FPGA implementacija Huffmanovog dekodiranja JPEG dekodera

Fodor, Fran

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1635

FPGA IMPLEMENTACIJA HUFFMANOVOG DEKODIRANJA JPEG DEKODERA

Fran Fodor

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1635

FPGA IMPLEMENTACIJA HUFFMANOVOG DEKODIRANJA JPEG DEKODERA

Fran Fodor

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1635

Pristupnik:	**Fran Fodor (0036543140)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	izv. prof. dr. sc. Daniel Hofman

Zadatak: **FPGA implementacija Huffmanovog dekodiranja JPEG dekodera**

Opis zadatka:

Kodiranje slike u format JPEG moguće je ubrzati korištenjem sklopova FPGA. Potrebno je proučiti rad algoritma JPEG. Posebnu pozornost potrebno je usmjeriti na Huffmanovo dekodiranje. Implementirati dijelove JPEG dekodera i istestirati cijeli dekoder na testnim slikama. Testirati rad pojedinih dijelova kodera na testnim podatcima. Usporediti rad dekodiranja na procesoru i na hardverskoj implementaciji. Napraviti dokumentaciju i objaviti programski kôd na repozitoriju Git.

Rok za predaju rada: 14. lipnja 2024.

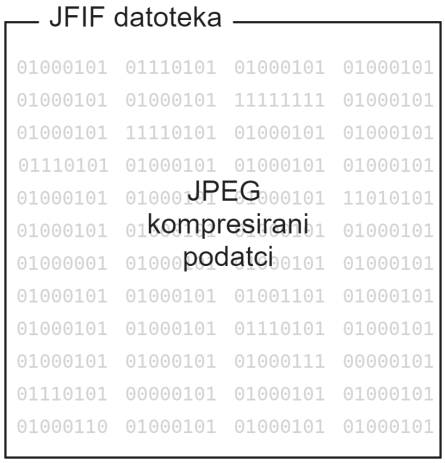
Sadržaj

1. Uvod

JPEG format ima široku primjenu danas na web stranicama budući da omogućuje znatno smanjenje slike bez gubitatka bitnih detalja. Za prikaz slike potrebno ju je dekodirati, stoga u ovom radu je razmatran FPGA akcelerator za dekodiranje JPEG slike. Korištena je PYNQ-Z1 pločica, a sam algoritam pisan je u jeziku C++, te sintetiziran za FPGA pomoću Vitis HLS-a.

1.1. JPEG

JPEG (engl. Joint Photographic Experts Group) nije format slike, već specifikacija (algoritam) za kompresiju slike s gubitcima nastala 1992. godine. Obično kada pričamo o JPEG-u mislimo na JFIF (engl. JPEG File Interchange Format). On služi kao omot (engl. wrapper) za podatke nastale JPEG algoritmom (slika 1.1.). No, radi lakšeg sporazumijevanja, koristit ću kraticu JPEG za format slike.



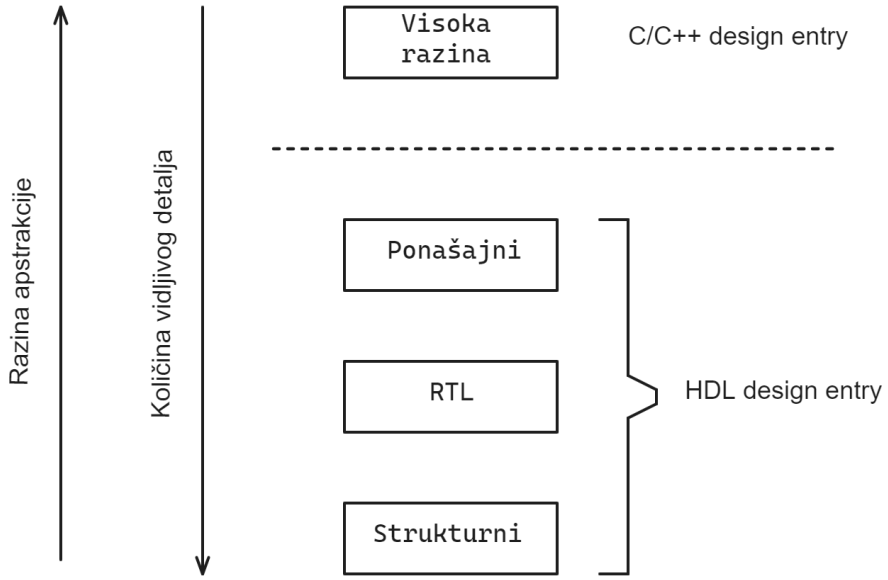
Slika 1.1. Slikovni prikaz odnosa između JFIF i JPEG [1]

Kompresija s gubitcima zamišljena je tako da eliminira nepotrebno sa slike, odnosno,

detalje koje ljudsko oko ne vidi. Naime, ljudsko oko ima značajno više receptora koji služe za raspoznavanje **svjetline** u odnosu na receptore koji služe za raspoznavanje **boja**. Sama svrha JPEG-a je da zadrži skoro sve detalje o svjetlini slike dok se detalji boja mogu znatno smanjiti kako bi se kompresirala slika. Više o kompresiji i samom radu govorit ću kasnije.

1.2. Vitis HLS

Vitis HLS (engl. High Level Synthesis) je alat koji korisnicima omogućuje pisanje kompleksnih algoritama za **FPGA pločice** direktno u C/C++, umjeo standardnog VHDL-a, Veriloga i sl. Radi na principu da prevoditelj C/C++ kod prevodi (sintetizira) u VHDL.



Slika 1.2. Slikovni prikaz razina apstakcije u FPGA dizajnu [2]

Strukturni model je najmanja razine apstrakcije u kojem se eksplicitno pozivaju, konfiguriraju i povezuju svi hardver elementi koji čine zadani dizajn. Češće korišteni model je **fizički ostvariv model** (engl. RTL Register Transfer Level) koji skriva detalje tehnologije (dakle, prenosiviji kod) ali se i dalje radi s registrima i operacijama između njih. **HLS** prevodi C/C++ kod upravo u **RTL** model.

Na kraju, dizajn s najvećom razinom apstrakcije je **ponašajni model** u kojem, umjesto da se ručno pišu operacije među registrima, algoritamski se opisuje kako će se sklop ponašati [3].

1.3. FPGA sklop

FPGA (engl. Field-programmable gate array) je vrsta integriranog sklopa koji se može reprogramirati. Sastoje se od velikog broja **programabilnih logičkih blokova**, tzv. LUT-ova (engl. Look Up Tables) i međuspojeva koji se mogu konfigurirati za izvođenje raznih zadataka. Programiraju se pomoću jezika za opis sklopovlja (VHDL, Verilog i sl.).

FPGA pločica koju ću koristiti u ovom radu je **PYNQ-Z1**. Najveća prednost ove pločice je ARM A9 procesor na kojem se vrti Linux s Jupyter Notebook-om okolinom što omogućava vrlo laku interakciju s pločicom (slanje podataka i bitstream-a). Više u poglavlju 4.

1.3.1. Postupak razvoja FPGA

Postupak razvoja (ostvarivanja željense funkcionalnosti) sastoji se od 5 koraka [3].

1. **Razvoj HDL modela** gdje se u nekom programu (simulatoru) programira funkcionalnost pločice, a rezultat je RTL model koji opisuje funkciju sklopa.

2. **Logička sinteza** u kojem se RTL model pretvara u listu povezanih standardnih logičkih sklopova te nastaje "gate-level" lista.

3. **Mapiranje gate-level liste na konkretno sklopovlje**.

4. **Razmještavanje elemenata na pločicu** što podrazumijeva optimiranje duljine vodova, utrošaka površine itd.

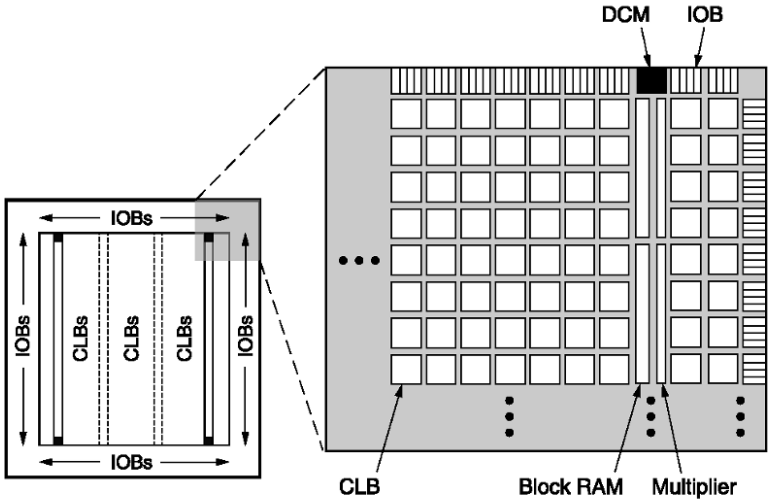
5. **Fizičko povezivanje elemenata liste žicama**. Zadnji korak nam daje datoteku s konfiguracijom sklopa koju šaljemo na FPGA pločicu (engl. **bitstream**).

Ovaj postupak generalno provodi alat u kojem razvijamo pločicu i rijetko se radi ručno.

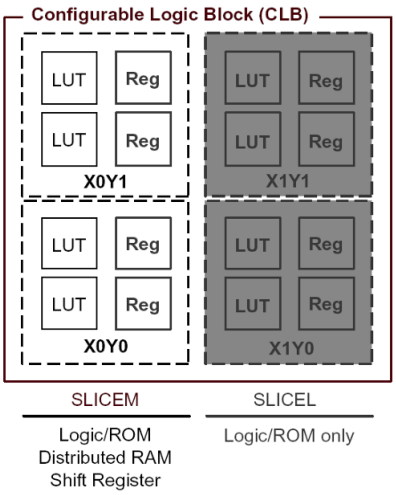
1.3.2. Građa FPGA sklopa

Slika 1.3. prikazuje osnovnu građu FPGA sklopa. Glavni gradivni elementi su konfigurabilni logički blok (CLB) i ulazno-izlazni blok (IOB).

CLB blokovi 1.4. se programiraju da ostvaruju zadane funkcionalnosti pisane u VHDL-u, a IOB služe za komunikaciju okoline i pločice.



Slika 1.3. Raspored osnovnih elemenata na integriranom krugu [3]



Slika 1.4. Blok shema CLB-a [3]

2. JPEG algoritam

JPEG algoritam se sastoji od pet glavna koraka:

- pretvorba prostora boja
- poduzorkovanje
- diskretna kosinusna transformacija
- kvantizacija
- huffmanvo kodiranje

2.1. Pretvorba prostora boja

U prikazu rada prvog koraka, kao pomoć, koristit ću sljedeću sliku:

Slika 2.1. Pomoćna slika

Svaka slika sastoji se od piksela, a svaki ima tri komponente - crvena, zelena i plava (engl. RGB). Svaka komponenta može imati vrijednosti od 0 do 255 i kombinacijom te tri komponente dobije se bilo koja boja. To je pogodno za digitalni prikaz, no jako loše za JPEG algoritam koji pokušava iskoristiti nedostatke ljudskog oka koji se u RGB formatu ne mogu iskoristiti. Iz tog razloga se prelazi na drugačiji prostor boja tzv. **YCbCr** (engl. Y - luminance, Cb - blue chrominance, Cr - red chrominance). Pomoću njega se vrlo efikasno kompresira slika jer su **razdvojeni kanali** koje ljudsko oko dobro vidi i one koje ne vidi.

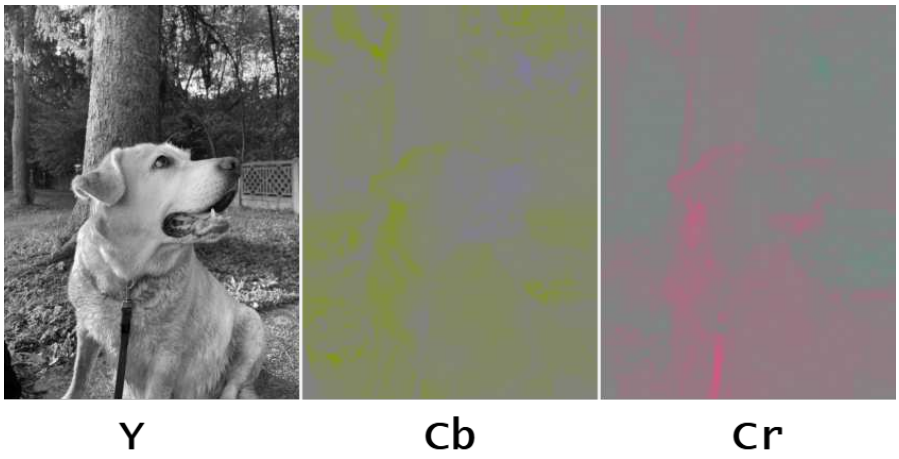
Dodatno, pretvorba je vrlo jednostavna i dvosmjerna:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B \tag{2.1}$$

$$Cb = -0.1687 * R + 0.3313 * G + 0.5 * B + 128 \tag{2.2}$$

$$Cr = 0.5 * R - 0.4187 * G - 0.0813 * B + 128 \tag{2.3}$$

Ovako to izgleda na primjeru slike:



Slika 2.2. Prikaz rastava orginalne slike na Y, Cb i Cr kanal [4]

Kao što je vidljivo iz Cb i Cr kanala, vrlo se malo može zaključiti o slici. Stoga ti detalji nisu toliko bitni i mogu se **smanjiti**.

2.2. Poduzorkovanje (engl. downsampling)

Ovaj korak je opcionalan, ali se gotovo uvijek koristi. Cilj mu je izbaciti podatke iz **Cb i Cr kanala** budući da ih ljudsko oko loše percipira. Po standardu, moguće je izbaciti pola ili četvrtinu informacija. "Izbacivanje" se radi na principu **spajanja** dva (ili četiri) susjedna piksela, **izračuna se srednja vrijednost** svih piksela i smanji slika tako da **više spojenih piksela zauzimaju jedan piksel**. Kada se slika ponovno sastavlja, Cb i Cr kanali se **povećavaju** na odgovarajuću veličinu (veličinu nepromijenjenog Y kanala).

2.2.1. Vrste poduzorkovanja

Postoje tri vrste poduzorkovanja:

1. 4:4:4 uzorkovanje - uopće nema uzorkovanja
2. 4:2:2 uzorkovanje - gdje se smanjuje za faktor 2 u horizontalnom smjeru
3. 4:2:0 uzorkovanje - gdje se smanjuje za faktor 2 u oba smjera (horizontalno i vertikalno)

i prikazuju se s pomoću tri broja J:a:b,

- J je referenca horizontalnog uzorkovanja, obično je 4 jer ne želimo smanjiti kanal svjetline
- a je broj uzoraka boje u prvom redu J piksela
- b je broj promjena uzoraka boje između prvog i drugog reda J piksela. b je uobičajeno 0 ili jednak a.

2.3. Diskretna kosinusna transformacija

Sljedeća dva koraka su ključna za algoritam jer se događa "pravo" kompresiranje. Zamisao je iskoristiti činjenicu da ljudsko oko loše razlikuje elemente **visoke frekvencije**, no, prvo se postavlja pitanje što to zapravo znači. Ljudsko oko lako percipira psa ili drvo na slici 2.1., no loše percipira individualne vlati trave (ili recimo niti na vlastitoj majici), promjene u sjenama, individualne dlake psa itd., i to su upravo promjene visokih frek-

vencija. Također, na slikama u kojima postoji efekt fokusa, razlike u bojama koje nisu u fokusu nam nisu toliko bitne.

Radi efektivnog izbacivanja elemenata različitih frekvencija potrebno je i same vrijednosti prebaciti u **frekvencijsku domenu**. Za to se koristi diskretna kosinusna transformacija (**DCT**).

Prvo se svaka komponenta slike podijeli u 8x8 blokove, svaki sa 64 piksela, (tzv. **MCU blok**). Sljedeća matrica će mi poslužiti kao primjer [5]:

$$\begin{bmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{bmatrix}$$

Zatim se vrijednost svakog piksela oduzme sa 128, kako bi normalizirali vrijednosti (kosinus poprima vrijednosti između -1 i 1 tako da bi bilo nezgodno koristiti 0 - 255 interval, a ako oduzmemo 128 dobivamo interval od -128 do 127).

$$g = \begin{bmatrix} -76 & -73 & -67 & -62 & -58 & -67 & -64 & -55 \\ -65 & -69 & -73 & -38 & -19 & -43 & -59 & -56 \\ -66 & -69 & -60 & -15 & 16 & -24 & -62 & -55 \\ -65 & -70 & -57 & -6 & 26 & -22 & -58 & -59 \\ -61 & -67 & -60 & -24 & -2 & -40 & -60 & -58 \\ -49 & -63 & -68 & -58 & -51 & -60 & -70 & -53 \\ -43 & -57 & -64 & -69 & -73 & -67 & -63 & -45 \\ -41 & -49 & -59 & -60 & -63 & -52 & -50 & -34 \end{bmatrix}$$

Nakon toga se radi DCT pomoću sljedeće formule:

$$G_{u,v} = \frac{1}{4}\alpha(u)\alpha(v)\sum_{x=0}^{7}\sum_{y=0}^{7} g_{x,y}\cos\left[\frac{(2x+1)u\pi}{16}\right]\cos\left[\frac{(2y+1)v\pi}{16}\right] \tag{2.4}$$

gdje je:

- u horizontalna prostorna frekvencija, $0 \leq u < 8$
- v vertikalna prostorna frekvencija, $0 \leq v < 8$
- α je normalizirajući faktor skale kako bi transformacija bila ortonormirana
- $g_{x,y}$ je vrijednost piksela na koordinatama (x, y)
- $G_{u,v}$ je DCT koeficijent na koordinatama (u, v)

te se dobiva sljedeća matrica:

$$G = \begin{bmatrix} -415.38 & -30.19 & -61.20 & 27.24 & 56.12 & -20.10 & -2.39 & 0.46 \\ 4.47 & -21.86 & -60.76 & 10.25 & 13.15 & -7.09 & -8.54 & 4.88 \\ -46.83 & 7.37 & 77.13 & -24.56 & -28.91 & 9.93 & 5.42 & -5.65 \\ -48.53 & 12.07 & 34.10 & -14.76 & -10.24 & 6.30 & 1.83 & 1.95 \\ 12.12 & -6.55 & -13.20 & -3.95 & -1.87 & a1.75 & -2.79 & 3.14 \\ -7.73 & 2.91 & 2.38 & -5.94 & -2.38 & 0.94 & 4.30 & 1.85 \\ -1.03 & 0.18 & 0.42 & -2.42 & -0.88 & -3.02 & 4.12 & -0.66 \\ -0.17 & 0.14 & -1.07 & -4.19 & -1.17 & -0.10 & 0.50 & 1.68 \end{bmatrix}$$

Ova matrica je ključna, jer se kompresija slike obavlja nad njom, i više se ne koriste YCbCr vrijednosti.

2.4. Kvantizacija

Sljedeći korak je korak gdje se odvija sva "magija" JPEG algoritma.

Princip je zapravo vrlo jednostavan, svaku vrijednost iz matrice G se dijeli s vrijednostima iz sljedeće **kvantizacijske tablice Q**:

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

i svaki se rezultat zaokruži na cijeli broj (**ovdje se događa gubljenje podataka**).

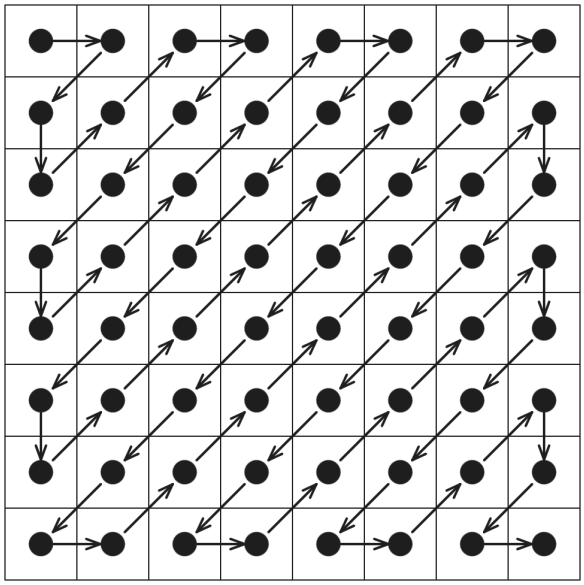
$$B = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Bitno je uočiti da kvantizacijska tablica ima veće brojeve u **donjem desnom kutu**. Tu se nalaze podatci **visoke frekvencije** koje naše oko ne raspoznaje. Kada se podijele svi brojevi uoči se puno nula u rezultantnoj tablici, što omogućuje vrlo efikasno kodiranje i samim time dobru kompresiju. JPEG algoritam koristi **dvije** kvantizacijske tablice, jednu za Y kanal a drugu za Cb i Cr kanale koja ima puno veće vrijednosti kako bi se dobilo što više nula na kanalima boja. Također, kao što se moglo naslutiti, kvantizacijska tablica određuje količinu kompresije i samim time kvalitetu slike. Iz toga proizlazi da mijenjanjem kvalitete JPEG slike zapravo mijenjamo kvantizacijsku tablicu.

2.5. Pohrana podataka

Pomoću prethodna dva koraka, svaki 8x8 blok piksela se pretvori u malo cijelih brojeva i jako puno nula. Takve blokove je potrebno **efektivno** kodirati kako bi se koristilo što **manje** prostora.

Na matrici je vidljivo da se spomenuti brojevi nalaze u **gornjem lijevom kutu**, te radi što efektivnijeg kodiranja, blok će se opisati **cik-cak uzorkom**:



Slika 2.3. Prikaz cik-cak uzorka za 8x8 blok

Nakon toga primjenjuje se tzv. **Run Length Encoding (RLE)** algoritam gdje se **grupira** ponavljanje istih uzastopnih brojeva, kao na sljedećem primjeru:

niz AAAABBBCCDAA RLE kompresira na sljedeći način:

- 4 uzastopna znaka A - 4A
- 3 uzastopna znaka B - 3B
- 2 uzastopna znaka C - 2C
- 1 znak D - 1D
- 2 uzastopna znaka A - 2A

iz čega slijedi sljedeća reprezentacija niza: 4A3B2C1D2A.

Već se može naslutiti koliko će ovakav način biti koristan kada se primjeni na skup od recimo 10 ili više uzastopnih nula dobivenih kvantizacijom.

Napokon, na dobiveni niz brojeva ćemo primijeniti algoritam **Huffmanovog kodiranja**, o kojem će biti riječi u nastavku.

2.6. Dekodiranje

Gornji koraci provedeni tim redoslijedom dat će nam kodiranu datoteku koja predstavlja sliku. Ako je želimo vidjeti na zaslonu potrebno ju je **dekodirati**. Dekodiranje je, u principu, gornji postupak proveden **obrnutim redoslijedom**. Dakle, provesti Huffmanovo dekodiranje datoteke, dekvantizaciju, IDCT (*inverzna* diskretna kosinusna transformacija) i na kraju vratiti se u RGB prostor i prikazati sliku na ekranu.

3. Huffmanovo kodiranje

Huffmanov kôd je **optimalan prefiksni kôd varijabilne duljine** nastao 1952. i koristi se kod kompresije podataka bez gubitaka. Ideja iza njega je da se simboli koji se **često pojavljuju** kodiraju s kodovima **manje** duljine, dok oni koji se rjeđe pojavljuju, dužom duljinom.

Prefiksni kôd je kôd gdje niti jedna kodna riječ ne smije biti prefiks neke druge kodne riječi, a **optimalnost** u kontekstu kodiranja podrazumijeva minimiziranje očekivane duljine kodiranih podataka. To se postiže zadovoljavanjem **Kraftove nejednakosti**:

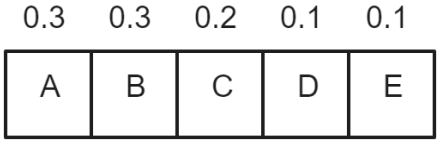
$$\sum_{i=1}^{n} d^{l_i} \leq 1 \tag{3.1}$$

Rad algoritma demonstriram na sljedećem primjeru.

3.1. Algoritam

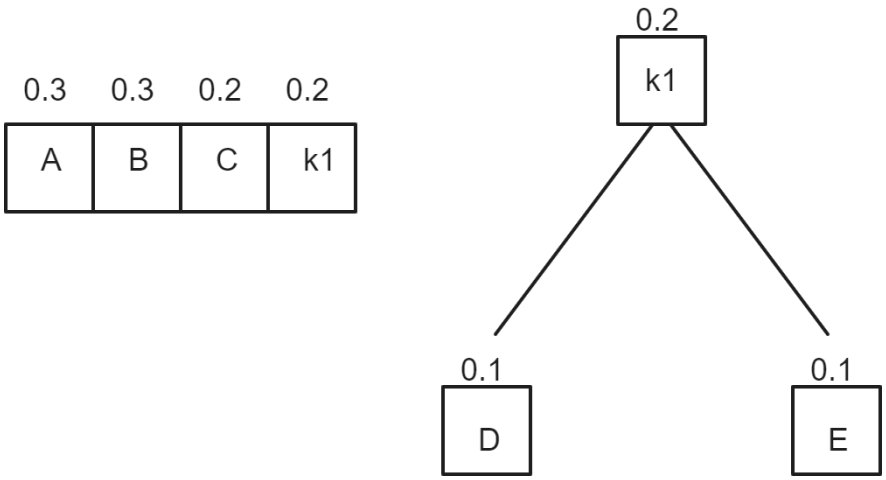
Na sljedećoj stranici postoji jako dobar simulator Huffmanovog stabla za proizvoljni niz. No, proći ću ga i ručno pa uzmimo za primjer sljedeći niz: *BACDAEABBC*

Prvi korak je da se simboli (u ovom slučaju znakovi abecede) poredaju po frekvenciji pojavljivanja (padajuće). Što daje sljedeće:



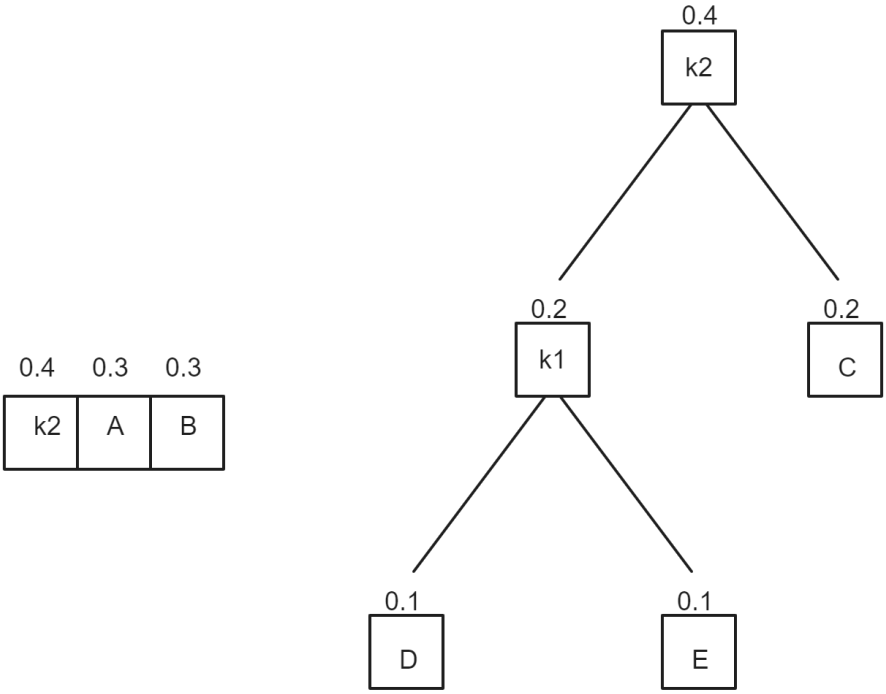
Slika 3.1. Ulazni niz poredan po frekvenciji pojavljivanja

Sad uzimam dva simbola s **najmanjim vrijednostima**, u ovom slučaju D i E, i "spajam" ih u jedan nadsimbol koji će imati frekvenciju pojavljivanja kao njihov zbroj. Nazvat ću ga k_1 i vrijednost mu je 0.2. Dobivamo sljedeću situaciju

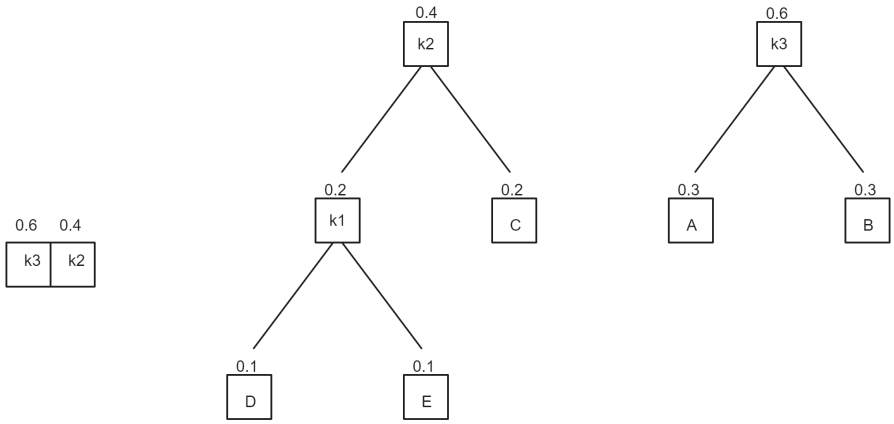


Slika 3.2. Stanje nakon prvog koraka algoritma

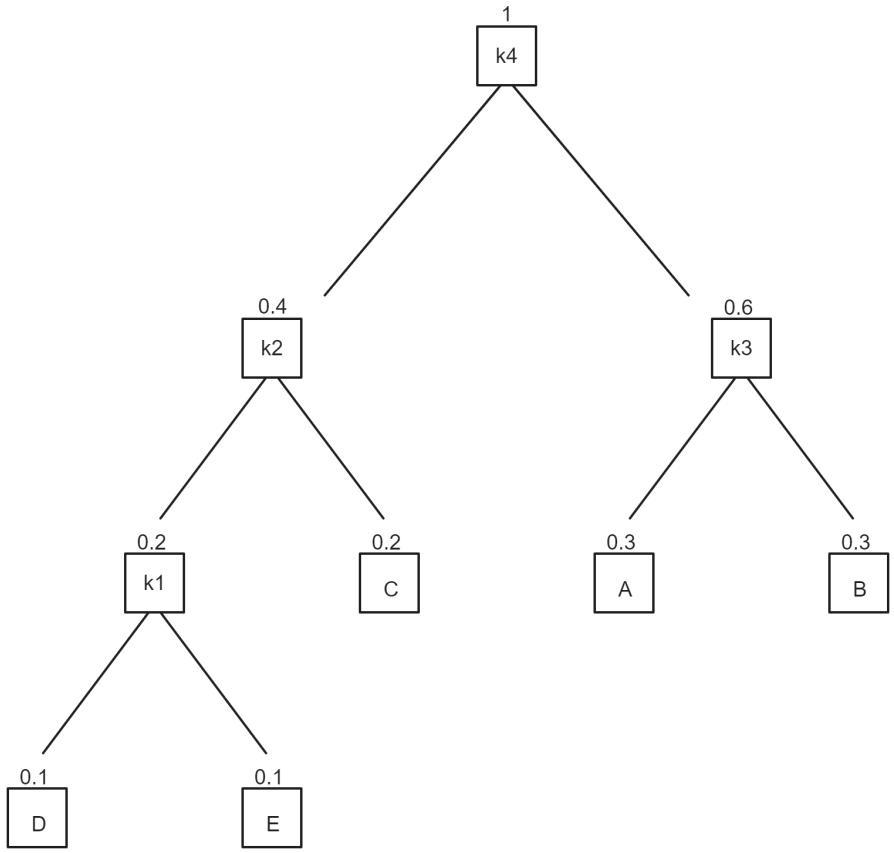
Sada **ponavljam prethodni korak** sve dok ne dobijem **jedan nadsimbol**.



Slika 3.3. Stanje nakon drugog koraka algoritma, spajam k_1 i C

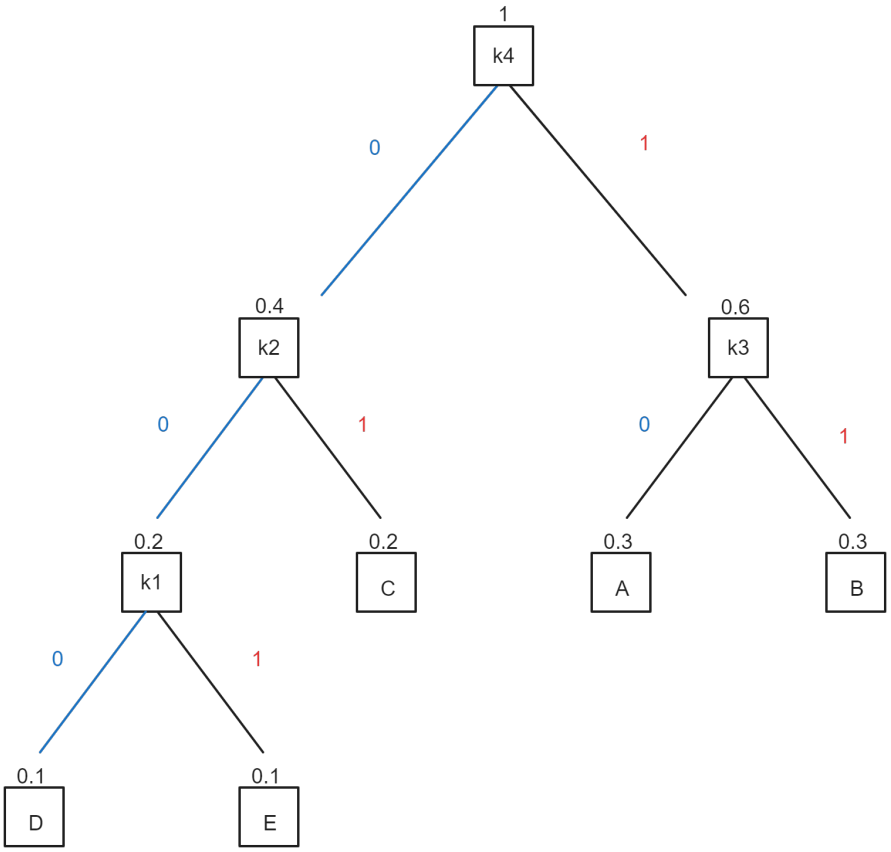


Slika 3.4. Stanje nakon trećeg koraka algoritma, spajam A i B



Slika 3.5. Stanje nakon petog koraka algoritma, spajam k_2 i k_3

Struktura koju dobivam je **binarno stablo** iz kojeg očitavam kodove za pojedine simbole na sljedeći način: kada imamo grananje, lijevoj grani dodjeljuje se vrijednost 0, a desnoj 1. To se ponavlja skroz do pojedinog simbola (lista) i vrijednosti koje očitamo su njegov kod. Recimo za simbol D možemo očitati da je njegov kôd 000 (slika 3.6).



Slika 3.6. Očitanje koda simbola D

Dekodiranje slijeda podataka je također dosta jednostavno. Svodi se na to da se uz niz podataka u datoteku stavi i cijelo stablo dobiveno kodiranjem i tako se vrlo lagano dekodira niz. Čita se znak po znak u nizu i kreće po stablu dok se ne dođe do lista.

3.2. Modifikacija prilagođena za FPGA

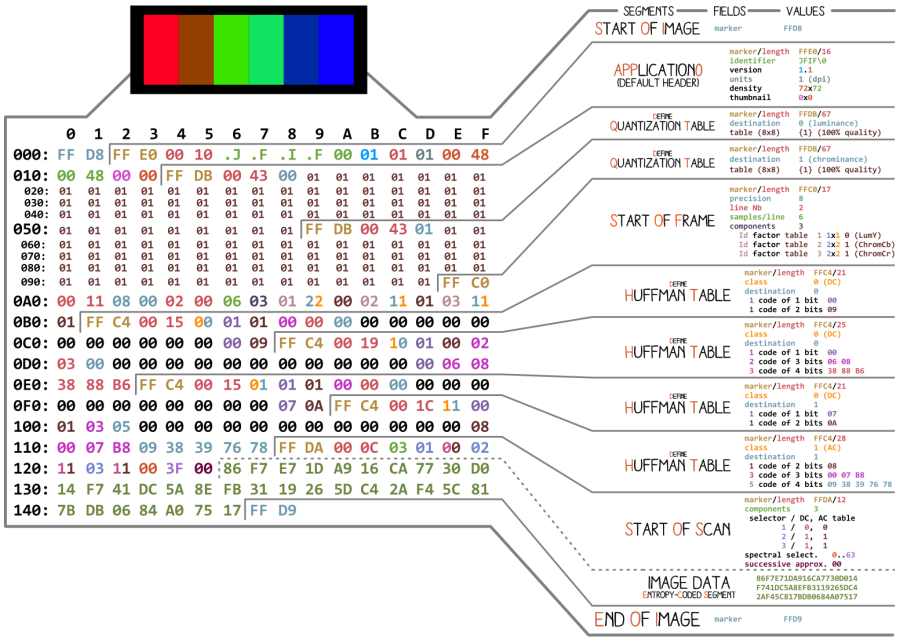
Gradnja stabla na FPGA pločici je nemoguća zato što stabla koriste dinamičke strukture podataka. Problem se rješava tako da se koristi drukčija metoda Huffmanovog dekodiranja. FER-ov JPEG koder, Jaguar, koristi metodu izgradnje stabla, te se kod mora promijeniti kako bi radio na drugačiji način. Više o tome u poglavlju o implementaciji.

4. Implementacija

Učitati podatke iz datoteke je prva stvar koju je potrebno napraviti prilikom dekodiranja. JPEG ima definirano **zaglavlje** (engl. header) pomoću kojega dohvaća sve tražene podatke o slici.

4.1. JPEG zaglavlje

JPEG zaglavlje je sastavljeno od oznaka tj. **markera** pomoću kojih se odvajaju dijelovi zaglavlja, kao što je prikazano na slici 4.1.



Slika 4.1. Slikovni prikaz JPEG zaglavlja [6]

Markeri se sastoje od **dva bajta**. Prvi bajt je uvijek 0xFF, a pomoću drugog bajta određujemo koji podatci slijede, koji je odnosno točno marker je u pitanju. Nakon ta dva bajta, obično dolaze dva bajta koji označavaju duljinu podataka koji se odnose na taj

marker. Markeri nemaju striktni redoslijed po kojem se pojavljuju, no najčešći redoslijed je sljedeći:

Prva oznaka u datoteci je **0xFFD8** što označava početak slike **(engl. SOI (Start of Image))**. Iza ove oznake nema nikakvih podataka, te nema nikakve duljine. Neposredno nakon SOI oznake dolazi aplikacijska **(engl. APPN (Application))** oznaka, **0xFFEN** (kojih može biti ukupno 16, N ide od 0x0 do 0xF). Oznaka predstavlja specifičnosti aplikacije (npr. ako je datoteka iz programa Photoshop ili slično), no generalno nije pretjerano bitna te se neće koristiti u ovom radu. Iza ove oznake slijedi dva bajta koja govore kolika je duljina APPN oznake (uključujući ta dva bajta).

Nakon aplikacijske oznake slijede **kvantizacijske tablice** označene s **0xFFDB**. Također iza ove oznake slijedi dva bajta duljine, no slijedi i jedan bajt koji je informacija o samoj tablici. Prva četiri bita u bajtu govore je li 8 ili 16 bitna tablica (najčešće se koriste 8 bitne tablice). Zadnja četiri bita predstavljaju ID tablice. Nakon tog bajta, slijedi sama tablica.

Poslije kvantizacijskih tablica slijedi **engl. SOF (Start of Frame)** oznaka. SOF oznaka ima 13 različitih, ali samo jedna se može naći u datoteci. Označava se s **0xFFCN** gdje je N između 0x0 i 0xF (**ne** uključujući 0x4, 0x8 i 0xC). Oznaka **0xFFC0** se najčešće koristi (**baseline JPEG**). Baseline znači da se nalazi samo jedan Huffmanov kodiran tok podataka u datoteci. Dodatno se nakon SOF oznake, osim duljine, nalazi informacija o preciznosti (uvijek je 8 bita), dimenzije slike, broj komponenti (jedan ili tri), te slijed informacija o svakoj komponenti (ID, faktor uzorkovanja i ID kvantizacijske tablice).

Postoji i **DRI** oznaka koja predstavlja **interval resetiranja**. Interval resetiranja označava koliko često se vrijednost **DC koeficijenta** u MCU bloku resetira, odnosno, pribroji nula umjesto vrijednosti u prethodnom MCU bloku. Oznaka je **0xFFDD** iza koje slijede duljina (koja iznosi četiri) i dva bajta koja označavaju broj koliko se često resetira.

Nakon toga dolazi oznaka Huffmanovih tablica **engl. DHT (Define Huffman Tables) 0xFFC4**. Iza čega slijedi duljina, informacije o tablici (prva četiri označavaju radi li se o AC ili DC tablici, druga četiri ID tablice), zatim 16 bajtova koji govore koliko je kodova te duljine (npr. 9. bajt koji pročitamo, označava koliko ima kodova duljine 9 bitova itd.). Zatim slijede **simboli**. Simbol koristimo za dekodiranje koeficijenta u MCU

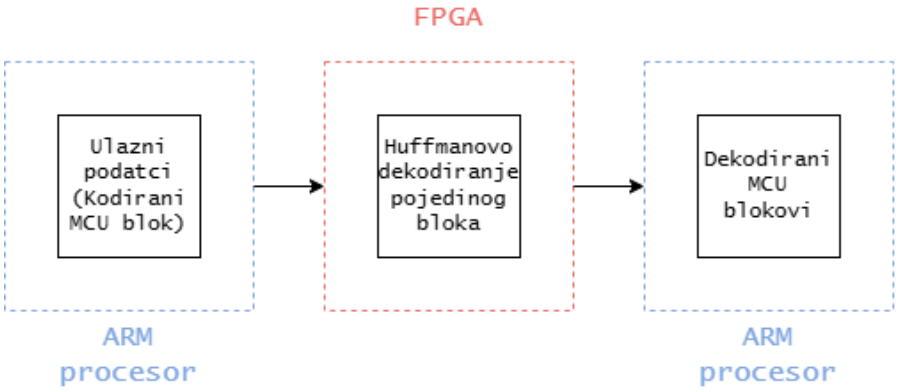
bloku. Prva četiri bita u simbolu označavaju koliko se nula nalazi prije trenutnog koeficijenta (0 do 15 u AC, 0 u DC koeficijentu), a druga četiri bita duljinu koeficijenta kojeg dekodiramo u bitovima (1 do 10 u AC, 0 do 11 u DC koeficijentu).

Nakon DHT oznake, napokon, slijedi **engl. SOS (Start of Scan)** koji predstavlja podatke kodirane Huffmanovim kodiranjem, dakle samu sliku. Oznaka je **0xFFDA**.

Čitanje zaglavlja je pisano u Python-u kako bi se pročitala slika u Jupyter Notebook-u i poslali podatci na FPGA.

4.2. Implementacija

Blok shema sustava prikazana je na slici 4.2. Kao što sam spomenuo, slanje podataka obavljat će se na Jupyter Notebook-u, odnosno na ARM procesoru PYNQ pločice. Dekodiranje pojedinog MCU bloka se zatim izvodi na FPGA pločici, te se šalju dekodirane vrijednosti nazad na ARM.



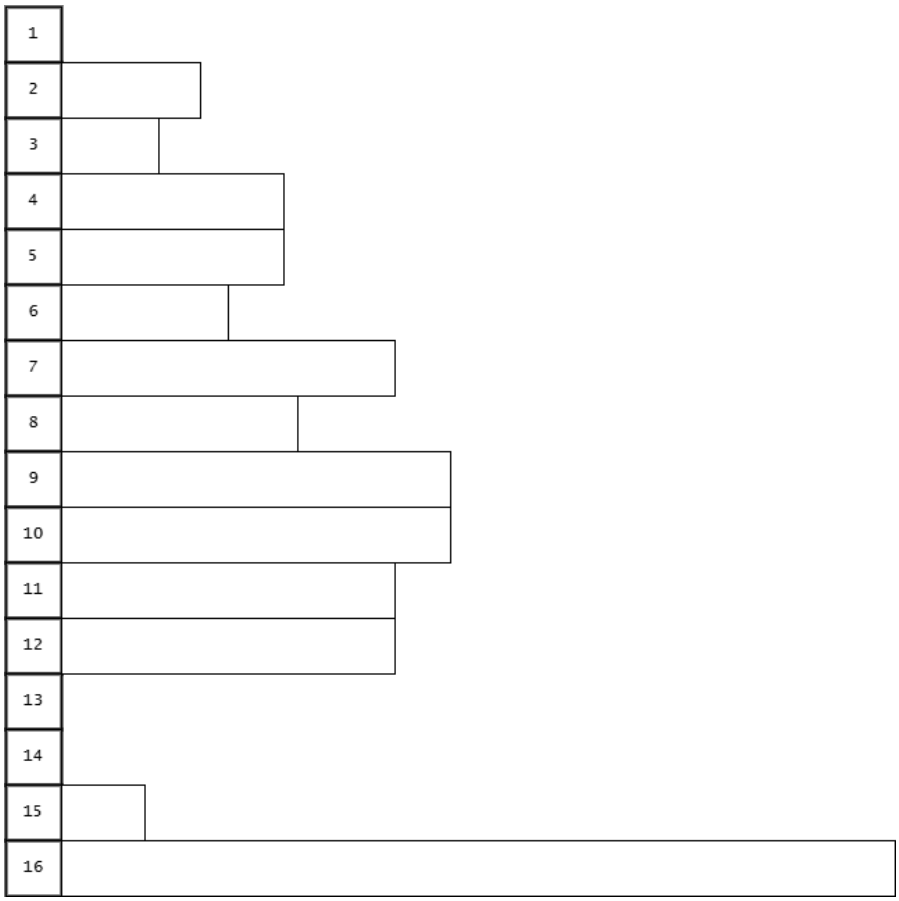
Slika 4.2. Blok shema sustava

Program može dekodirati slike kojima je veličina kodiranog bitstream-a manja od 1000000. Ovaj nedostatak se može riješiti korištenjem DMA sklopa, no u ovom radu nije korišten.

4.2.1. Huffmanovo dekodiranje

Budući da ne možemo graditi stabla na FPGA pločici morat ću prilagoditi dekodiranje Huffmanovih podataka.

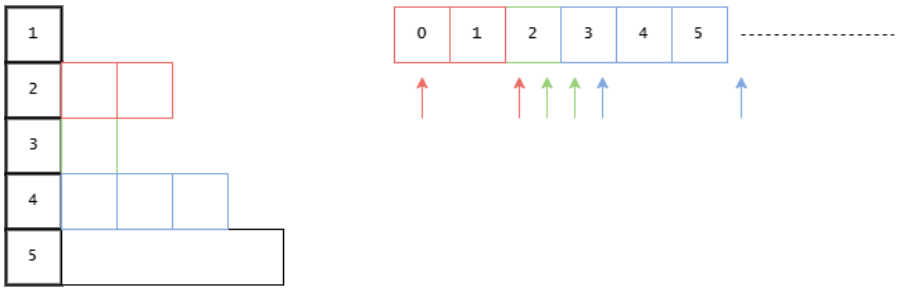
Struktura koju dobijemo iz čitanja Huffmanovih tablica u zaglavlju je **"zupčasto" dvo-dimenzionalno polje** [7] koje izgleda nešto kao na slici 4.3.



Slika 4.3. Prikaz zupčastog 2D polja

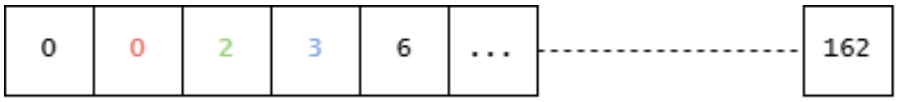
Problem kod ove strukture je njen prikaz, jer su veličine polja varijabilne duljine. Iz tog razloga koristim pomoćni niz `byte symbols[162]` u koji ću pohraniti sve simbole. Potrebno je znati kako grupirati te simbole. Taj problem ću riješiti na sljedeći način:

Niz simbola može izgledati ovako:



Slika 4.4. Prikaz `symbols` niza

Vidljivo je da simboli duljine 2 počinju na indeksu 0 i završavaju na indexu 2 (crvene strelice). Simboli duljine 3 počinu na indeksu 2 i završavaju na indeksu 3 itd. Zeleni indeksi počinju isto gdje i crveni završavaju. Te dvije informacije su suvisle pa možemo izbaciti informaciju gdje trenutna duljina završava. Te indekse (pomake) pohranjujemo u još jedan niz `byte offsets[17]` u kojem će se pohraniti **početnu poziciju duljine** u `byte symbols[162]` nizu. Taj niz može izgledati ovako:



Slika 4.5. Prikaz `offsets` niza

Sada kada imamo prikaz Huffmanove tablice pomoću ova dva niza potrebno je odrediti kodne riječi.

Algoritam je sljedeći:

- za svaki kod duljine n, pohrani trenutnu kodnu riječ,
- onda uvećaj kodnu riječ za 1,
- svaki put kada se pomaknemo na novu duljinu, dodamo 0 na kraj trenutne kodne riječi.

U sljedećem primjeru donosim kako odrediti kodne riječi. Uzimam niz sa slike 4.4.

Dakle, ulaz je broj koliko kodova ima svake duljine.

Krećem od duljine jedan i trenutne kodne riječi 0. Gotovo nikad nećemo imati kodova duljine jedan jer bi bile jedine kodne riječi budući da se radi o prefiksnom kodu.

S obzirom, da kodova duljine jedan nema, prelazim na kodne riječi duljine dva kojih ima dvije. Svaki put kada povećam duljinu riječi dodat ću jednu 0 na kraj trenutne kodne riječi, pa je trenutna kodna riječ 00. Budući da ima kodova duljine dva, prvo ću pohraniti 00 u skup kodnih riječi i povećati za 1 (dakle dobijem 01). Kako ima dva koda duljine dva, još jednom ponovim postupak. Dakle, spremim kodnu riječ 01, povećam za 1, dobijem 10.

Sada prelazim na sljedeću duljinu, duljinu tri. Kodova duljine tri ima samo jedan pa ponavljam postupak samo jednom. Prvo dodajem 0 na kraj trenutne kodne riječi (100), pohranjujem trenutnu riječ i uvećavam za 1 (101).

I tako ponavljam postupak dok ne dođem do kraja, odnosno duljine 16.

4.2.2. Programski kod

U kodu se mogu vidjeti tzv. **pragme**. To su direktive koje prevoditelju daju nekakve informacije. Najčešće korištene pragma je `#pragma HLS pipeline off` koja navodi prevoditelja, odnosno sintetizatora da ne pokuša napraviti protočnu strukturu za danu `for` petlju. Isključili smo pipeline kako ne bi došlo do brkanja podataka dok se iterira po nizovima. Osim te koristi se i pragme koje služe za komunikaciju s pločicom:

```
#pragma HLS INTERFACE s_axilite port=x
#pragma HLS INTERFACE m_axi port=x depth=y offset=slave
```

`s_axilite` sučelje je jednostavno sučelje koje omogućava upis podatka direktno u memoriju pločice. Koristio sam ga za slanje jednostavnih podataka (ne polja).

Polja su nešto kompleksnija pa sam koristio `m_axi` sučelje koje mapira fizičku adresu polja napravljenog u Jupyter Notebook-u sa FPGA.

Cijeli kod (dekodera) nalazi se na GitHub repozitoriju, ovdje ću proći samo najbitnije funkcije vezane uz Huffmanovo dekodiranje [8] [9].

Korištene strukture i klase

Čitanje zaglavlja odvija se na Jupyter Notebook-u pa sam za zaglavlje napisao klasu u Python-u:

```python
class Header:
    def __init__(self) -> None:
        self.quantization_tables = [QuantizationTable() for _ in range(4)]
        self.huffman_dc_tables = [HuffmanTable() for _ in range(4)]
        self.huffman_ac_tables = [HuffmanTable() for _ in range(4)]
        self.color_components = [ColorComponent() for _ in range(3)]

        self.frame_type = 0
        self.height = 0
        self.width = 0
        self.num_components = 0
        self.zero_based = False

        self.start_of_selection = 0
        self.end_of_selection = 0
        self.successive_approximation_high = 0
        self.successive_approximation_low = 0

        self.huffman_data = []

        self.restart_interval = 0

        self.valid = True

        self.mcuHeight = 0
        self.mcuWidth = 0
        self.mcuHeightReal = 0
        self.mcuWidthReal = 0

        self.horizontalSamplingFactor = 1
        self.verticalSamplingFactor = 1
```

Slika 4.6. Struktura zaglavlja

U C++-u sam napravio zasebnu strukturu za Huffmanove tablice kako bi bilo lakše snaći se u kodu:

```cpp
struct HuffmanTable {
    int offsets[17]  = { 0 };
    int symbols[162] = { 0 };
    int codes[162]   = { 0 };
    int set          = false;
};
```

Slika 4.7. Struktura za tablice

Generiranje kodnih riječi

Sljedeći isječak koda je odgovoran za gornje opisani algoritam Huffmanovog dekodiranja.

```
def generate_codes(hTable):
    code = 0
    for i in range(0, 16):
        for j in range(hTable.offsets[i], hTable.offsets[i + 1]):
            hTable.codes[j] = code
            code = code + 1
        code = code << 1
```

Slika 4.8. Kod za generiranje kodova

MCU Dekodiranje

Sljedeća funkcija je tzv. **top** funkcija koda, točnije funkcija od koje kreće sinteza u VHDL. Deklaracija funkcije izgleda kao na slici 4.9. Mogu se uočiti prethodno spomenute direktive za pojedine argumente.

```
void decodeMCUComponent(int nextByte, int nextBit,
                        int data[10000], int dataSize, int align, int bitValues[2],
                        int component[64], int previousDC, int &fail,
                        int dc_table[162+162+17+1], int ac_table[162+162+17+1]) {
#pragma HLS INTERFACE s_axilite port = nextByte
#pragma HLS INTERFACE s_axilite port = nextBit
#pragma HLS INTERFACE s_axilite port = dataSize
#pragma HLS INTERFACE s_axilite port = align
#pragma HLS INTERFACE s_axilite port = previousDC
#pragma HLS INTERFACE s_axilite port = fail
#pragma HLS INTERFACE s_axilite port = bitValues

#pragma HLS INTERFACE m_axi port = component depth=64 offset=slave
#pragma HLS INTERFACE m_axi port = dc_table depth=342 offset=slave
#pragma HLS INTERFACE m_axi port = ac_table depth=342 offset=slave
#pragma HLS INTERFACE m_axi port = data depth=10000 offset=slave

#pragma HLS INTERFACE s_axilite port = return
```

Slika 4.9. Deklaracije funkcije

Funkcija pretvara niz Huffmanovih simbola u MCU blok.

Dio koda koji dekodira DC koeficijent prikazuje slika 4.10., a slika 4.11. prikazuje dekodiranje AC koeficijenata. Također na slici 4.10. se vidi i učitavanje podataka (tablica) iz ulaznog niza.

```
{
    fail = 200;

    HuffmanTable dcTable;
    HuffmanTable acTable;

    BitReader b = BitReader(nextByte, nextBit, data, dataSize);

    if (align) {
        b.align();
    }

    // read from input tables
    int symbols_index = 0, codes_index = 0, offsets_index = 0;
    for (int i = 0; i < 342; i++) {
#pragma hls pipeline off
        if (i < 162) {
            dcTable.symbols[symbols_index] = dc_table[i];
            acTable.symbols[symbols_index] = ac_table[i];
            symbols_index++;
        }
        else if (i < 162+162) {
            dcTable.codes[codes_index] = dc_table[i];
            acTable.codes[codes_index] = ac_table[i];
            codes_index++;
        } else if (i < 162+162+17) {
            dcTable.offsets[offsets_index] = dc_table[i];
            acTable.offsets[offsets_index] = ac_table[i];
            offsets_index++;
        }
        else {
            dcTable.set = dc_table[i];
            acTable.set = ac_table[i];
        }
    }

    fail = 300;

    byte length = getNextSymbol(b, dcTable);
    if (length == (byte)-1) {
        // std::cout << "Error - Invalid DC value\n";
        fail = 500;
        return;
    }
    if (length > 11) {
        // std::cout << "Error - DC coefficient length greater than 11\n";
        fail = 500;
        return;
    }

    int coeff = b.readBits(length);
    if (coeff == -1) {
        // std::cout << "Error - Invalid DC value\n";
        fail = 500;
        return;
    }
    if (length != 0 && coeff < (1 << (length - 1))) {
        coeff -= (1 << length) - 1;
    }
    component[0] = coeff + previousDC;

```

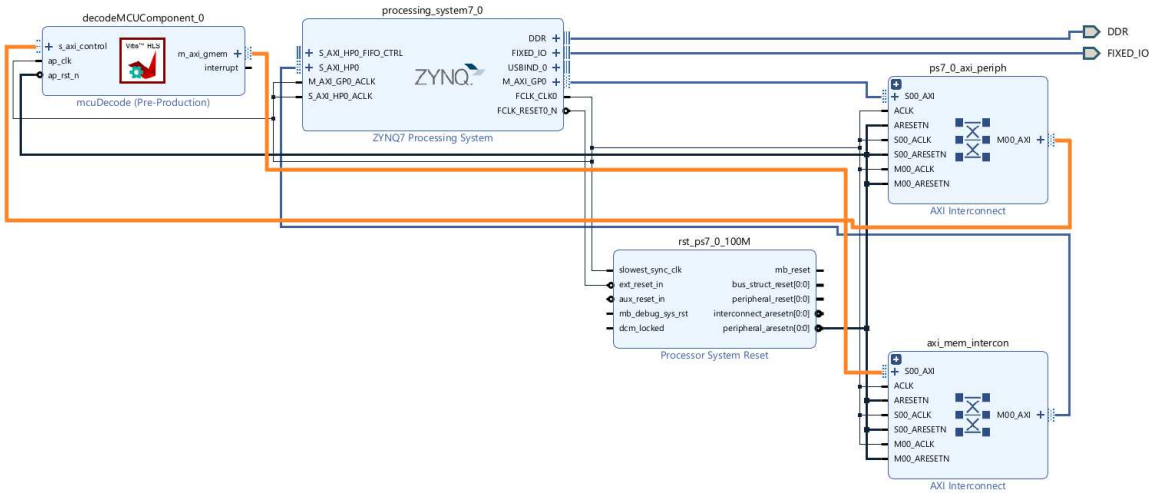
Slika 4.10. Kod za dekodiranje MCU bloka (DC dio)

```
    // get the AC values for this MCU component
    uint i = 1;
    while (i < 64) {
#pragma hls pipeline off
        byte symbol = getNextSymbol(b, acTable);
        if (symbol == (byte)-1) {
            // std::cout << "Error - Invalid AC value\n";
            fail = 500;
            return;
        }

        // symbol 0x00 means fill remainder of component with 0
        if (symbol == 0x00) {
            for (; i < 64; ++i) {
#pragma hls pipeline off
                component[zigZagMap[i]] = 0;
            }

            fail = 400;

            bitValues[0] = b.nextByte;
            bitValues[1] = b.nextBit;
            return;
        }

        // otherwise, read next component coefficient
        byte numZeroes = symbol >> 4;
        byte coeffLength = symbol & 0x0F;
        coeff = 0;

        // symbol 0xF0 means skip 16 0's
        if (symbol == 0xF0) {
            numZeroes = 16;
        }

        if (i + numZeroes ≥ 64) {
            // std::cout << "Error - Zero run-length exceeded MCU\n";
            fail = 500;
            return;
        }
        for (uint j = 0; j < numZeroes; ++j, ++i) {
#pragma hls pipeline off
            component[zigZagMap[i]] = 0;
        }

        if (coeffLength > 10) {
            // std::cout << "Error - AC coefficient length greater than 10\n";
            fail = 500;
            return;
        }
        if (coeffLength ≠ 0) {
            coeff = b.readBits(coeffLength);
            if (coeff == -1) {
                // std::cout << "Error - Invalid AC value\n";
                fail = 500;
                return;
            }
            if (coeff < (1 << (coeffLength - 1))) {
                coeff -= (1 << coeffLength) - 1;
            }
            component[zigZagMap[i]] = coeff;
            i += 1;
        }
    }
    fail = 400;
    bitValues[0] = b.nextByte;
    bitValues[1] = b.nextBit;
    return;
}

```

Slika 4.11. Kod za dekodiranje MCU bloka (AC dio)

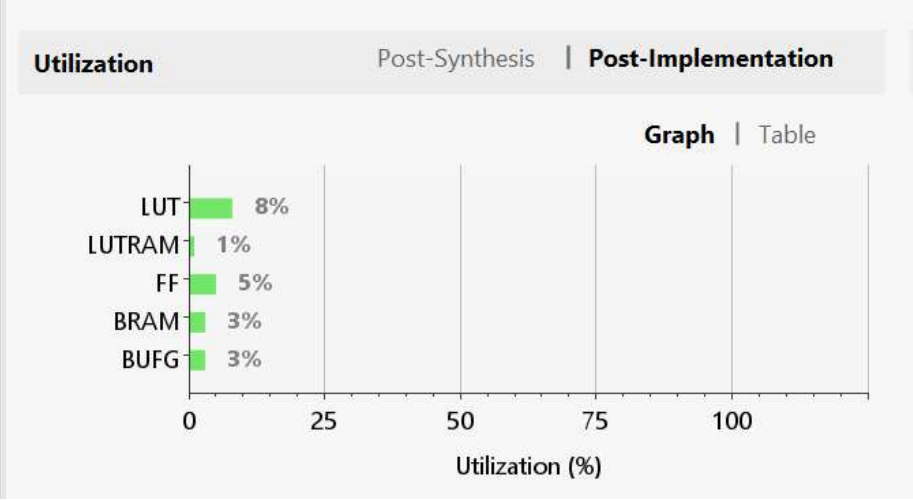
4.2.3. FPGA

Nakon što sam dobio IP iz Vitis-a, potrebno ga je ubaciti u Vivado i napraviti blok dijagram sustava, slika 4.12. (uočiti **narančaste linije**, to su prethodno spomenuti `m_axi` priključci koje su mapirali pomoći direktivi). Pomoću blok dijagrama napravi se bitstream za pločicu i sve je spremno za testiranje.

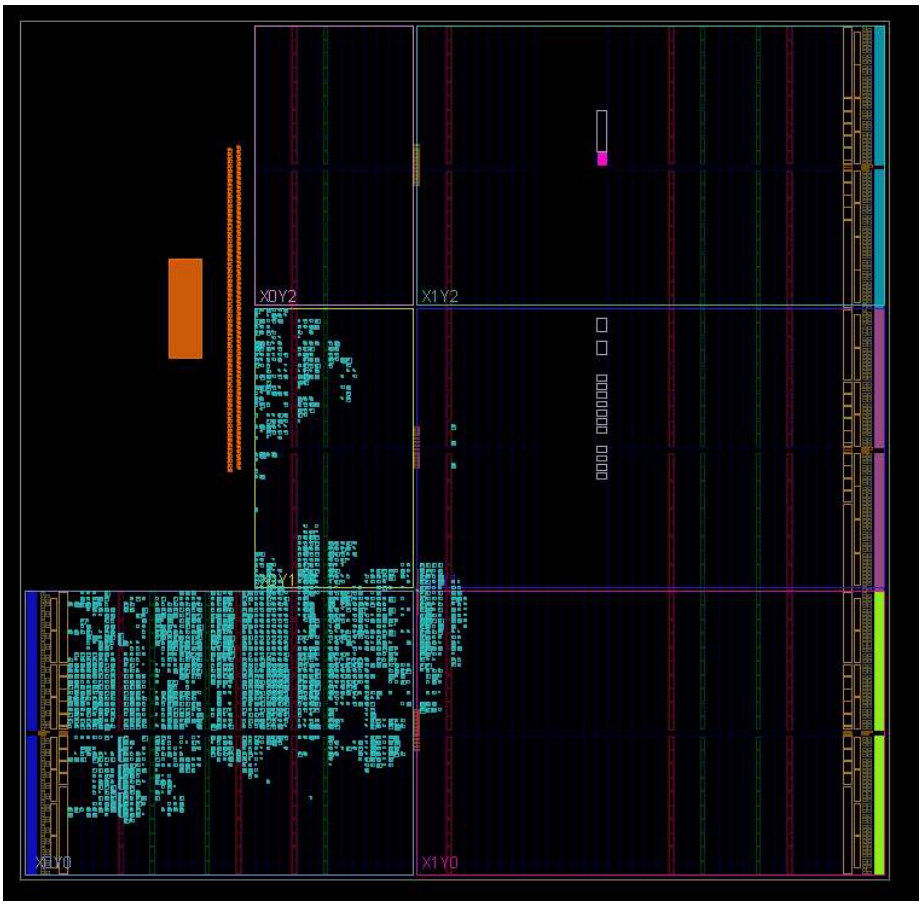


Slika 4.12. Blok dizajn sustava

Dodatno, nakon što se napravi bitstream može se vidjeti i konačan dizajn na pločici, prikazan na slici 4.14., kao i iskorištenost resursa na pločici na slici 4.13.



Slika 4.13. Iskorištenost FPGA pločice



Slika 4.14. Prikaz dizajna na FPGA pločici

4.2.4. Jupyter Notebook

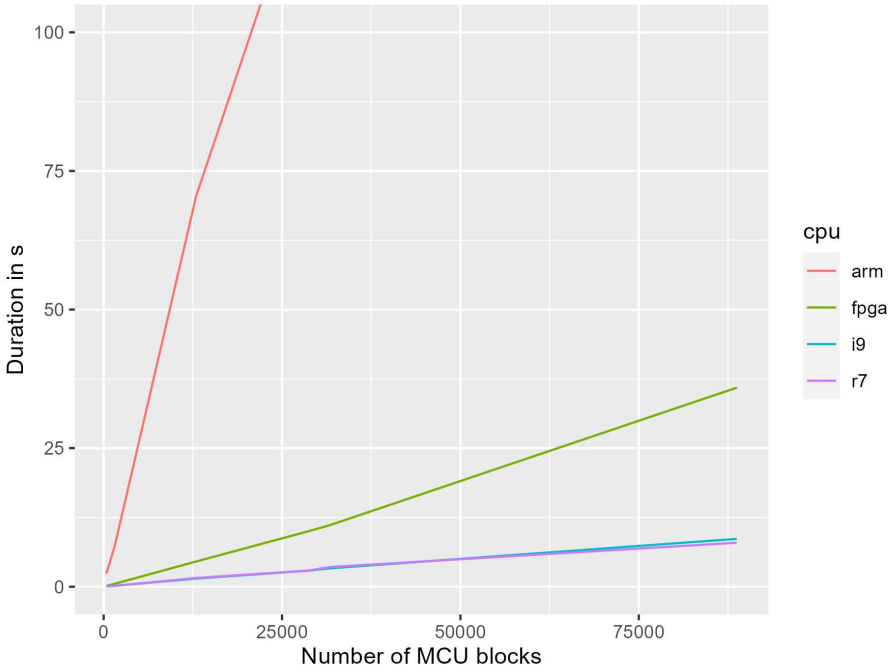
Slijedi prebacivanje bitstreama na pločicu. Prvo je potrebno `bit` i `hwh` datoteke prebaciti na Jupyter Notebook i učitati željenu sliku. Nakon toga pokrećem Python skriptu koja će prebaciti bitstream na pločicu, poslati potrebne podatke za dekodiranje i pokrenuti proces dekodiranja.

4.3. Usporedba dekodiranja

Uspoređene su performanse dekodiranja na pločici (**hardveru**), na procesoru **ARM** na pločici, na procesoru **i9-12900k** i na procesoru **Ryzen 7 5700U**. Tablica 4.1. prikazuje spomenutu usporedbu, sva vremena su prikazana u sekundama, a na slici 4.15. prikazan je graf usporedbe performansi.

Tablica 4.1. Usporedba performansi

Slika	Veličina slike	Broj MCU blokova	Izvođenje na 12900k	Izvođenje na 5700U	Izvođenje na ARM	Izvođenje na FPGA
butterdog	128x128	384	0.093693	0.053888	2.371583	0.146735
jpeg420	256x256	1536	0.184036	0.161097	7.257743	0.527120
leaf	640x427	12960	1.479456	1.587361	70.439218	4.548730
img_0_8b	1280x960	28800	2.910319	2.92357	131.08171	10.043925
dog	1000x1334	31752	3.327081	3.55682	171.12383	11.124405
pero	1387x1354	88740	8.630684	7.937819	505.23192	35.909214



Slika 4.15. Graf usporedbe performansi

5. Zaključak

U radu je implementiran JPEG dekoder na FPGA pločici. Komunikacija s pločicom je ostvarena pomoću Jupyter Notebook sučelja. Uloga FPGA pločice je da bude akcelerator koji će ubrzati dio algoritma, u ovom slučaju dekodiranje pojedinog MCU bloka. Iz rezultata vidimo da su vremena dekodiranja usporediva s procesoroma u današnjim računalima, te puno bolja od ARM procesora na pločici.

Algoritam može dekodirati proizvoljne veličine slika budući da se šalje samo jedan blok. Dana implementacija se može znatno poboljšati modifikacijom komunikacije s pločicom, odnosno, korištenje DMA sklopa. No, radi jednostavnosti DMA sklop nije korišten.

Literatura

[1] J. Griffin, “The ultimate guide to jpeg including jpeg compression and encoding”, https://www.thewebmaster.com/jpeg-definitive-guide, 2023.

[2] L. Crockett, *The Zynq Book*. Strathclyde Academic Media, 2014.

[3] M. Vučić, *Alati za razvoj digitalnih sustava - Materijali za predavanja*. FER-ZESOI, 2009.

[4] Browserling, “Ycbcr image color extractor”, https://onlinetools.com/image/show-ycbcr-image-colors.

[5] Wikipedia, https://en.wikipedia.org/wiki/JPEG.

[6] Y. Khalid, https://yasoob.me/posts/understanding-and-writing-jpeg-decoder-in-python, july 14, 2020.

[7] D. Harding, https://www.youtube.com/watch?v=TlrNCT15NM4, october 28, 2018.

[8] F. Fodor, https://git.hpc.fer.hr/dhofman/jaguarhls.

[9] D. Harding, https://github.com/dannye/jed.

Sažetak

FPGA implementacija Huffmanovog dekodiranja JPEG dekodera

Fran Fodor

Implementacija JPEG dekodera na FPGA PYNQ-Z1 pločici pisana je u C++-u i sintetiziran pomoću Vitis HLS-a. HLS automatizira implementaciju C/C++ koda u Verilog koji se stavlja na pločicu. Upravljanje pločicom ostvareno je pomoću Jupyter Notebook-a. Preko Jupyter Notebook-a šalju se potrebni podatci i primaju dekodirane vrijednosti. Također je provedena usporedba dekodiranja na FPGA, ARM procesoru na pločici, na procesoru Ryzen 7 5700U te na procesoru i9-12900k.

Ključne riječi: JPEG; FPGA; PYNQ; Huffmanovo kodiranje

Abstract

FPGA implementation of Huffman decoding in JPEG decoder

Fran Fodor

Implementation of JPEG decoder on PYNQ-Z1 FPGA board written in C++ and synthesized using Vitis HLS. HLS automatically translates C++ code into Verilog code which is used to program the functionality of board. Communication with the board is achieved using Jupyter Notebook through which data is sent and received. Comparison is done using the board, ARM CPU on the board, Ryzen 7 5700U CPU and i9-12900k CPU.

Keywords: JPEG; FPGA; PYNQ; Huffman encoding