, int cols, int high)`

Ova funkcija predstavlja srž algoritma za detekciju ivica. Ona prvo poziva funkciju *setup_masks()* da kopira odgovarajuće vrijednosti maski, zatim definiše 2 vrijednosti *min=0* i *max=255*, nakon čega prolazi kroz sve vrijednosti piksela i računa konvolucionu sumu po 8 smjerova korištenjem 8 maski, pri čemu su smjerovi "djelovanja" maski *mask_0* do *mask_7* dati sa

3 2 1

4 X 0

5 6 7

gdje sa X označena pozicija trenutnog piksela koji se obrađuje. Za svaku od 8 konvolucionih suma se provjerava da li su veće od *max* vrijednosti ili manje od *min* vrijednosti. Ako je suma veća od *max*, sumi se dodijeli vrijednost *max*, a ako je manja od *min* dodjeljuje se vrijednost *min*. Zatim se provjerava da li je suma veća od vrijednosti trenutnog piksela izlazne slike, ako jeste njemu se dodjeljuje vrijednost konvolucione sume, ako nije zadržava staru vrijednost. Vrijednosti *min* i *max* zapravo predstavljaju ***crnu*** i ***bijelu*** boju, respektivno. Dodatno, kako bi se eliminisali svi sivi pikseli iz izlazne slike (tj. pikseli koji nisu ni crni ni bijeli) po završetku konvolucija prolazi se kroz cijelu izlaznu sliku i svim pikselima koji imaju vrijednost veću od paramtera *high* dodjeljuje se vrijednost 0, a svim pikselima koji imaju vrijednost manju od *high* dodjeljuje se vrijednost 255. Čime se postiže da pikseli unutar segmenata oivičenih detektovanim crnim ivicama budu bijeli.

- `void fix_outer_edges(void)`

Ova funkcija prolazi kroz sliku dobijenu prethodnim korakom i svim spoljašnjim ivičnim pikselima (koji omeđuju sliku) dodjeljuje vrijednost 0 (pošto oni takođe trebaju biti crni, a primjenom funkcije *perform_convolution()* su ostali bijeli) dok svim ostalim pikselima zadržava staru vrijednost.

- `void detect_edges(int detect_type, int high, int rows, int cols)`

Ovo je funkcija koja prvo poziva funkciju *perform_convolution()*, a zatim funkciju *fix_outer_edges()*.

- `void quick_edge(int high, int rows, int cols)`

Ova funkcija je realizovana na sličan način kao i funkcija *perform_convolution()*, sa tom razlikom da se ovde konvoluciona suma računa samo jednom (sa *quick* maskom), a ostatak postupka je isti. I ona na kraju poziva funkciju *fix_outer_edges()*.

Napomena: Detaljan opis parametara ovih funkcija dat je u komentarima uz deklaracije funkcija u header fajlu ImageProcessing.h. Takođe, njohov kod nije neaveden ovdje zbog njegove glomaznosti.

2.2.4. Kodovanje slike

Na Slici 2.4. prikazan je kod funkcije *encode_image()* kojom je implementirano kodovanje.

```
void encode_image(void)
{
    byte colors = 40;

    #pragma SIMD_for
    for (int i = 0; i < HEIGHT; i++) {
       for (int j = 0; j < WIDTH; j++) {
          if(edge_detection_pixels[i][j] == BLACK)
              edge_detection_pixels[i][j] = BLACK;
          else if(edge_detection_pixels[i][j]==WHITE && edge_detection_pixels[i-1][j]==BLACK
                  && edge_detection_pixels[i][j-1]==BLACK && (edge_detection_pixels[i-1][j+1]!=BLACK))
              edge_detection_pixels[i][j]=edge_detection_pixels[i-1][j+1];
          else if(edge_detection_pixels[i][j]==WHITE && edge_detection_pixels[i-1][j]==BLACK
                  && edge_detection_pixels[i][j-1]==BLACK)
          {
              edge_detection_pixels[i][j] = colors;
              colors += 40;
          }
          else if(edge_detection_pixels[i][j]==WHITE && (edge_detection_pixels[i-1][j]!=BLACK))
               edge_detection_pixels[i][j]=edge_detection_pixels[i-1][j];
          else if(edge_detection_pixels[i][j]==WHITE && (edge_detection_pixels[i][j-1]!=BLACK))
               edge_detection_pixels[i][j]=edge_detection_pixels[i][j-1];
          //else if(edge_detection_pixels[i][j]==255 && edge_detection_pixels[i-1][j]==0
          //&& edge_detection_pixels[i][j-1]==0 && (edge_detection_pixels[i-1][j+1]!=0))
               //edge_detection_pixels[i][j]=edge_detection_pixels[i-1][j+1];
       }
    }
}
```

Slika 2.4. Kod funkcije *encode_image()*

Funkcija najprije inicijalizuje početnu kodnu vrijednost *colors* na 40 (ova vrijednost može biti i drugačija), nakon čega prolazi kroz sliku dobijenu kao izlaz detektora ivica i provjerava da li je trenutni piksel crni, ako jeste, zadržava staru vrijednost. Ako je trenutni piksel bijeli i ako su njegovi susjedi gore i lijevo crni i ako je njegov susjed gore-desno različit od crne boje, trenutnom pikselu se dodjeljuje ta vrijednost. Ako je trenutni piksel bijeli i ako su njegovi susjedi gore i lijevo crni, onda se pikselu dodijeli trenutna kodna vrijednost *colors*, a zatim se ta vrijednost uveća za 40 (inkrement može biti drugačiji, s tim da bude dovoljno velik da se kasnije na obojenoj slici vizuelno jasno može detektovati razlika). Ako je trenutni piksel bijeli i njegov susjed gore je različit od crne boje, trenutnom pikselu se dodjeljuje vrijednost tog piksela. Isto važi i ako je trenutni piksel bijeli i njegov susjed lijevo različit od crne.

2.2.5. Bojenje slike

Bojenje slike je implementirano inverznim *greyscale* algoritmom koji se primjenjuje na piksele dobijene kodovanjem slike (i konvertovanjem slike iz 2-D u 1-D format). Na Slici 2.5. dat je kod inverzne *grayscale* funkcije, *reverse_grayscale()*.

2.2.6. Čuvanje slike

Nakon završene obrade, pikseli ***slike sa detektovanim ivicama*** i ***obojene slike*** se u *CCES* smiještaju u *header* fajlove *edge_detected_pixels.h* i *colored_pixels.h*, respektivno, nakon čega se ti fajlovi uvoze u *Visual Studio Code* projekat gdje se vrši upis slike u *bmp* format korištenjem funkcije *write_image()* dostupne na linku datom u tekstu projektnog zadatka. Na Slici 2.6. dat je kod koji u *CCES* vrši upis slike sa detektovanim ivicama u odgovarajući *header* fajl.

```
void reverse_grayscale(byte *pixels, int width, int height) {

    //#pragma SIMD_for
    for (int i = 0; i < width * height; i++) {
        int gray = pixels[i * 3];
        int red = gray / 0.3;
        int green = gray / 0.59;
        int blue = gray / 0.11;

        if(red<255)
            pixels[i * 3] = red;
        else
            pixels[i * 3] = red%256;

        if(green<255)
            pixels[i * 3 + 1] =  green;
        else
            pixels[i * 3 + 1] =  green%256;

        if(blue<255)
            pixels[i * 3 + 2] = blue;
        else
            pixels[i * 3 + 2] = blue%256;
    }
}
```

Slika 2.5. Kod funkcije *reverse_grayscale()*

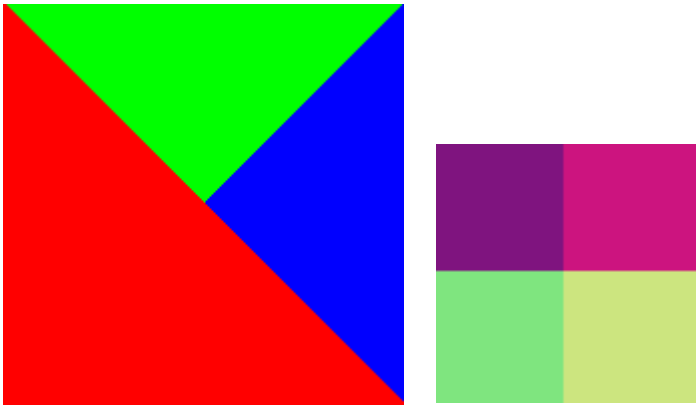
```
fprintf(fp1, "#define WIDTH %d\n", WIDTH);
fprintf(fp1, "#define HEIGHT %d\n", HEIGHT);
fprintf(fp1, "unsigned char edge_detection_pixels[%d][%d] = {\n", WIDTH, HEIGHT);

START_CYCLE_COUNT(start_count);
//#pragma SIMD_for
for (int i = 0; i < HEIGHT; i++){
    fprintf(fp1, "{ ");
    for(int j=0; j< WIDTH; j++){
        fprintf(fp1, "%u", edge_detection_pixels[i][j]);
        if (j < WIDTH - 1) {
            fprintf(fp1, ", ");
        }
        else if (i == (HEIGHT-1))
            fprintf(fp1, "}");
        else
            fprintf(fp1, "},\n");
   }
}
fprintf(fp1, "};\n");
```

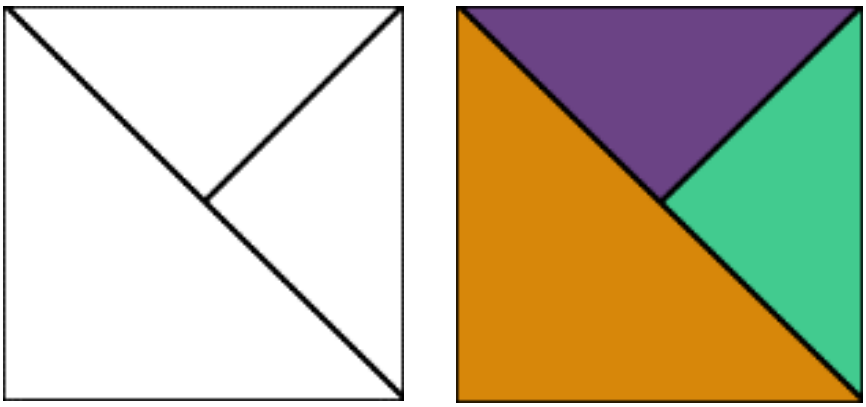
Slika 2.6 Kod za upis piksela slike sa detektovanim ivicama u *header* fajl

3. REZULTATI IZVRŠAVANJA

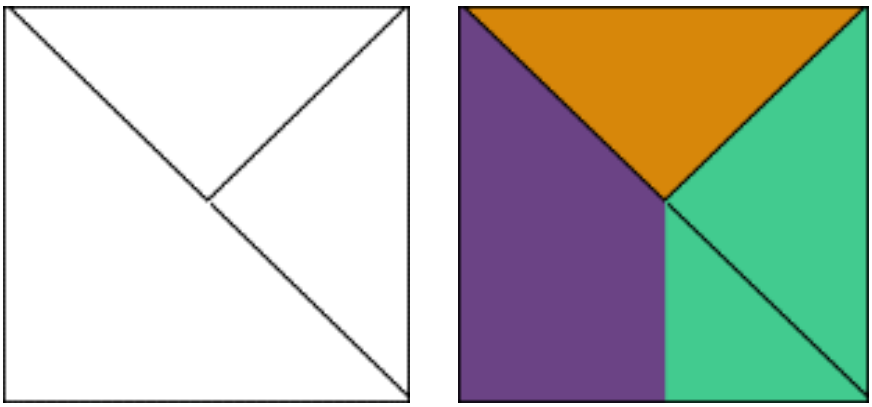
Verifikacija funkcionalnosti implemetiranih algoritama sprovedena je na 2 testne slike: *test_img1.bmp* dimenzija 200x200 piksela i *test_img2.bmp* dimenzija 100x100 piksela. Na slikama ispod prikazani su rezultati izvršavanja algoritama nad ulaznim testnim slikama.



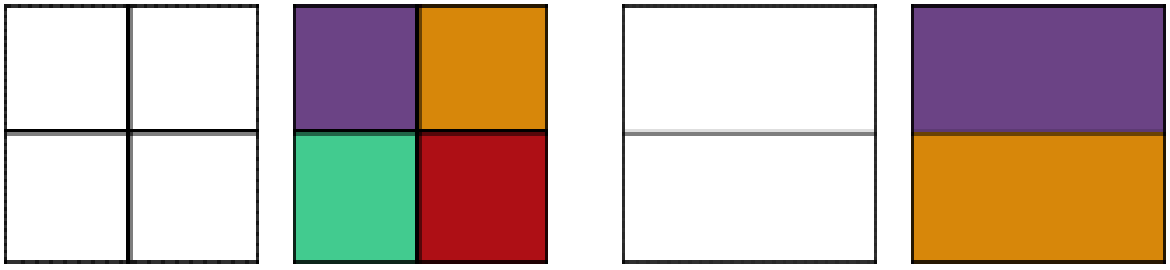
Slika 3.1. Originalne testne slike: *test_img1* (lijevo) i *test_img2* (desno)



Slika 3.2. *Sobel* detektor ivica na *test_img1*: detektovane ivice (lijevo), obojena slika (desno)



Slika 3.3. Q*uick* detektor ivica na *test_img1*: detektovane ivice (lijevo), obojena slika (desno)



Slika 3.4. Primijenjen *Sobel* (lijevo) i *quick* detektor ivica (desno) na *test_img2* – u oba slučaja su prikazane slike sa detektovanim ivicama i obojene slike

Rezultati izvršavanja za *Prewitt* i *Kirsch* detektor ivica su analogni kao kod *Sobel* detektora ivica, pa nisu priloženi ovde.

Na osnovu rezultata izvršavanja može se zaključiti da detektori ivica koji vrše konvoluciju u 8 smjerova (*Sobel, Kirsch* i *Prewitt*) detektuju ivice slike na ispravan način, kao što je i očekivano, dok *quick* detektor ivica koji računa samo jednu konvolucionu sumu ne detektuje ivice na očekivan način. Međutim, njegova prednost se sastoji u tome što koristi manje procesorskih resursa za svoje izvršavanje. O zauzeću resursa će biti više riječi u nastavku.

4. ZAUZEĆE RESURSA I OPTIMIZACIJA

4.1. Zauzeće meorije

ADSP-21489 ima internu statičku memoriju (SRAM) za instrukcije i podatke koja je podijeljena u 4 bloka. Takođe, na razvojnoj ploči ADSP-21489 postoji eksterna memorija sa kojom je processor povezan preko eksternog porta.

U nastavku slijedi pregled osnovnih problema detektovanih prilikom izrade zadatka, a tiču se zauzeća memorije.

4.1.1. Problemi sa zauzećem memorije

Prilikom izrade zadatka svi nizovi korišteni za skladištenje piksela slike (bilo 1-D ili 2-D) u *CCES*-u definisani su kao globalne promjenjive tipa *byte (unsigned char)* i cjelokupna obrada slike svodi se na obradu pojedinačnih bajtova. Pri pokušaju definisanja nizova kao što je prikazano ispod

```
byte pixels_2d[WIDTH][HEIGHT];
byte edge_detection_pixels[WIDTH][HEIGHT];
byte edge_detection_pixels_tmp[WIDTH][HEIGHT];
byte pixels_1d[WIDTH*HEIGHT*BYTESPERPIXEL];
byte pixels[WIDTH*HEIGHT*BYTESPERPIXEL];
```

i nakon *build*-anja projekta javile su se greške prikazane u nastavku.

```
[Error li1040] "D:\CrossCore-Primjeri\ImageProject\system\startup_ldf\app.ldf":313
Out of memory in output section 'dxe_block1_dm_data_prio3' in processor 'p0'
        Total of 0x15fd4 word(s) were not mapped.
For more details, see 'linker_log.xml' in the output directory.
```

```
[Error li1040] "D:\CrossCore-Primjeri\ImageProject\system\startup_ldf\app.ldf":377
Out of memory in output section 'dxe_block3_dm_data_prio3' in processor 'p0'
        Total of 0x15fd4 word(s) were not mapped.
For more details, see 'linker_log.xml' in the output directory.
```

```
[Error li1040] "D:\CrossCore-Primjeri\ImageProject\system\startup_ldf\app.ldf":474
Out of memory in output section 'dxe_block2_overflow_data' in processor 'p0'
```

```
        Total of 0x15fd4 word(s) were not mapped.
For more details, see 'linker_log.xml' in the output directory.
```

Ovo znači da je došlo do preteka memorijskih blokova koji se mogu identifikovati u okviru projekta u *CCES* u fajlu ***/system/startup_ldf/app.ldf*** i to je prikazano ispod.

```
   ** ------------------------- BLOCK 1 ---------------------------------------
   ** 0x000B2000 to 0x000BDFFF  Normal word (32) Space (1.5 Mbit RAM)
   mem_block1_dm32         { TYPE(DM RAM) START(0x000B2000) END(0x000BDFFF)
WIDTH(32) }
   ** ------------------------- BLOCK 2 ---------------------------------------
   ** 0x000C0000 to 0x000C7FFF  Normal word (32) Space (1 Mbit RAM)
   mem_block2_pm32         { TYPE(PM RAM) START(0x000C0000) END(0x000C7FFF)
WIDTH(32) }

   ** ------------------------- BLOCK 3 ---------------------------------------
   ** 0x000E0000 to 0x000E7FFF  Normal word (32) Space (1 Mbit RAM)
   mem_block3_dm32       { TYPE(DM RAM) START(0x000E0000) END(0x000E7FFF) WIDTH(32)
}
```

4.1.2. Rješenje problema

Kako bi se riješio prethodni problem i obezbijedila dovoljna količina memorije za skladištenje 30 000 bajtova (100x100x3) u slučaju testne slike *test_img2*, tj. 120 000 bajtova (200x200x3) u slučaju testne slike *test_img1* ili slika drugih dimenzija, definisana je korisnička memorijska sekcija ***seg_hp2*** u eksternoj SRAM memoriji dodavanjem sljedećeg koda

```
dxe_seg_hp2
{
      INPUT_SECTIONS( $OBJECTS(seg_hp2) )
} > mem_sram
```

u */system/startup_ldf/app.ldf* fajl na kraju oblasti *SECTIONS*.

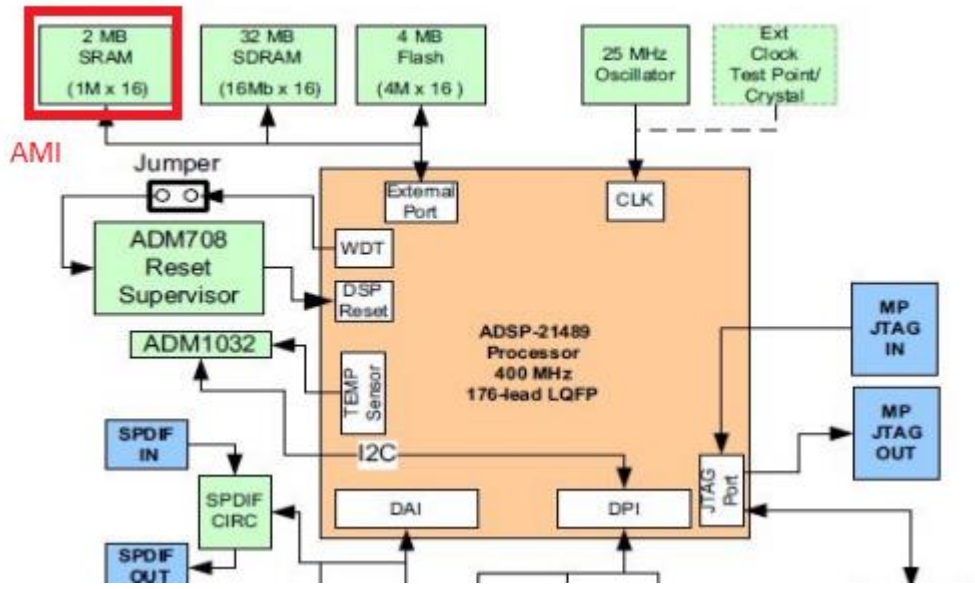
Nakon toga, gore-navedeni globalni nizovi se sada mogu definisati u okviru novokreirane sekcije memorije korištenjem pretprocesorske direktive ***#pragma section("seg_heap2")*** kao što je prikazano ispod.

```
#pragma section("seg_hp2")
byte pixels_2d[WIDTH][HEIGHT];
#pragma section("seg_hp2")
byte edge_detection_pixels[WIDTH][HEIGHT];
#pragma section("seg_hp2")
byte edge_detection_pixels_tmp[WIDTH][HEIGHT];
#pragma section("seg_hp2")
byte pixels_1d[WIDTH*HEIGHT*BYTESPERPIXEL];
#pragma section("seg_hp2")
byte pixels[WIDTH*HEIGHT*BYTESPERPIXEL];
```

Ovo znači da je problem neuspješnog zauzeća memorije riješen korištenjem eksterne SRAM memorije čiji je adresni prostor dat sa

```
mem_sram { TYPE(DM RAM ASYNCHRONOUS) START(0x0C000000) END(0x0C07FFFF) WIDTH(16) }
```

Ova memorija je preko ***AMI - asinhronog memorijskog interfejsa*** povezana sa ADSP-21489 procesorom kao što je prikazano na Slici 4.1.



Slika 4.1. Veza procesora i eksterne SRAM memorije

4.2. Zauzeće procesorskih resursa i optimizacija

U Tabeli 4.1. su dati svi algoritmi implementirani u okviru projektnog zadatka i broj procesorskih ciklusa potrebnih za njihovo izvršavanje bez uključene optimizacije i korištenjem različitih tehnika optimizacije. Rezultati su dobijeni na testnoj slici dimenzija 100x100. Procesorski ciklusi su izmjereni korištenjem funkcija iz *cycle_count* zaglavlja. U prvoj koloni tabele su rezultati dobijeni bez optimizacije. U drugoj koloni su rezultati dobijeni uključivanjem optimizacije kompajlera po brzini (***#pragma optimize_for_speed***). U trećoj i četvrtoj koloni su rezultati dobijeni sa uključenom optimizacijom kompajlera po brzini i primijenjenim pretprocesorskim direktivama ***#pragma no_vectorization*** i ***#pragma SIMD_for***, respektivno, iznad ciljnih petlji u okviru algoritama. *#pragma no_vectorization* daje naredbu kompajleru da ne vektorizuje petlju (onemogućuje paralelno izvršavanje više od jedne iteracije u petlji), a *#pragma SIMD_for* daje naredbu komajleru da se 2 iteracije mogu paralelno izvršavati.

Tabela 4.1. Broj ciklusa potrebnih za izvršavanjae algoritama uz različite tehnike optimizacije

ALGORITAM		**TERHNIKA OPTIMIZACIJE**							
		Bez optimizacije		***#pragma optimize_for_speed***		***#pragma no_vectorization***		***#pragma SIMD_for***	
Čitanje slike		1680009		1080005		1080005		1080005	
Konverzija u *grayscale*		1680037		1060024		1060023		1060023	
Detekcija ivica [*Sobel*]	**Detekcija ivica [*Quick*]**	**51544285**	**8340251**	**14275382**	**2718440**	14275382	2718440	14405394	2718440
Upis segmentirane slike		14126426		13884245		13884251		13884242	
Kodovanje slike		1869494		1170437		1170437		1170437	
Bojenje slike		1602081		900035		900035		900035	

Upis obojene slike	40822629	**40126206**	40126211	40126206

Na osnovu rezultata prikazanih u tabeli može se zaključiti da najviše procesorskih ciklusa (**51 544 285**) zahtijeva **detekcija ivica *Sobel*-ovim detektorom** (analogno važi i za ***Prewitt*** i ***Kirsch*** detektore) što je posljedica računanja 8 konvolucionih suma. S druge strane, ***quick*** detektor ivica troši skoro 7x manje ciklusa i ovo je očekivani rezultat koji je i ranije pomenut, s obzirom da se pri detekciji ivica računa samo 1 suma. Sljedeći najzahtjevniji algoritam je upis obojene slike u *header* fajl (40 822 629 ciklusa) i prati ga upis segmentirane slike (14 126 426).

Nakon uključenja optimizacije kompajlera vidimo da se broj procesorskih ciklusa za sve algoritme značajno smanjio, izuzev algoritma za upis obojene slike (broj ciklusa je manji za približno 700 000), pa je sada najkritičniji algoritam za upis obojene slike (**40 126 206**). To je posljedica mnogobrojnih poziva funkcije ***fprintf().*** Takođe, možemo zaključiti da naredbama *no_verctorization* i *SIMD_for* ne unosimo nikakve značajnije promjene po pitanju uštede broja procesorskih ciklusa, pa ćemo u slučaju algoritama za upis, kao najkritičnijih tačaka zauzeća procesora, pokušati pribjeći još nekoj od tehnika vektorizacije.

Naime, u okviru algoritama za upis slike možemo detektovati provjere uslova:

- Kod upisa segmentirane slike (ukupan broj iteracija je 10 000): `if(j < (WIDTH - 1))`
- Kod upisa obojene slike (ukupan broj iteracija je 30 000):
  `if(i < (WIDTH*HEIGHT*BYTESPERPIXEL - 1)`

Ovde se može iskoristiti ugrađena funkcija ***expected_true()*** iz ***builtins*** zaglavlja, zato što je očekivani rezultat provjere ovih uslova ***true*** u svim slučajevima osim za *i = WIDTH* u prvom slučaju i *i = WIDTH*HEIGHT*BYTESPERPIXEL* u drugom slučaju. U Tabeli 4.2. prikazani su rezultati primjene funkcije *expected_true()*.

Tabela 4.2. Rezultati primjene ugrađene funkcije *expected_true()*

Upis segmentirane slike	Upis obojene slike
13 884 253	40 126 195

Na osnovu tabele zaključujemo da se za slučaj upisa obojene slike broj ciklusa smanjio za 10-15 ciklusa, što je u poređenju sa cifrom od oko 40 miliona ciklusa gotovo zanemarljivo.

4.3. Nedostaci implementacije i prijedlozi poboljšanja

- ***Nedostaci:*** Algoritmi upisa troše suviše procesorskih ciklusa i izvršavaju se sporo.
- ***Prijedlozi za poboljšanje:*** Implementacija čitanja i upisa slike u *bmp* formatu direktno na razvojnoj platformi, realizacija pojačanja ivica (engl. *edge enhancement*) prije algoritma za detekciju ivica u slučaju korištenja kompleksnih slika, izbjegavanje negativnih indeksa korištenih kod implementacije konvolucionih suma jer mogu degradirati performanse.

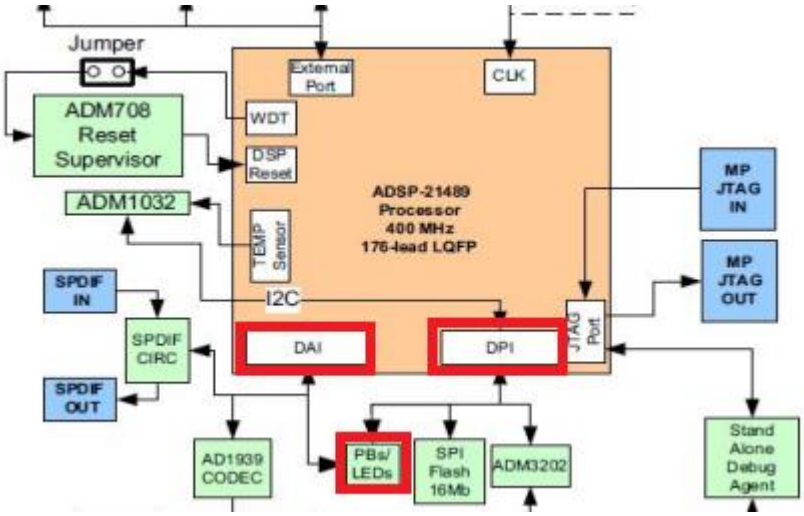
5. DODATNE FUNKCIONALNOSTI

Od predloženih dodatnih funkcionalnosti implementirane su:

- Obrada slika različitih dimenzija
- Vizuelni prikaz segmentacije
- Prikaz progresa obrade korištenjem dioda na ploči

5.1. Prikraz progresa obrade korištenjem dioda na ploči

Za prikaz progresa obrade korišteno je 8 LED dioda dostupnih na ADSP-21489 razvijnoj ploči koje su preko ***DPI*** (prve 3 diode) i ***DAI*** interfejsa (ostalih 5 dioda) povezane sa procesorom, kao što je prikazano na Slici 5.1.



Slika 5.1. Veza procesora i LED dioda

Na početku izvršavanja glavnog programa sve LED diode se isključuju, zatim se uključuje 1. dioda, a onda se nakon završetka svakog od implementiranih algoritama (čitanje slike, konverzija slike u *grayscale*, detekcija ivica, upis segmentirane slike, kodovanje slike, bojenje slike i upis obojene slike) uključuje po 1 LED dioda čime se postiže vizuelni efekat prikaza postotka obrade, slično *loading* baru pri instalaciji programa.

6. LITERATURA

1. ***ADSP-2148 processor***: https://www.analog.com/media/en/dsp-documentation/processor-manuals/ADSP-214xx_HRM_rev0.3.pdf
2. ***CCES C/C++ Compiler Manual for SHARC Processors:*** https://www.analog.com/media/en/dsp-documentation/software-manuals/cces-sharccompiler-manual.pdf
3. ***Loop Optimization:*** https://el.etfbl.net/pluginfile.php/70292/mod_resource/content/1/SharcCompilerLoops.pdf