Elektrotehnički fakultet

Univerzitet u Banjoj Luci

IZVJEŠTAJ PROJEKTNOG ZADATKA

iz predmeta

Ugrađeni računarski sistemi

Tema: Pikselizacija slike

Studenti:

Aleksandar Ritan	1150/14
Milan Medić	1135/16
Dejan Milojica	1150/16

Mentori:

Profesor: prof. dr Zlatko Bundalo

Asistent: mr Miladin Sandić

April 2020

Sadržaj:

1. Uvod

1.1. Tekst projektnog zadatka

Tokom izrade projektnog zadatka bilo je potrebno: Napisati program koji vrši pikselizaciju zadate slike. Program kao argument prima naziv slike koja se nalazi na SD kartici. Pikselizacija se vrši na akciju pomjeranja jednog od prekidača dostupnih na DE1-SoC razvojnom okruženju, nakon čega se pikselizovana slika sačuva takođe na SD karticu. U zavisnosti od rednog broja prekidača, vršiće se pikselizacija blokom određene veličine. Blokovi 4x4, 8x8, 16x16, 32x32 odgovaraju prekidačima SW0, SW1, SW2, i SW3 respektivno. Takođe, po pomjeranju određenog prekidača, treba da zasvijetli vezana LED dioda LED0, LED1, LED2, ili LED3 respektivno, kao i da se na 7SEG displejima ispiše veličina primijenjenog bloka za pikselizaciju. Na DE1-SoC ploči treba prethodno biti podignut Linux OS.

1.2. Uvodna razmatranja

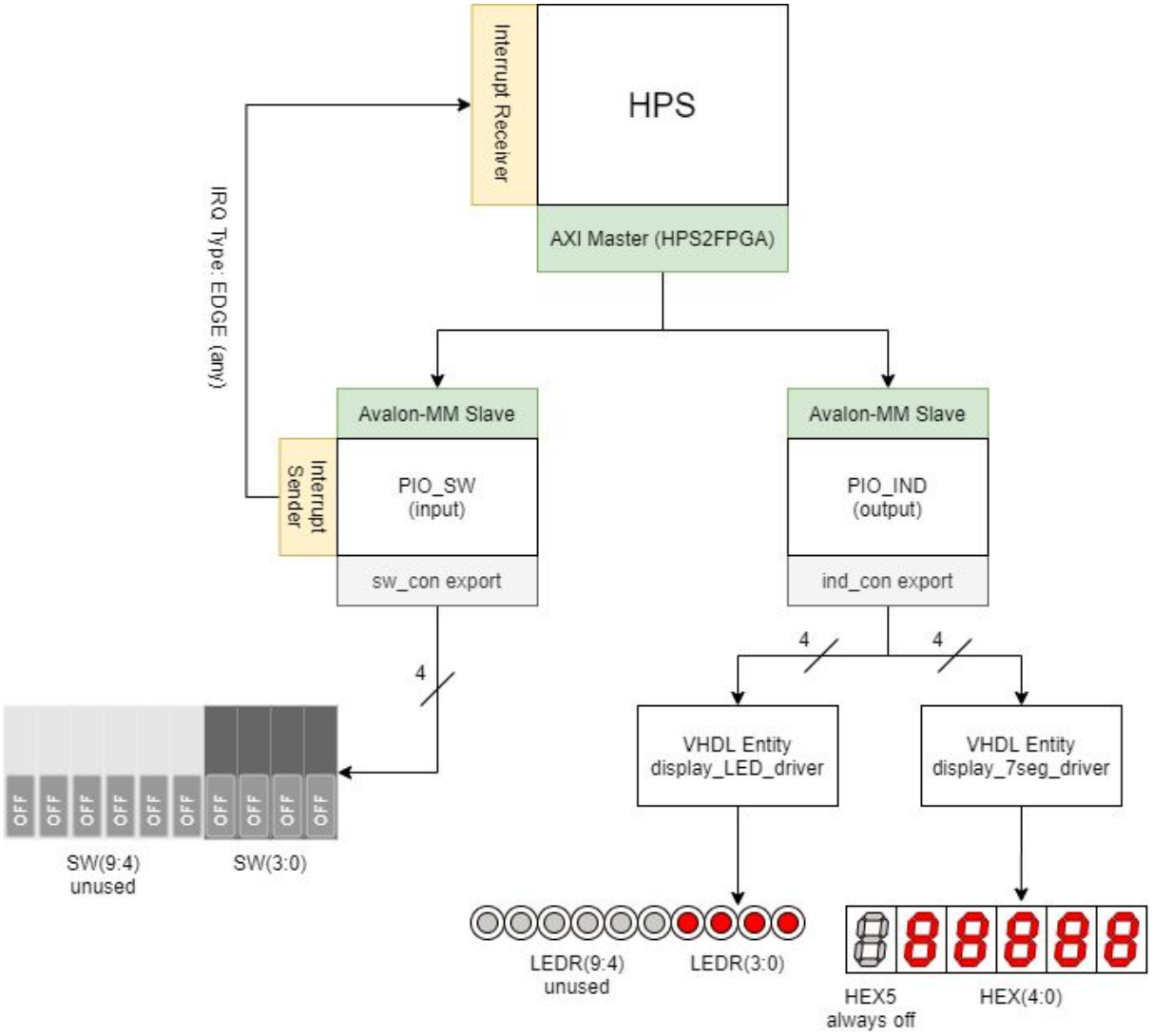
Za izradu projektnog zadatka na raspolaganju nam je bila DE1-SoC platforma, Linux operativni sistem kreiran pomoću Buildroot alata i skripti za kreiranje Embedded Linux OS, te dio izvornog koda potreban za konverziju iz RGB kolor prostora u YUV420 kolor prostor. Resursi koji su iskorišteni na DE1-SoC platformi su Cyclone V FPGA, četiri prekidača (SW3-SW0), četiri LE diode (LED3-LED0) te šest 7-segmentnih displeja.

Da bi se omogućilo lakše upravljanje LE diodama i 7-segmentnim displejima iskorištene su dvije Parallel I/O (PIO) komponente iz Alterinog IP kataloga povezane sa HPS-om putem AXI magistrale kao Avalon MM-Slave uređaji. Jedna PIO komponenta nazvana *“pio_sw”* predstavlja ulaznu komponentu, koja se dovodi na AXI magistralu HPS-a, a sa nje postoji i interrupt linija koja se takođe povezuje na HPS. Pošto je u tekstu projektnog zadatka naglašeno da se pikselizacija slike vrši na pomjeranje prekidača, interrupt se “okida” na obje ivice, nakon čega se unutar C izvornog koda čita vrijednost prekidača i na osnovu nje vrši pikselizacija slike, ali i prosljeđuje četverobitni podatak na drugu PIO komponentu *“pio_ind”*, koja predstavlja izlaznu komponentu, na osnovu kojeg se uključuje odgovarajuća LE dioda i prikazuju odgovarajuće cifre na sedmo-segmentnim displejima. Iako tekstom projektnog zadatka nije naglašeno, implementirano je težinsko očitavanje vrijednosti prekidača, odnosno ukoliko imamo podignuta dva prekidača (npr. SW3 i SW2), nakon očitavanja njihovih vrijednosti na *“pio_ind”* komponentu prosljeđuje se podatak “1000”, odnosno vrijednost sa prekidača SW3.

2. Quartus dizajn

2.1. Blok dijagram

Na slici 1 prikazan je blok dijagram projektovanog sistema visokog nivoa. U narednim poglavljima biće detaljnije opisana uloga svake komponente te način na koji su integrisane u sistem.



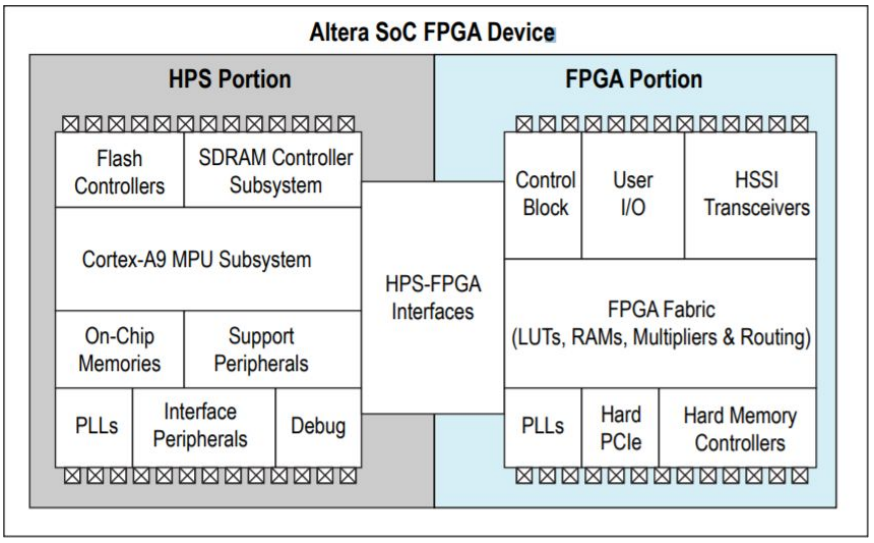
Slika 1. *Blok dijagram sistema*

2.2. IP komponente

Ono što nam nudi Quartus softverski paket i Qsys kao dio paketa jeste jednostavno dizajniranje sistema i mogućnost korištenja već postojećih komponenti u vlastitom dizajnu.

2.2.1. HPS

Cyclone V SE SoC sastoji se od dva izvojena dijela: HPS i FPGA dio. Za razliku od Soft-Core procesora koji se implementira u sklopu FPGA u nekim drugim FPGA varijantama (npr. Cyclone V E), HPS (Hard Processor System) je namjenski projektovana hardverska komponenta koja ma ugrađen ARM Cortex-A9 mikroprocesor. Na slici 2 možemo da vidimo ilustraciju Cyclone V SE i ostalih Altera SoC FPGA varijanti sa HPS dijelom.



Slika 2. *Altera SoC FPGA blok dijagram*

HPS IP komponenta u Qsys-u nalazi se u *Processors and Peripherals -> Hard Processor Systems -> Arria V / Cyclone V Hard Processor System*. Potrebno je da podesimo parametre HPS-a koji odgovaraju modelu komponente na našem uređaju (DE1-SoC). Parametre podešavamo kao u trećem poglavlju iz [3].

2.2.2. PIO (Parallel I/O)

PIO komponenta pruža memorijski mapiran interfejs između Avalon-MM slave port-a i GPIO portova. I/O portovi se konektuju ili na on-chip korisničku logiku (u slučaju PIO_IND), ili na hardverske I/O pinove (u slučaju PIO_SW). Avalon-MM slave port se konektuje na Avalon-MM Master port (ili AXI Master preko Interconnect međusloja), te je na taj način omogućeno memorijski mapirano čitanje registara PIO komponente.Registarska mapa PIO komponente prikazana je na slici ispod.

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	direction (1)		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	interruptmask (1)		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture (1) , (2)		R/W	Edge detection for each input port.				
4	outset		W	Specifies which bit of the output port to set. Outset value is not stored into a physical register in the IP core. Hence it's value is not reserve for future use.				
5	outclear		W	Specifies which output bit to clear. Outclear value is not stored into a physical register in the IP core. Hence it's value is not reserve for future use.				

Note :

1. This register may not exist, depending on the hardware configuration. If a register is not present, reading the register returns an undefined value, and writing the register has no effect.
2. If the option **Enable bit-clearing for edge capture register** is turned off, writing any value to the edgecapture register clears all bits in the register. Otherwise, writing a 1 to a particular bit in the register clears only that bit.

Slika 3. *Registarska mapa PIO komponente*

PIO komponenta može maksimalno da ima širinu od 32 bita. U našem slučaju, obje PIO komponente imaju širinu od 4 bita.

1) PIO_SW (skraćeno od PIO Switch) je ulazna PIO komponenta povezana na četiri prekidača (SW0, SW1, SW2 i SW3) preko eksportovanog *sw_con* interfejsa. S druge strane, povezana je preko Avalon-MM slave port-a na AXI master port HPS-a, te je na taj način moguće memorijski mapirati registre PIO_SW komponente unutar C programa i čitati stanja prekidača. Pored toga, ova PIO komponenta ima omogućene i linije prekida (eng. interrupt), pa je na osnovu sadržaja edgecapture registra moguće zaključiti koji switch je izazvao prekid.
2) PIO_IND (skraćeno od PIO Indicator) je izlazna PIO komponenta na čiji su eksportovani *ind_con* interfejs povezane dvije projektovane VHDL komponente koje su opisane u narednom poglavlju. Takođe, kao i PIO_SW komponenta, preko Avalon-MM slave port-a, PIO_IND je povezan na AXI master port HPS-a, te je na taj način moguće memorijski mapirati registre PIO_SW komponente unutar C programa i upisivati sadržaj u njegov data registar. Sam smisao upisanog podatka objašnjen je u narednom poglavlju.

2.2.3. System ID Peripheral

Kao što i samo ime sugeriše, na osnovu ove komponente cijeli sistem dobija jedinstveni identifikator koji služi za provjeru kompatibilnosti izvršnih fajlova koji se pokreću na sistemu. Ukoliko je izvršni fajl kompajliran za različitu hardversku strukturu, doći će do greške pri pokretanju programa.

2.2.4. JTAG UART

Ova komponenta omogućava serijsku komunikaciju između host računara i DE1-SoC platforme na koju će biti programiran sistem. Na ovaj način, eliminiše se potreba za izdvojenim RS232 interfejsom prema host računaru, te uređaju možemo na jednostavan način pristupiti preko USB konekcije i programa PuTTY.

2.2.5. Clock Source

Ova komponenta se automatski dodaje prilikom kreiranja Qsys sistema. Pomoću ove komponente se čitavom sistemu obezbjeđuje pristup clock signalu.

2.3. Projektovane VHDL komponente

Osim komponenata koje se mogu pronaći unutar Alterinog IP kataloga komponenata bilo je potrebno projektovati dvije dodatne komponente za upravljanje 7-segmentnim displejima i LE diodama, nazvane *display_7seg_driver* i *display_LED_driver* respektivno.

2.3.1. display_7seg_driver

Ova komponenta je implementirana kao VHDL entitet sa jednim ulaznim portom širine 4 bita i šest izlaznih portova širine 7 bita koji se mapiraju na odgovarajuće pinove 7-segmentnih displeja uz pomoć *Pin Planner*-a unutar Quartus Prime razvojnog okruženja.

Način funkcionisanja ove komponente je sljedeći: Podatak širine 4 bita se dovodi na njen ulaz sa *pio_ind* komponente koja ovaj podatak dobija od strane C aplikacije i HPS-a putem Avalon MM interfejsa. Na osnovu vrijednosti primljenog podataka se pobuđuju odgovarajući pinovi 7-segmentnih displeja, odnosno na izlaz komponente se prosljeđuju odgovarajuće vrijednosti sedmobitnih vektora. Vrijednosti izlaznih vektora u zavisnosti od ulaznog su date u tabeli 1, pri čemu posljednja kolona predstavlja ASCII zapis koji se vidi na displejima. Potrebno je napomenuti da se pojedini segmenti displeja pobuđuju niskim naponskim nivoom, odnosno da su *active-low* tipa.

ulazni podatak	HEX5	HEX4	HEX3	HEX2	HEX1	HEX0	ASCII reprezentacija
0000	1111111	1111111	1111111	1111111	1111111	1111111	xxxxxxx[1]
0001	1111111	1111111	1111111	0011001	0111111	0011001	xxx4-4
0010	1111111	1111111	1111111	0000000	0111111	0000000	xxx8-8
0100	1111111	1111001	0000010	0111111	1111001	0000010	x16-16
1000	1111111	0110000	0100100	0111111	0110000	0100100	x32-32

Tabela 1. *Vrijednosti izlaznih vektora*

2.3.2. display_LED_driver

Druga projektovana VHDL komponenta predstavlja drajver za LE diode i takođe je implementirana kao VHDL entitet. Ovaj entitet ima jedan ulazni port širine 4 bita i jedan izlazni port širine 10 bita koji se mapira u *Pin Planner*-u na odgovarajuće pinove LE dioda.

Osnovna funkcionalnost ove komponente je da u zavisnosti od podatka koji se dovede na ulaz (isti podatak koji se dovodi na ulaz *display_7seg_driver*) uključi odgovarajuću LE diodu i time signalizira koji od prekidača je podignut. Logika koja odlučuje koja dioda treba da se uključi je implementirana u dijelu C aplikacije, tako da ova komponenta ima isključivo karakter drajvovanja, odnosno pobuđivanja dioda.

[1] x u ovom slučaju označava ugašen displej

3. Linux i kros-kompajliranje

3.1. Generisanje Linux image-a

Postupak generisanja Linux image-a urađen je na osnovu trećeg i četvrtog poglavlja iz [3] (uz određene modifikacije koje će biti opisane) i sastoji se iz sljedećih koraka:

1) Kompajliranje Quartus dizajna koje rezultuje generisanjem .sof fajla.
2) Konverzija .sof fajla u Raw Binary File (.rbf), potreban za konfigurisanje FPGA dijela.
3) Generisanje preloader-a pomoću *bsp-editor* alata (iz Intel EDS paketa). Preloader se generiše na osnovu podešavanja HPS-a i njegovih periferala, koja smo odradili unutar Quartus projekta. Na taj način preloader može uspješno da inicijalizuje potrebne periferale u toku boot procesa.
4) Generisanje Device Tree Source (.dts) i Device Tree Blob (.dtb) fajlova pomoću *sopc2dts* alata (takođe iz Intel EDS paketa).
5) Kreiranje u-boot.scr skripte pomoću koje će se, u toku u-boot dijela boot procesa, kofigurisati FPGA dio sistema na osnovu prethodno generisanog .rbf fajla.

Fajlovi koji su generisani u prethodnim koracima su: *impix_system.rbf*, *preloader-mkpimage.bin*, *impix_system.dtb*, *impix_system.dts*, te *u-boot.scr*.
Pored navedenih fajlova, za generisanje Linux image-a je potrebno konfigurisati i buildroot alat pomoću kojeg generišemo *zImage* (kompresovani Linux kernel image), *rootfs.tar* (Root File System arhivu) i *u-boot.img* (kompajlirani u-boot bootloader). Verzija korištenog buildroot-a je *buildroot-2017.02-rc1*.
Nakon podešavanja buildroot alata kako je opisano u četvrtom poglavlju iz [3], da bismo na najlakši način mogli pristupiti našem uređaju i jednostavno kopirati fajlove između host računara i uređaja preko SSH protokola, potrebno je modifikovati `/etc/ssh/sshd_config` fajl. Modifikacija je odrađena putem *post_build.sh* skripte koja se automatski pokreće nakon završenog build-a fajl sistema, u sklopu buildroot *make* procesa. Sadržaj skripte izgleda ovako:

```
#!/bin/sh
sed '/PermitRootLogin/ c PermitRootLogin yes' -i "${1}/etc/ssh/sshd_config"
sed '/PasswordAuthentication/ c PasswordAuthentication yes' -i "${1}/etc/ssh/sshd_config"
```

Isječak 1. *Sadržaj post_build.sh fajla*

Ova skripta, korištenjem **sed** alata, modifikuje dvije linije u *sshd_config* fajlu na način se omogući root pristup putem SSH konekcije, kao i autentikacija pristupa putem password-a.

Da bi skripta bila pokrenuta u sklopu *make* procesa buildroot-a, potrebno je dodati putanju do skripte (relativnu u odnosu na korjenski direktorijum buildroot-a, ili apsolutnu) u System configuration -> Scripts to run before creating filesystem images unutar menuconfig-a.

Takođe, izvršena je i modifikacija /etc/network/interfaces fajla, te je podešena statička IP adresa na **eth0** interfejsu na 192.168.10.5, što eliminiše potrebu za DHCP serverom na host računaru, ukoliko se LAN konekcija pravi direktno između računara i DE1-SoC uređaja. Fajl *interfaces* je modifikovan takozvanim rootfs overlay metodom.
Roots overlay metod omogućuje da kreiramo ili prepišemo (eng. overwrite) bilo koji fajl na fajl sistemu uređaja. Ono što je potrebno je da kreiramo stablo direktorijuma do određenog fajla, unutar proizvoljnog foldera, tako da ono izgleda isto kao putanja do tog fajla na fajl sistemu uređaja. U našem slučaju, kreiran je folder rootfs-overlay i unutar njega stablo direktorijuma etc/network/ u koji je smješten modifikovani fajl *interfaces* koji sada izgleda ovako:

```
# interfaces(5) file used by ifup(8) and ifdown(8)

# Configure Loopback
auto lo
iface lo inet loopback

# Configure eth0 with static IP
auto eth0
iface eth0 inet static
address 192.168.10.5
netmask 255.255.255.0
network 192.168.10.0
broadcast 192.168.10.255
gateway 192.168.10.1
```

Isječak 2. *Modifikovani interfaces fajl*

Nakon što je fajl kreiran, potrebno je unutar menuconfig-a dodati putanju do rootfs-overlay foldera (relativnu u odnosu na korjenski direktorijum buildroot-a, ili apsolutnu) u System configuration -> Root filesystem overlay directores.

Prethodne modifikacije su rađene u skladu sa preporukama iz [10], poglavlje 9 (Project-specific customization).
Nakon opisane konfiguracije buildroot-a, potrebno je uraditi *make* koji će generisati *zImage*, *rootfs.tar* i *u-boot.img*.

Konačno, sdcard image je kreiran korištenjem *make_sdimage.py* python skripte, kako je opisano u trećem poglavlju iz [3].

3.2. Kreiranje header fajla i kros-kompajliranje

Po završetku kreiranja *Linux* slike potrebno je kreirati *header* fajl koji sadrži odgovarajuće bazne adrese IP komponenti opisanih u odjeljku 2.2, adresu korištene magistrale i ostale podatke potrebne za uspješnu komunikaciju sa *FPGA* komponentama putem *HPS-a*. Sve komponente korištene u projektu koriste *Avalon-MM* insterfejs, odnosno postavljene su kao *Avalon-MM Slave* uređaji, dok *HPS* ima *Avalon-MM Master*. Komunikacija između *HPS* i *FPGA* dijela je moguća samo preko *AXI Bridge-a*, koji radi prelaz (ponaša se kao spojnica) između *AXI* i *Avalon* interfejsa i time omogućava komunikaciju (*HPS* dio komunicira *AXI* interfejsom, *FPGA* dio komunicira *Avalon* interfejsom). Pri izradi ovog projektnog zadatka korištena je *AXI* magistrala, a svim komponentama u *Qsys-u* su dodjeljene bazne adrese, koje se koriste kao *offset-i* na *AXI* magistrali preko kojih se pristupa pojedinim komponentama. *Header* fajl se kreira uz pomoć *.sopcinfo* fajla koji se dobija pri kompajliranju projekta unutar *Quartus Prime* razvojnog okruženja a za kreiranje header fajla je iskorišten *SOPC builder*. Postupak korišten za kreiranje header fajlova je dat ispod.

1. Kopiramo *<naziv_projekta>.sopcinfo* fajl u direktorijum na putanji <intelFPGA_dir>/<version>/quartus/sopc_builder/bin
2. Zatim se pozicioniramo u direktorijum <intelFPGA_dir>/<version>/embedded i pokrenemo embedded command shell
3. Iz *embedded command shell-a* se pozicioniramo u direktorijum <intelFPGA_dir>/<version>/quartus/sopc_builder/bin
4. Na sistemsku varijablu *PATH* dodamo trenutnu poziciju (trenutnu poziciju možemo dobiti pozivom komande *pwd*)
5. Pokrenemo *sopc-create-header-files* skriptu sa komandom

```
./sopc-create-header-files "./<naziv_projekta>.sopcinfo" --single \
<naziv_header-a>.h --module <naziv_hps_modula>
```

Kada imamo kreiran *header* fajl, potrebno je izvršiti kros-kompajliranje napisane C aplikacije za izvršavanje na DE1-SoC platformi. Najjednostavniji način za to jeste kreiranje *Makefile-a* i njegovo pokretanje sa naredbom *make*.

U ovom projektnom zadatku napravljene su dvije varijante C programa (koje će biti opisane u narednom poglavlju) i svaka od njih ima poseban Makefile. Primjer *Makefile-a* iskorištenog u ovom projektnom zadatku (za pixelization_rt varijantu C programa) prikazan je u isječku 3. Pokretanjem ovog *Makefile-a* dobija se binarni fajl koji se potom prebacuje na DE1-SoC platformu naredbom *scp* i tu po potrebi pokreće.

```
TARGET = pixelization_rt

ALT_DEVICE_FAMILY ?= soc_cv_av
SOCEDS_ROOT ?= $(SOCEDS_DEST_ROOT)
HWLIBS_ROOT = $(SOCEDS_ROOT)/ip/altera/hps/altera_hps/hwlib
CROSS_COMPILE = arm-linux-gnueabihf-
CFLAGS = -g -Wall -lpthread -D$(ALT_DEVICE_FAMILY)
-I$(HWLIBS_ROOT)/include/$(ALT_DEVICE_FAMILY) -I$(HWLIBS_ROOT)/include
LDFLAGS =  -g -Wall -lpthread
CC = $(CROSS_COMPILE)gcc
ARCH = arm

build: $(TARGET)

$(TARGET): main.o
	$(CC) $(LDFLAGS) $^ -o $@
%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
	rm -f $(TARGET) *.a *.o *~
```

Isječak 3. *Makefile za kros-kompajliranje pixelization_rt programa*

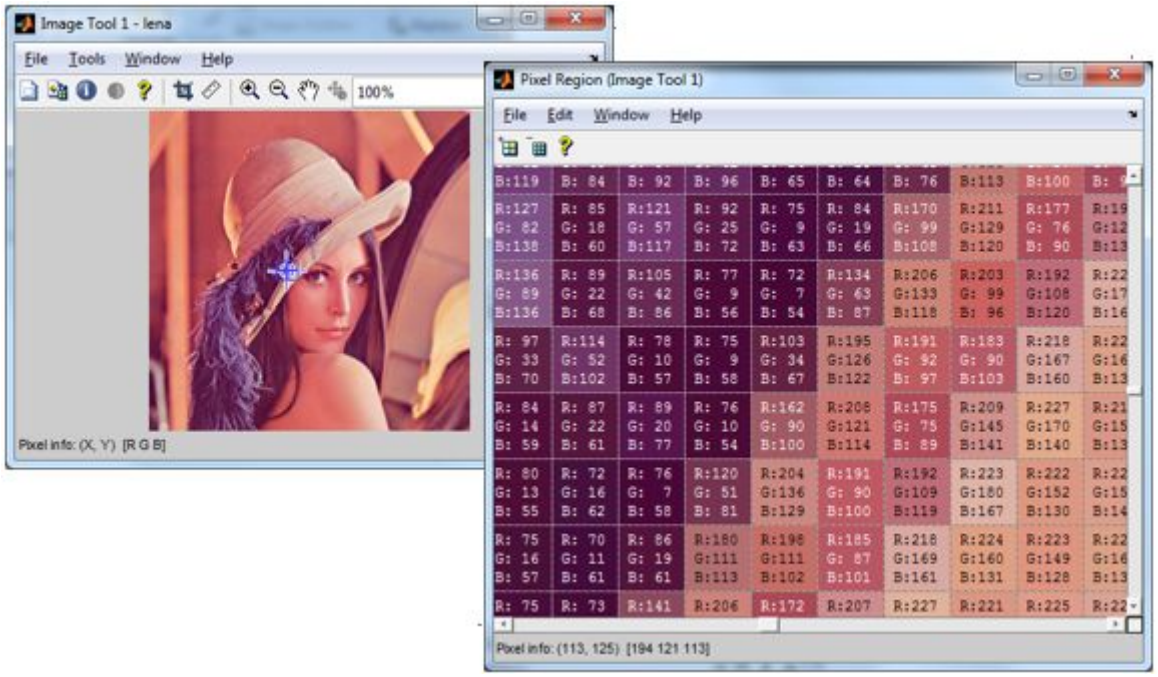
4. C aplikacija i rezultati

4.1. C aplikacija

4.1.1. RGB kolor prostor

Termin *RGB*, potiče od prvih slova engleskih riječi, kojima se označavaju crvena, zelena i plava boja, odnosno Red, Green i Blue. Ovakav sistem se bazira na na tome da se svaki piksel određuje sa ukupno tri informacije koje ga opisuju, a svaka od njih predstavlja pomenute tri boje, koje su ujedno i osnovne (gradivne). Često za ovakav sistem, kažemo da je riječ o takozvanom *aditivnom* sistemu mješanja boja, kojim se zapravo kombinuju svjetlosti, odnosno intenziteti svjetlosti svake od pomenutih boja. Primjenom pravila ovog sistema, odnosno kombinovanjem intenziteta svjetlosti svake od navedenih komponenti na različite načine dobijaju se sve moguće boje iz ovog kolor prostora (16777216 boja). Za dobijanje određene boje, kao što je pomenuto, neophodno je kombinovati u određenoj mjeri osnovne boje, što se posmatra na procentualnoj skali, odnosno koliki udio dobijene boje, jedna od osnovnih komponenti ima. Procentualno, svaka osnovna komponenta se kreće u rasponu od 0 do 100, dok bi u računarskom sistemu, to bilo predstavljeno jednim bajtom, odnosno dekarno od 0 do 255. Na taj

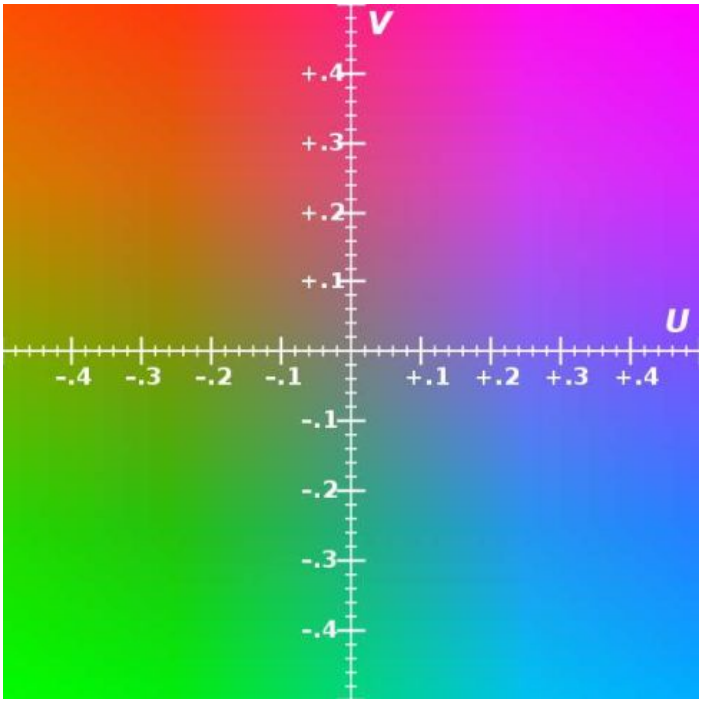
način, svaki piksel novoformirane slike, karakteriše 3 bajta(24bita), koji reprezentuju osnovne boje, i tada kažemo da se radi o RGB 24. Pored prethodnog, neki od značajnijih formata su RGB 8, 16 i 32.



Slika 4. *RGB prostor boja, primjer jednog bloka piksela*

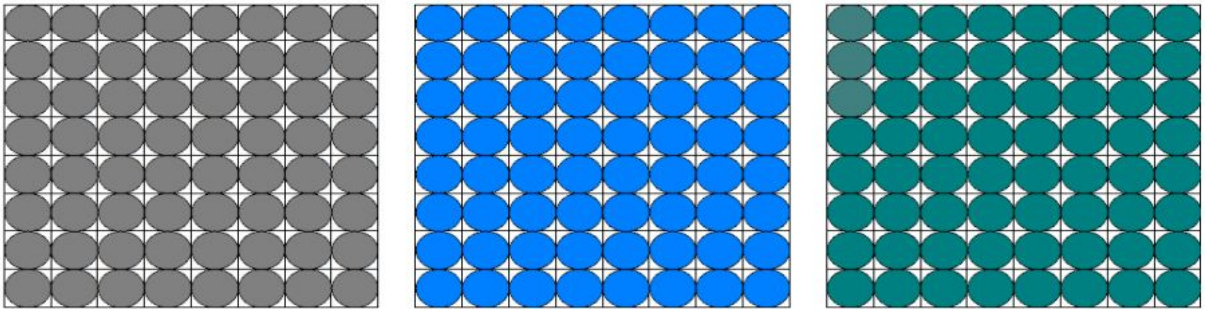
4.1.2. YUV kolor prostor

Jedan od standardnih prostora boja, koji nam omogućava kompresiju slike, kao i video materijala. Komprimuje sliku u boji, ili video zapise uzimajući u obzir ljudsku percepciju, omogućavajući smanjenu širinu opsega za komponente hrominanse(U I V), što obično omogućava da greške koje se ostvaruju prilikom prenosa sadržaja, ili artefakti ostvareni pri kompresiji, budu efikasnije “maskirani” tj. njihov uticaj bude smanjen za ljudsku percepciju, u odnosu korišćenja "direktne" RGB reprezentacije. Ostale kompresije u boji, imaju slična svojstva, a glavni razlog za implementaciju ili istraživanje svojstava YUV-a bi bio povezivanje s analognom ili digitalnom televizijskom i fotografskom opremom koja je u skladu s određenim YUV standardima. Dobija se iz RGB prostora boja, izvođenjem određenih računskih operacija nad datim RGB pikselima. Ovakav vid formata, se bazira na na tri komponente, Y, U i V komponente, odnosno, jedne komponente lume (Y) i dvije komponente za kromiranje, nazvane U (plava projekcija) i V (crvena projekcija), a sama kompresija se bazira na redukciji U i V komponenti, odnosno smanjenju njihovog zauzeća. Definišemo nekoliko formata, od kojih su najznačajniji: YUV 440, YUV 422 i YUV 420.



Slika 5. *YUV prostor boja*

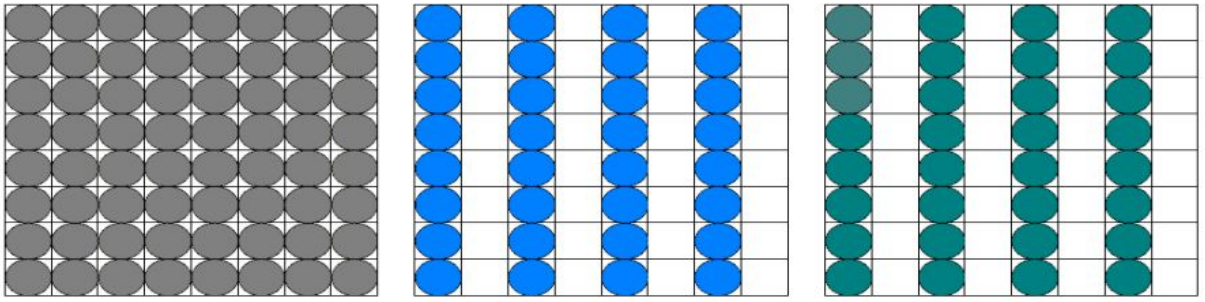
Osnovna razlika između prethodno pomenutih formata, se ogleda u samom zapisu, odnosno nivou kompresije. U skladu sa tim, YUV 444 zapis, se zasniva na postojanju tri komponente ,Y U i V, pri čemu svaka tačka, odnosno piksel, ima svoju odgovarajuću, jedinstvenu komponentu. Ako posmatramo na nivou memorije, pri čemu svaka od komponenata zauzima jedan bajt, tada će nam za svaki piksel, biti neophodno ukupno 3 bajta. Kako smo kod RGB24 formata, takođe koristili 3 bajta, na ovaj način se ne ostvaruje smanjenje memorijskog zauzeća, čemu zapravo kompresijom i težimo. Ukoliko imamo primjer bloka piksela 8x8, tada će sve tri komponente, biti veličine 8x8, kao što je prikazano na slici 2.3.



Slika 6. *Primjer YUV 444 formata*

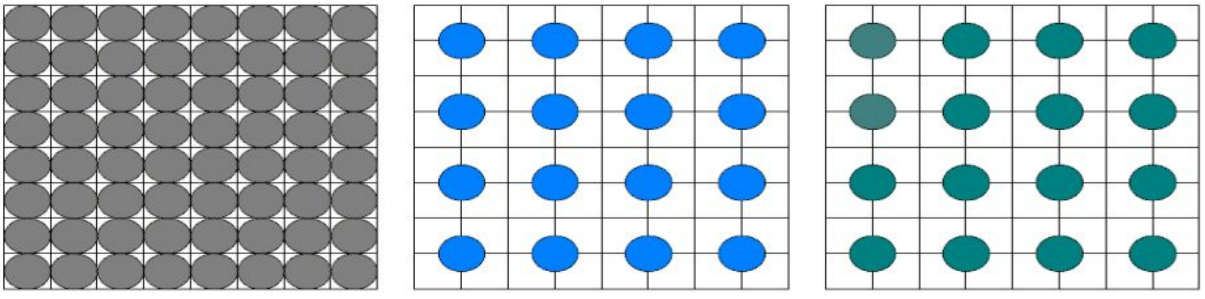
Kao što smo vidjeli, YUV 444, iako spada u grupu YUV formata, ne ostvaruje smanjenje memorijskog zauzeća, što je cilj kod kompresije slike/videa. U skladu sa tim, posmatraćemo YUV 442 kolor prostor. YUV 442 se bazira na tome da svaki piksel, posjeduje „svoju“ jedinstvenu Y komponentu, dok dva susjedna piksela dijele, odnosno

imaju zajedničke U i V komponente. Na taj način, ukoliko imamo primjer bloka 8x8 piksela, ukupan broj Y komponenti, odnosno veličina cjelokupne Y komponente, će biti jednaka veličini RGB slike, dok će veličina cjelokupnih U I V komponenti, biti, u ovom slučaju 8x4, jer kao što je rečeno dva piksela u redu imaju zajedničku komponentu, što je prikazano na slici 2.4, te se na ovaj način ostvaruje kompresija u smislu memorijskog zauzeća, jer će U i V zapisi biti značajno manji, što se pokazuje jako korisno kod slika sa ekstremno velikim brojem piksela. Ako posmatramo iz memorijskog aspekta, za svaki piksel se koristi jedan bajt Y komponente, te kako dva piksela dijele po jednu u i v komponentu, možemo teorijski smatrati da svaki od njih zauzima po pola bajta zasebne komponente, što kao rezultat daje da jedan piksel zauzima 2 bajta, što je manje od 3 bajta koje smo imali kod YUV 444.



Slika 7. *YUV 442 format*

Posljednji, i ujedno najčešće korišćen YUV format, je YUV 420 format. Ovaj format ostvaruje najviši stepen kompresije, što se bazira na dodatnoj redukciji veličine U i V komponenti. Osnova ovog formata je da i dalje svaki piksel karakteriše odgovarajuća zasebna Y komponenta dok 4 susjedna piksela, što se posmatra kao blok piksela 2x2, dijele po jednu U i V komponentu kao što je prikazano na slici 2.5.



Slika 8. *YUV 420 format*

Posmatrajući iz memorijskog aspekta, na nivou piksela, imamo jedan bajt zauzeća Y komponente, te po jednu četvrtinu bajta (teorijski razmatrano) U i V komponente, što je kao rezultat jedan i po bajt. Naravno ova cifra reprezentuje prosjek i kao takva može se mijenjati u zavisnosti od dimenzija slike (da se može podijeliti na blokove 2x2), čime se ostvaruje prethodno opisana kompresija. U slučaju da dimenzije nisu djeljive sa dva, dobićemo krajnje/rubne piksele koji će biti primorani posjedovati

sopstveni, nedjeljivi bajt U i V komponenti ili eventualno dva piksela koja dijele istu U i V komponentu. Bez obzira na probleme sa krajnjim/rubnim pikselika, ostvaruje se značajno memorijsko smanjenje, što izraženije kod slika sa velikim brojem piksela.

4.1.3. Konverzija u/iz RGB kolor prostora

Kao što je već pomenuto, YUV reprezentacija se generiše iz RGB reprezentacije, primjenom odgovarajućih matematičkih operacija. Jedan od standardnih pristupa je taj da se vrijednosti R, G i B sumiraju, da bi se proizvela Y komponenta, što je zapravo mjera ukupne svjetline ili osvetljenja. Komponente U i V se računaju kao skalirane razlike između vrednosti Y, B i R. Prema ***ITU-R BT.601*** standardu, koji se odnosi na kopresiju u digitalnoj televiziji za konverziju se koriste matematički izrazi dati Slikom 2.x:

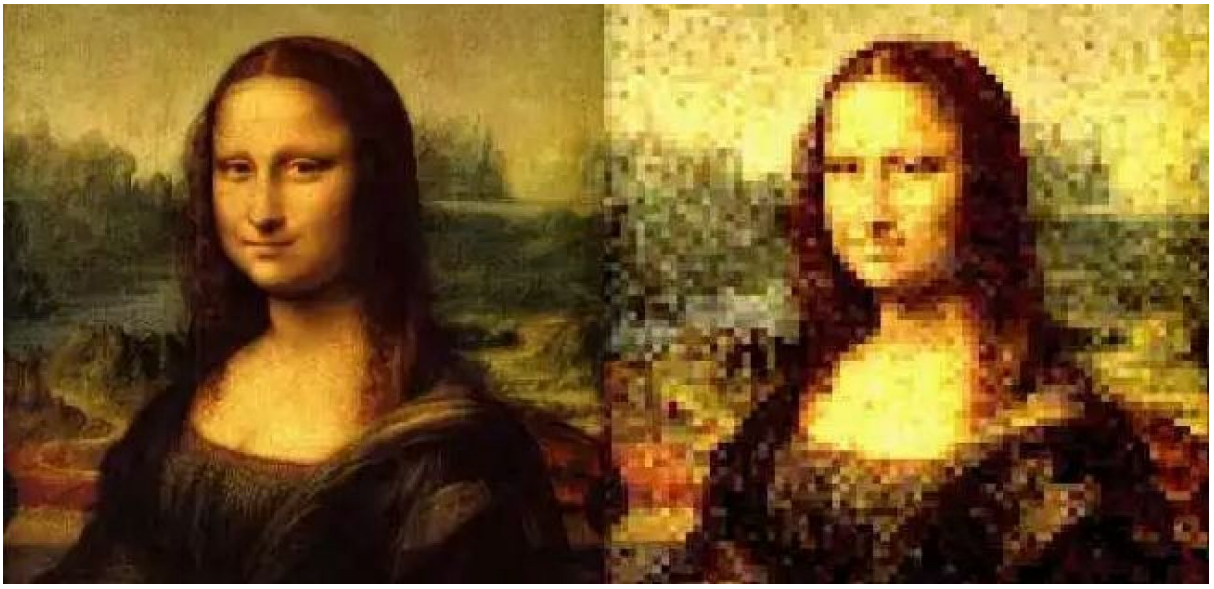
Slika 9. *Matematički zapisi konverzije RGB : YUV*

Prethodni način konverzije se koristi u digitalnoj televiziji, te nije način koji se direktno implementira kod konverzije na nivou RGB – YUV420, koji je iskorišten u projektnom zadatku. U skladu sa tim za konverziju iz RGB u YUV 420 kolor prosto je iskorištena funkcija *void convert_rgb_to_yuv420(),* koja prolazi kroz niz koji predstavlja RGB zapis učitane slike, pri čemu su komponente jednog piksela smještene jedna za drugom. Nad tim nizom koji za sada ne čuva prostornu informaciju slike generiše se matrica dimenzija 16x48 u koju smiještaju RGB komponente sa prostornom informacijom, te se tako formirana matrica koristi za formiranje nova tri niza, koji predstavljaju privremene bafere YUV zapisa. Za to se koristi funkcija *void rgb_to_yuv420(uint8_t rgb[16][48], uint8_t y[16][16], uint8_t u[8][8], uint8_t v[8][8]),* koja prolazeći kroz primljenu RGB matricu, vrši matematičke operacije, te popunjava privremene bafere YUV formata. Privremeni baferi dobijeni pozivom funckije *rgb_to_yuv420* se upisuju na odgovarajuća mjesta matrica koje predstavljaju Y, U i V komponente. Ovime imamo na raspolaganju zapis slike u YUV formatu i gdje možemo vršiti razne manipulacije sa slikom. Povratak u RGB format, se ostvaruje inverzno prethodnom načinu, kroz funkciju *void*

convert_yuv420_to_rgb(), koja na osnovu matrica YUV formata, kreira privremene bafere, te adekvatno popunjava RGB matricu.

4.1.4. Pikselizacija

Pikselizacija je postupak kod kog se iz ulazne slike uzima blok piksela, vrijednosti piksela u bloku se usrednjavaju i tako usrednjene vrijednosti dodjeljuju svim pikselima u istom bloku. Na taj način dolazi do smanjenja kvalitete slike, što i jeste osnovni cilj ovog postupka. U skladu sa mogućnostima koja nam pikselizacija pruža, jedna od primarnih upotreba je cenzura.



Slika 10. *Primjer pikselizacije (original i pikselizovana slika)*

Ono što se dodatno može uraditi je pikselizacija samo određenih dijelova slike, što nam omogućava višestruku primjenu, naročito kod prikazivanja sadržaja sa željom skrivanja samo određenih djelova, kao što je prikazano na sljedećoj slici.

Slika 11. *Prikaz djelimične pikselizacije, čime se vrši cenzura određenog dijela slike*

Kao što je pomenuto, algoritam koji obavlja pikselizaciju ima za cilj da smanji kvalitet piksela, odnosno da "zamuti" sliku, tako što se kvadrat od NxN piksela popunjava srednjom vrednošću piksela iz istog kvadrata. Veličina N definiše stepen, odnosno nivo zamućenja, jer što nam je blok za pikselizaciju veći, to je i zamućenost slike veća, što je rezultat usrenjavanja vrijednosti većeg broja piksela.
U praktičnom smislu, bez obzira da li vršimo potpunu ili djelimičnu pikselizaciju, pokazuje se da je u većini slučajeva, sa stanovišta skrivanja informacija, dovoljna pikselizacija sa N od 32 ili 64, jer se vrši usrednjavanje na nivou 1024, odnosno 4096 piksela.
Pored toga što se pikselizacija primjenjuje nad slikom predstavljenom u RGB formatu, česta je situacija primjene pikselizacije u kompresovanom obliku, što je u našem slučaju YUV 420. Osnovna razlika izvršavanja algoritma se ogleda u tome da u RGB kolor prostoru radimo pikselizaciju nad sve tri komponente, dok u YUV formatu data transformacija se primenjuje samo na Y komponentu. Na taj način se ostvaruju prednosti kao što su veća efikasnost izvršavanja, manje memorijsko zauzeće i slično, jer kao što je rečeno, pikselizacija se vrši samo nad jednom komponentom, za razliku od RGB, gdje to radimo nad tri komponente.
Izvršavanje pikselizacije se obavlja u funkciji *perform_filtering*, na način da se prolazi kroz Y komponentu date slike, uzima blok veličine NxN i nad njim radi usrednjavanje. Dati program je prilagođen za izvršavanje nad veličinama 4, 8, 16 i 32, što se ostvaruje kroz aktiviranje jednog od prekidača DE1-SoC uređaja, pri čemu najviši prioritet ima prekidač na najvećoj poziciji (SW3 -> SW0). Za učitavanje slike, koristi se f-ja *read_bmp,* te se prilikom učitavanja slike iz memorije prvo učitava BMP header date slike, što je u našem slučaju reprezentovano strukturom *BITMAPFILEHEADER*, koja sadrži osnovne podatke o samoj slici, kao što su dimenzije, tip slike, zauzeće i slično. Dodatno se zahtijeva da dimenzije slike budu množilac broja 16, čime se ostvaruje sigurna pikselizacija nad 4, 8 i 16 veličinama bloka. Što se tiče bloka 32x32, tu se javlja određen vid odstupanja, jer nemamo uvodnu provjeru, tako da se ta provjera zahtijeva unutar f-je pikselizacije. Ukoliko su dimenzije takve da nemamo djeljivost sa 32, algoritam se izvršava do onih granica gdje je moguće uzeti blok 32x32, dok se preostali dio slike pikselizuje sa blokovima manje veličine. Nakon izvršavanja pikselizacije vrši se povratak u RGB kolor prostor, te se tako pikselizovana slika binarno upisuje (zajedno sa *BITMAPFILEHEADER)* u definisanu datoteku. Potrebno je pomenuti i to da se naziv ulazne/izlazne slike navode kao argumenti komandne linije, prilikom pokretanja programa i to:

- argv[0] = Naziv programa koji se izvršava,
- argv[1] = Naziv slike nad kojom se vrši pikselizacija,
- argv[2] = Naziv fajla u koji se smiješta pikselizovana slika.

perform_filtering funkcija prikazana je na slici ispod.

```
void perform_filtering()
{
    int32_t i, j;
    int32_t average = 0;
    int32_t vertical_offset,horizontal_offset;
    int32_t overhead; // Za rubne piksele gdje mozda nije 32x32 blok
    for (vertical_offset = 0; vertical_offset < (YSIZE+16);vertical_offset+=N ) {
        for (horizontal_offset = 0;horizontal_offset < (XSIZE+16); horizontal_offset+=N ) {
            average   = 0;
            overhead  = 0;
            for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                    if ((XSIZE+16)*(YSIZE+16)>(XSIZE + 16)*(i + vertical_offset)+ j+horizontal_offset)
                        average +=y_pic[(XSIZE + 16)*(i + vertical_offset)+ j+horizontal_offset];
                    else
                        overhead++;
                }
            }
            for (i = 0; i < N; i++) {
                for (j = 0; j < N; j++) {
                    if ((XSIZE+16)*(YSIZE+16)>(XSIZE + 16)*(i + vertical_offset)+ j+horizontal_offset) {
                        if (overhead)
                            y_pic[(XSIZE + 16)*(i + vertical_offset)+ j+horizontal_offset] = average/overhead;
                        else
                            y_pic[(XSIZE + 16)*(i + vertical_offset)+ j+horizontal_offset] = average/(N*N);
                    }
                }
            }
        }
    }
}
```

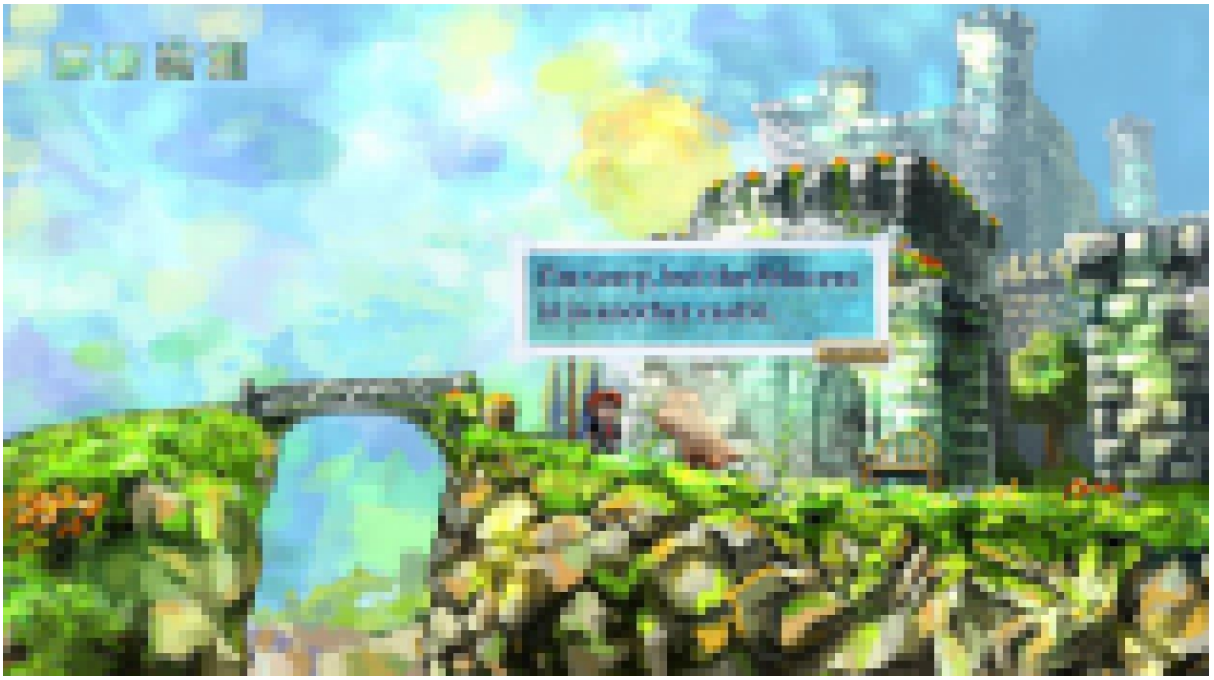
Slika 12. *Funkcija koja izvršava pikselizaciju nad Y komponentom slike*

4.2. Rezultati

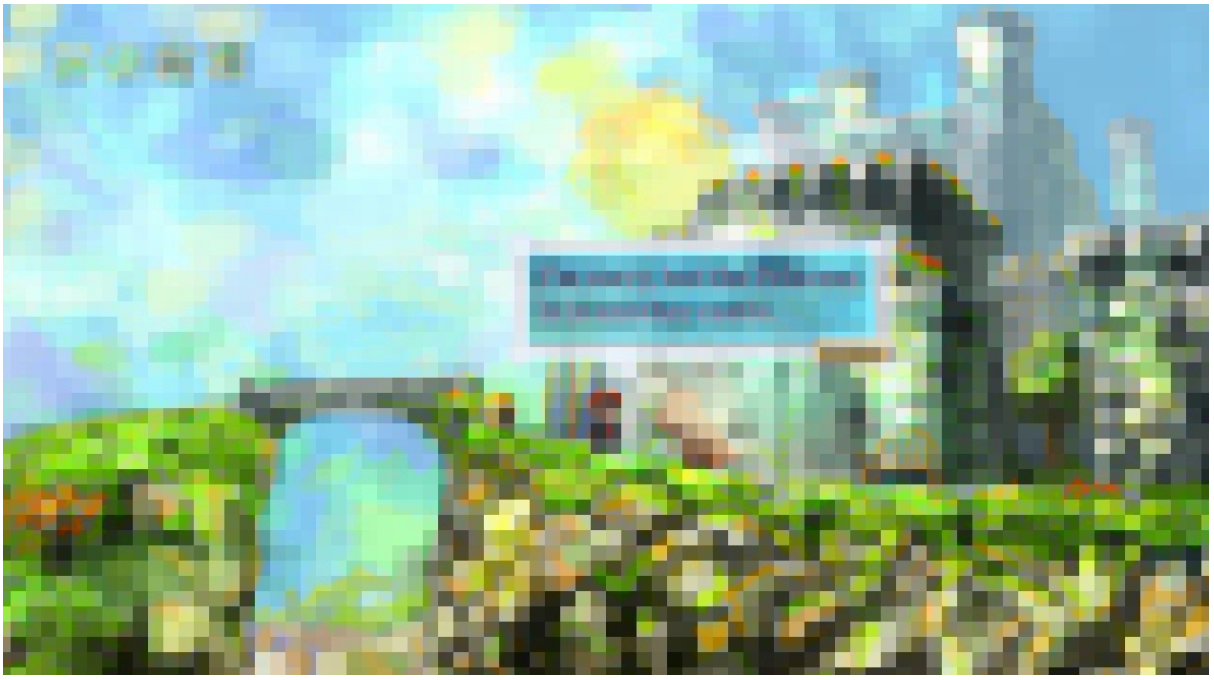
Slika nad kojom se vrši pikselizacija, kao i pikselizovane slike prikazane su ispod.

Slika 13. *Slika nad kojom se vrši pikselizacija (Braid.bmp)*

Slika 14. *Slika pikselizovana sa veličinom bloka 4x4*



Slika 15. *Slika pikselizovana sa veličinom bloka 8x8*



Slika 16. *Slika pikselizovana sa veličinom bloka 16x16*

Slika 17. *Slika pikselizovana sa veličinom bloka 32x32*

5. Literatura

[1] ***DE1-SoC User Manual:***
http://www.ee.ic.ac.uk/pcheung/teaching/ee2_digital/DE1-SoC_User_manual.pdf

[2] ***Cyclone V Hard Processor System - Technical Reference Guide:***
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_54001.pdf

[3] ***Embedded Systems - Using Quartus and Buildroot for building Embedded Linux Systems:*** ***http://oa.upm.es/45352/1/DE1-SoC_Embedded_Linux_Systems_1_9.pdf***

[4] ***Parallel Input/Output (PIO) and Interrupt:***
https://class.ece.uw.edu/469/peckol/doc/Tutorials/05_PIO_and_Interrupt_student.pdf

[5] ***How to configure Linux to receive IRQs from the FPGA? A step-by-step guide for Altera Cyclone V SoC:***
https://wiki.epfl.ch/prsoc/documents/Cyclone_V_SoC_Linux_Interrupt-2.pdf

[6] ***Mouse show - Damjan Prerad, Slaven Smiljanić:***
https://github.com/embeddme/Mouse_show

[7] ***FPGA designs with VHDL[2]:*** **https://vhdlguide.readthedocs.io/en/latest/**

[8] ***FPGA USB Altera blaster driver installed in Linux[3]:***
https://www.youtube.com/watch?v=hEi8nqpQv60

[9] ***YUV color space:*** ***https://en.wikipedia.org/wiki/YUV***

[10] ***Buildroot User Manual:*** ***https://buildroot.org/downloads/manual/manual.html***

[2] Najbolji sajt za uvod u VHDL programiranje
[3] Potrebno da bi se DE1-SoC mogao programirati iz Quartus Prime softvera na Linux OS