Hardverska implementacija Viola-Jones algoritma

Risto Pejašinović

Sadržaj

Deo I
Viola-Jones algoritam

1 Uvod

Namena algoritma je detekcija i lokalizacija objekata na slici. Osmišljen od strane Paul Viola i Michael Jones 2001. godine [1].

Dugo godina je zbog brze i pouzdane detekcije bio standardan način detekcije lica na slici. I danas je prisutan u velikom broju mobilnih telefona i digitalnih kamera, ali danas postaje polako zamenjen konvolucionim neuronskim mrežama.

Pouzdanost i brzina su postignuti uvođenjem tri ključna doprinosa:

- **Integralna slika** omogućava brzo izračunavanje obeležja.
- **AdaBoost** algoritam za učenje, odabiranjem obeležja povećava brzinu i pouzdanost detekcije.
- **Kaskadni klasifikator** organizovanjem obeležja u kaskadama omogućava brzo odbacivanje pozadine slike.

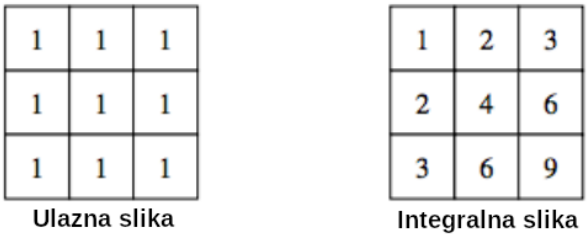
1.1 Integralna slika

Kao jedan od ključnih delova algoritma, integralna slika omogućava izračunavanje površine svakog pravouganog obeležja u konstantnom vremenu.

Intenzitet piksela u integralnoj slici na poziciji x,y je zbir svih piksela koji se nalaze gore i levo od pozicije x,y.

$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y') \tag{1.1}$$

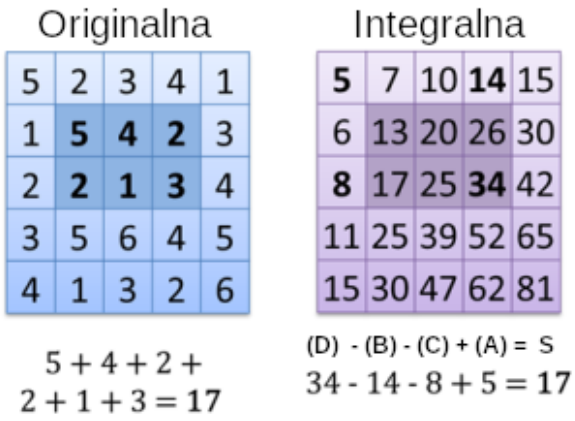
Gde je ii(x,y) integralna slika, a i(x,y) originalna slika.



Slika 1.1: Primer integralne slike

Piksele integralne slike je moguće računati u paraleli, ili sekvencijalno. Izbor algoritma za računanje integralne slike značajno utiče na performanse i potrebne hardverske resurse. U paralelnoj implementaciji cena je više pristupa memoriji i više potrebnih sabirača, dok je kod sekvencijalne implementacije manja brzina proračunavanja.

Osobina koja integralnu sliku čini pogodnu za korišćenje u Viola-Jones algoritmu je da je za računanje bilo koje pravougaone površine unutar integralne slike potrebno 2 oduzimanja i 1 sabiranje.



Slika 1.2: Primer računanja površine pravougaonika [2]

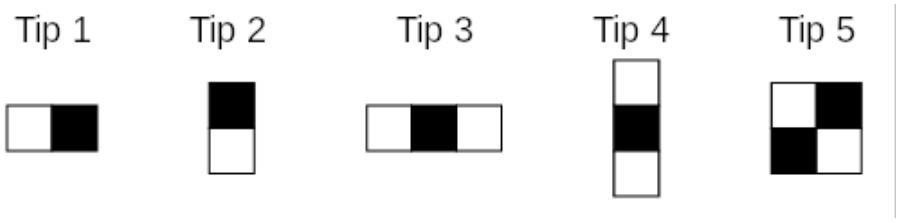
Na slici (1.2) je prikazano računanje površine pravougaonika na originalnoj slici i na integralnoj slici. Kao što se može videti za površinu pravougaonika MxN na originalnoj slici nam je potrebno MxN-1 sabiranja.
Dok je kod integralne slike broj operacija 2 oduzimanja i 1 sabiranje i ne zavisi od dimenzija pravougaonika.

$$\sum_{(x,y)\in ABCD} i(x,y) = ii(D) + ii(A) - ii(B) - ii(C)\text{[3]} \tag{1.2}$$

1.2 AdaBoost i HAAR obeležja

1.2.1 HAAR obeležja

Detekcija na slici se vrši na prozorima manjih dimenzija. Na ovim prozorima računaju se HAAR obeležja koja se sastoje od dva ili više pravougaonika, kao na slici (1.3).
Svako obeležje se računa sabiranjem piksela crnih pravougaonika potom oduzimanjem zbira piksela belih pravougaonika.
Reprezentacija prozora pomoću integralne slike omogućava da se obeležja izračunavaju u konstantnom vremenu.



Slika 1.3: HAAR obeležja [4]

Dimenzije prozora se razlikuju od modela klasifikatora. Jedan često korišćeni model koristi prozor dimezije 24x24.

Slika 1.4: Primer obeležja za detekciju lica [1]

Oblik odabranih obeležja će zavisiti od namene detektora, na slici(1.4) se mogu videti dva tipična obeležja koja su od interesa za detekciju lica.
Prvo obeležje namenjeno je merenju razlike intenziteta regiona čela i očiju. Obeležje koristi činjenicu da je oblast čela svetlija od očiju.
Dok drugo obeležje poredi intenzitet regiona mosta nosa sa očima.

Kako su obeležja od interesa koja se koriste u modelima na slici(1.3). Za datu dimenziju prozora kombinacije svih mogućih varijacija oblika i pozicija datih obeležja čini skup od 160.000 različitih obeležja. Kako je većina ovih obeležja slična i davaće slične rezultate, ovaj broj se može drastično smanjiti korišćenjem algoritma za učenje AdaBoost.

1.2.2 AdaBoost

Kako je već rečeno može se dobiti oko 160.000 obeležja za prozor dimenzije 24x24. Od ovog broja samo neka obeležja mogu dati dobre rezultate prilikom detekcije, kao na slici(1.4) u primeru detektora lica.

Kako bi se odabrao skup korisnih obeležja može se koristiti neki od algoritama mašinskog učenja. Viola i Jones predlažu modifikovani AdaBoost algoritam.

Ideja AdaBoost-a je kombinovanje više *weak learner*-a kako bi se dobila pouzdana detekcija.
Weak learner je kalsifikator koji ima pouzdanost pogađanja malo bolju nego nasumičnu. Odnosno pouzdanost *weak learner*-a mora biti bar malo iznad 50%.

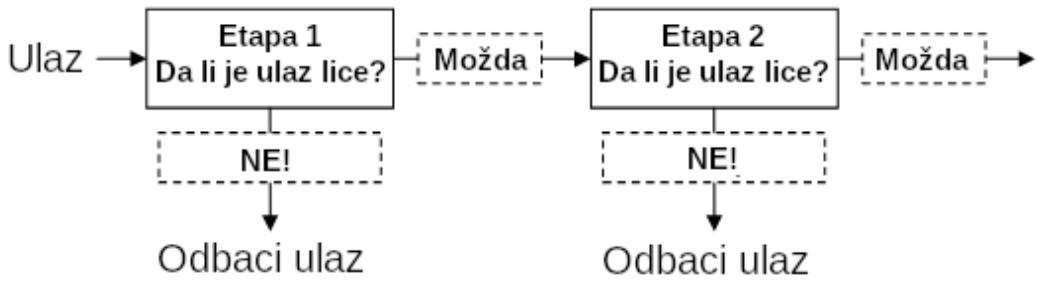
Kombinacijom ovako dobijenih *weak learner*-a može se dobiti *strong classifier*

Kao rezultat AdaBoost algoritma dobićemo skup obeležja, u ovom radu se koristi model sa skupom od 2913 obeležja.

1.3 Kaskadni klasifikator

Osnovni princip Viola-Jones algoritma je da se na osnovu svih obeležja u modelu dobije informacija da li se na trenutnom položaju prozora nalazi traženi objekat (npr. lice).
Kako na slici većina skeniranih regiona ne sadrži lice, računanje svih obeležja na svakoj poziciji bi bilo suvišno. Tako da je korišćenje jednog jakog klasifikatora neefikasno.

Ideja obrazovanja kaskadnog klasifikatora je da se prozori na kojima se očigledno ne nalazi lice odbace brzo, nakon samo nekoliko izračunatih obeležja.

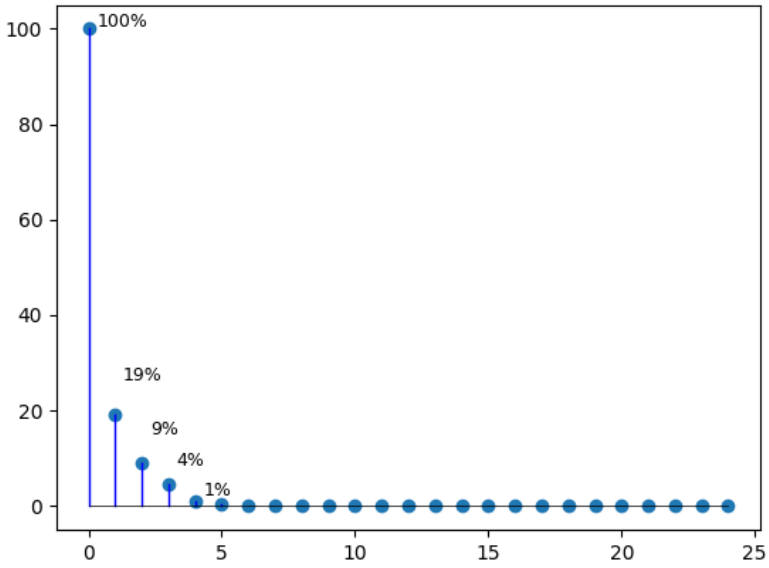


Slika 1.5: Kaskadni klasifikator [4]

Kako bi se prozori bez lica brzo odbacili predlog je da se jaki klasifikatori grupišu u etape (eng. *stage*). Svaka etapa treba da bude dobra u odlučivanju da li se na analiziranom prozoru definitivno ne nalazi lice. Ukoliko je to slučaj taj prozor će se brzo odbaciti.
Ukoliko rezultat etape ukazuje na to da se na prozoru možda nalazi lice, preći će se na izvršavanje sledeće etape.

Konačno ukoliko sve etape u klasifikatoru na analiziranom prozoru daju rezultat da se možda nalazi lice, može se zaključiti da se na toj poziciji zaista nalazi lice.
Zahvaljujući ovome postiže se veoma pouzdan klasifikator sa malim procentnom pogrešno negativnih (eng. *false negative*) rezultata na krajnjim etapama.

Kao primer u ovom radu će se koristiti model kaskadnog klasifikatora za prepoznavanje lica, sa 25 etapa i 2913 obeležja raspoređenih po etapama.
U prvoj etapi se nalazi samo 9 obeležja, dok taj broj raste do 211 u kasnijim etapama.



Slika 1.6: Procenat prolaska etapa

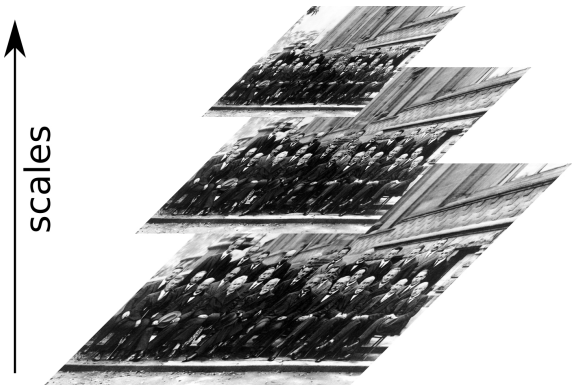
Na slici(1.6) je prikazana statistika izvršavanja etapa na *Caltech Dataset-u*[5], koji sadrži 450 slika od 27 različitih ljudi pod različitim osvetljenjima, izrazima i pozadinama.

Vrednosti različitih tačaka na grafu predstavlja procenat izvršavanja date etape na svih 450 slika. Prva etapa će se naravno uvek izvršiti, dok će se druga etapa izvršiti samo u 19% analiziranih prozora, druga 9% itd...
Vidimo da je posle pete etape procenat izvršavanja manji od 1%.

1.4 Invarijantnost veličine

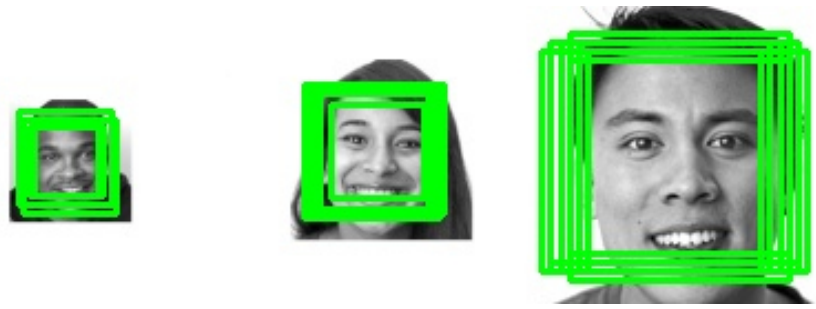
Kako lica na slikama mogu biti bliže ili dalje kameri odnosno mogu biti različitih dimenzija, potrebno je obezbediti da se ona detektuju nezavisno od veličine.

Pošto su obeležja istrenirana da detektuju samo lica koja su iste dimenzije kao i veličina obeležja potrebno je obezbediti skaliranje slike ili obeležja kako bi mogli da detektujemo lica veća od dimenzije obeležja.
Pri čemu je najmanja dimenzija lica koje je moguće detektovati jednaka dimenziji obeležja.



Slika 1.7: Piramida slike[6]

Ovo je rešeno uvođenjem skaliranja slike. Kao na slici(1.7) prvo je potrebno odraditi detekciju na originalnoj slici, zatim se slika smanjuje sa nekim faktorom i ponovo se vrši detekcija. Skaliranje se vrši sve dok je dimenzija skalirane slike veća od dimenzije obeležja.



Slika 1.8: Različite veličine lica

Na slici(1.8) može se videti rezultat klasifikatora za 3 lica različitih dimenzija.

1.5 Invarijantnost osvetljaja

Lica se mogu naći pod raznim osvetljenjima što uzrokuje problem ovom algoritmu.

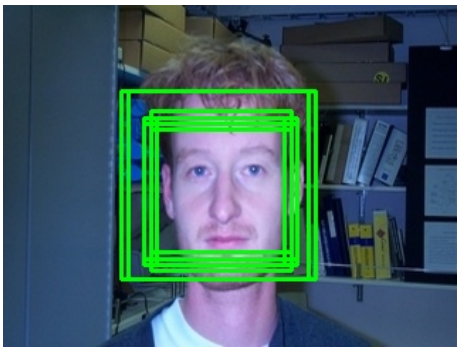
Slika 1.9: Previše osvetljaja[5]

Slika 1.10: Premalo osvetljaja[5]

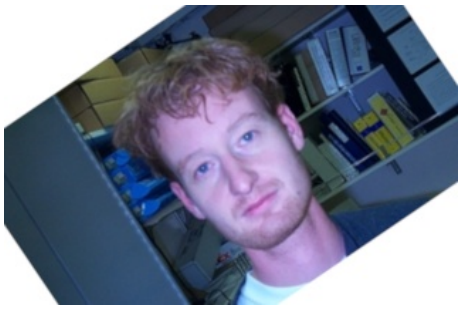
Na slikama(1.9, 1.10) su prikazane dve slike koje zbog nepovoljnog osvetljenja nisu detektovane od strane klasifikatora. Slika(1.9) nije detektovana zbog prevelikog osvetljenja lica, dok Slika(1.10) nije detektovana zbog slabog osvetljenja.

Kao delimično rešenje ovog problema uvedeno je računanje standardne devijacije prozora koji se detektuje. Ovo može poboljšati detekciju u nekim slučajevima, ali ne i ekstremnim kao u prethodnom primeru.

1.6 Varijantnost rotacije



Slika 1.11: Ne rotirana[5]



Slika 1.12: Rotirana[5]

2 OpenCV modeli

OpenCV sadrži alat za treniranje kaskada i detekciju objekata pomoću Viola-Jones algoritma. Takođe dolazi sa istreniranim i testiranim klasifikatorima[1] [7]

Rezultat treniranja se smešta u .xml fajl koji sadrži sledeće informacije:

- Dimenzija obeležja (*height, width*)
- Broj etapa (*stageNum*)
- Maksimalan broj obeležja u etapi (*maxWeakCount*)
- Informacije o etapama (*stages*)
  - Broj obeležja u etapi (*maxWeakCount*)
  - Prag etape (*stageThreshold*)
  - Informacije o obeležjima (*weakClassifiers*)
    - Prag obeležja (*internalNodes*)
    - Vrednosti listova (*leafValues*)
- Informacije o obeležjima (*features*)
  - Koordinate i težine tačaka pravougaonika (*rects*)
    Svako obeležje može imati 2 ili 3 pravougaonika
    Gde su prve 2 vrednosti x i y koordinate gornje leve tačke,
    Treća i četvrta vrednost širina i visina pravougaonika
    Poslednja vrednost težina pravougaonika

2.1 OpenCV model za frontalna lica

Često korišćeni model za detekciju lica je `haarcascade_frontalface_default.xml`[2].
Ovaj model se koristi za frontalnu detekciju lica.
Neke njegove karakteristike su:

- Dimenzija obeležja: 24x24
- Broj etapa: 25
- Maksimalan broj obeležja u etapi: 211
- Ukupan broj obeležja: 2913

Rezultati detekcije ovog modela mogu se videti na slikama(1.8,1.9,1.10,1.11,1.12) iz sekcije 1.

[1] `https://github.com/opencv/opencv/tree/master/data/haarcascades`

[2] `https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml`

Deo II
Projekat

3 Sažetak

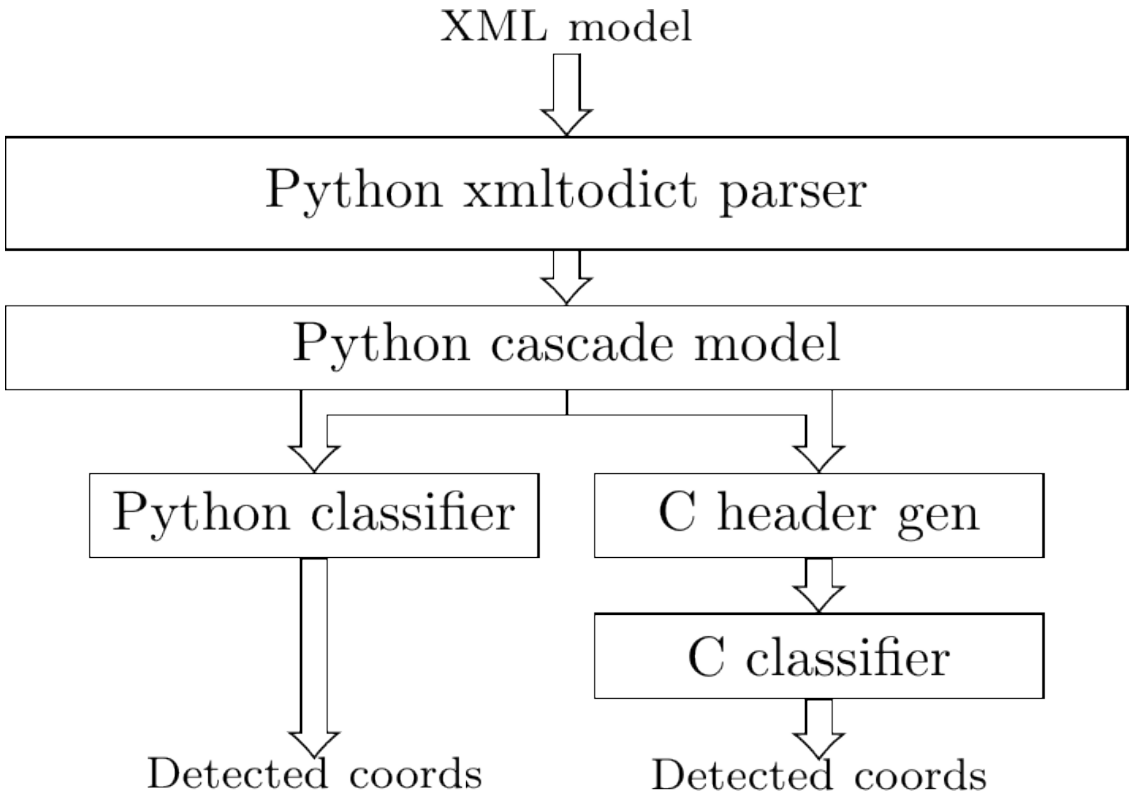
Ovaj projekat sadrži:

- Pisanje specifikacije u Python i C programskom jeziku za izvršavanje i pomoć pri particionisanju i projektovanju hardvera.
- Projektovanje arhitekture hardverskog akceleratora za Viola-Jones algoritam opisam u delu I.
- Pisanje HDL modela za sintezu u SystemVerilog RTL metodologiji i PyGears[3] metodologiji.
- Pisanje verifikacionog okruženja u SystemVerilog UVM[4] i Python PyGears okruženju.
- Poređenje dve metodologije i analiza prednosti i mane obe metodologije.
- Poređenje komercijalnog Questa Sim[5] simulatora i besplatnog open-source Verilator[6] simulatora.
- Implementacija projektovanog IP jezgra na MYIR Z-Turn Board[7] sa Zynq 7020 SoC.
- Pisanje Linux Kernel drajvera i korisničke aplikacije za korišćenje jezgra za detekciju lica na Xilinx Zynq platformi.

[3] https://github.com/bogdanvuk/pygears
[4] https://www.accellera.org/downloads/standards/uvm
[5] https://www.mentor.com/products/fv/questa/
[6] https://www.veripool.org/wiki/verilator
[7] http://www.myirtech.com/list.asp?id=502

4 Specifikacije za izršavanje

Kako bi se hardver efikasno projektovao i particionisao potrebno je početi od softverske implementacije na višem nivou apstrakcije.
Jezici visokog nivoa kao što je Python su veoma dobri za ovu namenu.



Slika 4.1: Veza Python modela sa XML modelom i C specifikacijom

Na slici(4.1) prikazana je struktura koda u slučaju Python i C klasifikatora.

Na ulazu se nalazi XML model dobijen treningom pomoću OpenCV opisan u sekciji 2.
Kao jezik za parsiranje XML fajla se koristi Python. Zbog velikog broja Python paketa dostupnih sa gotovim rešenjima za većinu softverskih problema, problem parsiranja XML fajla se može rešiti korišćenjem paketa xmltodict[8].
Xmltodict parsira XML fajl i skladišti ga u Python dictionary.
Klase *CascadeClass*, *StageClass*, *FeatureClass* i *RectClass* [9] napisane u Pythonu predstavljaju Python model kaskadnog klasifikatora.

Napisan je i Python specifikacija za isvršavanje koja direktno koristi Python klase za detekciju objekata.
Veza između Python modela i C specifikacije realizovana je preko *C Header* fajla koji je generisan pomoću Python-a.

[8] `https://pypi.org/project/xmltodict/`
[9] `cascade_classifier/python_model/cascade.py`

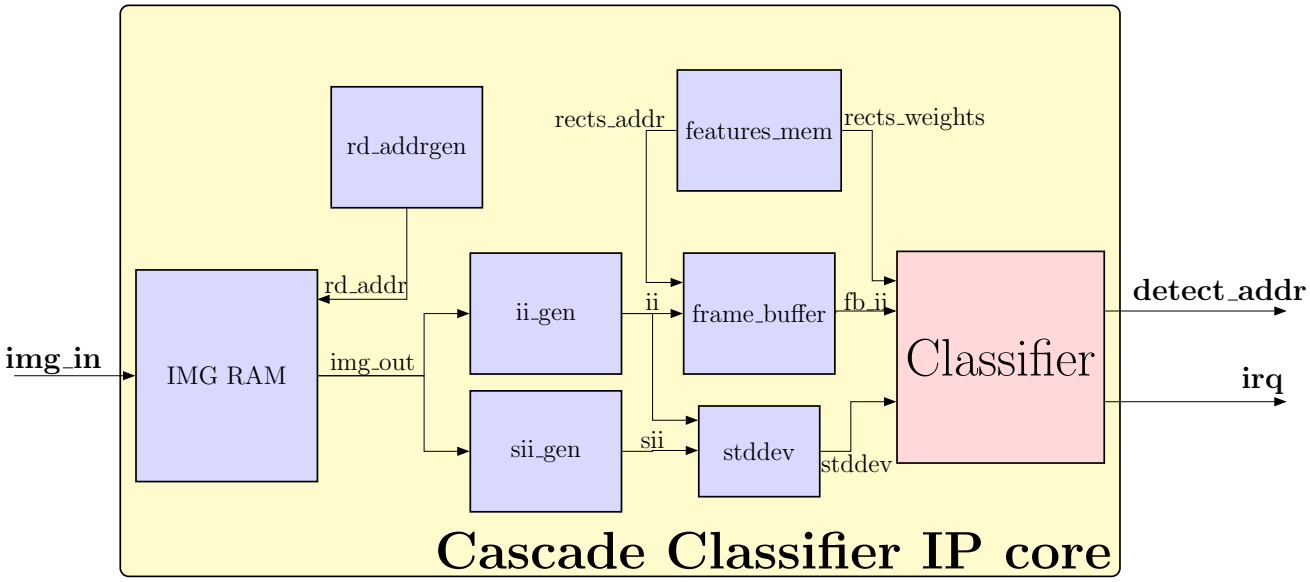
5 Arhitektura hardvera

5.1 Uvod

U ovom poglavlju biće opisana arhitektura hardvera koja je projektovana.
Postoje razlike između implementiranih arhitektura u SystemVerilog-u i PyGears-u.
Pošto obe metodologije sa sobom nose neke karakteristike koje otežavaju ili olakšavaju određene principe prilikom projektovanja, ove razlike će biti opisane u kasnijim poglavljima.

Iako su razlike između ove dve implementacije male opis u ovom poglavlju će biti bliži implementaciji u PyGears-u koja je novija i ispravljene su neke sistemske greške.

Arhitektura će biti opisana na nivou modula od kojih se sastoji njihovih portova, funkcija koje obavljaju i eventualno kritičnih funkcionalnih i memorijskih jedinica.



Slika 5.1: Arhitektura hardvera kaskadnog klasifikatora

Na slici(5.1) prikazan je uprošćen blok dijagram realizovanog IP jezgra.

5.2 Interfejsi IP jezgra

IP jezgro se povezuje pomoću 3 interfejsa:

- Ulazni interfejs **img_in** sastoji se od 8-bitnog podatka vrednosti piksela slike u *grayscale* formatu (nijanse sive).
- Izlazni interfejs **detect_addr** ukoliko jezgro detektuje lice na slici postaviće x i y koordinate na ovom interfejsu.
- Izlazni interfejs **irq** je signal koji označava završetak obrade slike i signalizira da je jezgro spremno za novu sliku. Namenjen da se koristi kao prekidni signal za procesor.

5.3 Modul IMG RAM

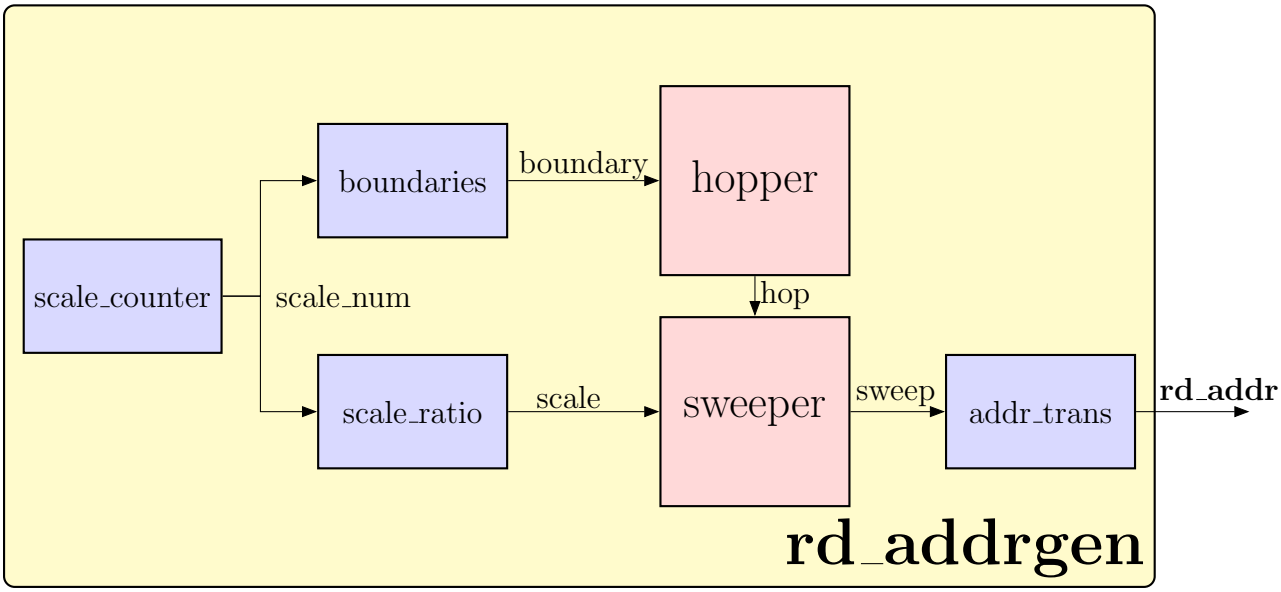
Modul **IMG RAM** je zadužen za skladištenje slike koja se obrađuje. Sastoji se od RAM memorije i brojača za generisanje adrese za upis.
Nakon primljenog podatka na ulazu brojač adrese upisa se poveća za 1.
Skladištena slika je u 8-bitnom formatu i predstavlja grayscale vrednost piksela originalne slike.
Veličina memorije je $width * height * 8$ bit. Za dimenziju slike 240x320 veličina memorije je 614400 bita.

Komunicira sa okolinom pomoću 3 interfejsa:

- Ulazni interfejs **img_in** ekvivalentan je sa istoimenim interfejsom na višem nivou hijerarhije.
- Ulazni interfejs **rd_addr** predstavlja linearnu adresu generisanu od strane **rd_addrgen** modula. Koristi se za čitanje podataka iz RAM memorije.
- Izlazni interfejs **img_out** predstavlja iščitane podatke iz RAM memorije.

5.4 Modul rd_addrgen

Modul **rd_addrgen** je zadužen za generisanje adrese za čitanje iz **img_ram** modula. Blok dijagram ovog modula prikazan je na slici (5.2).



Slika 5.2: Blok dijagram **rd_addrgen** modula

Pored generisanja adrese pomoću ovog modula je dodatno rešeno i skaliranje slike opisano u sekciji 1.4.

5.4.1 Scale_counter

Scale_counter je brojač koji se poveća za jedan kada klasifikator završi obradu slike na trenutnom nivou piramide prikazanoj na slici (1.7) sekciji 1.4.

5.4.2 Boundaries

Boundaries kao ulaz dobija trenutni stepen skaliranja **scale_num** i na osnovu njega na izlazu daje **X** i **Y** granice do kojih **hopper** treba da broji. Ove granice osiguravaju ispravno generisanje adrese, odnosno obezbeđuju da izlazna adresa nikad ne iskoči iz opsega **img_ram** memorije.
Realizovan je pomoću multipleksera i softverski izračunatih konstantnih vrednosti granica.

5.4.3 Scale_ratio

Scale_ratio ima ulogu da dostavi **sweeper**-u potrebne parametre za računanje skalirane adrese. Realizovan je isto kao i **boundaries** modul.

5.4.4 Hopper

Hopper se može zamisliti kao dvostruka ugnježdena for petlja gde iteratori petlje predstavljaju koordinate **X** i **Y**.
Prvo se iterira po **X** kordinati pa po **Y**.

```
1  for(int y = 0; y < boundary_y[scale_num]; y++){
2    for(int x = 0; x < boundary_x[scale_num]; x++){
3      // Sweeper
4    }
5  }
```

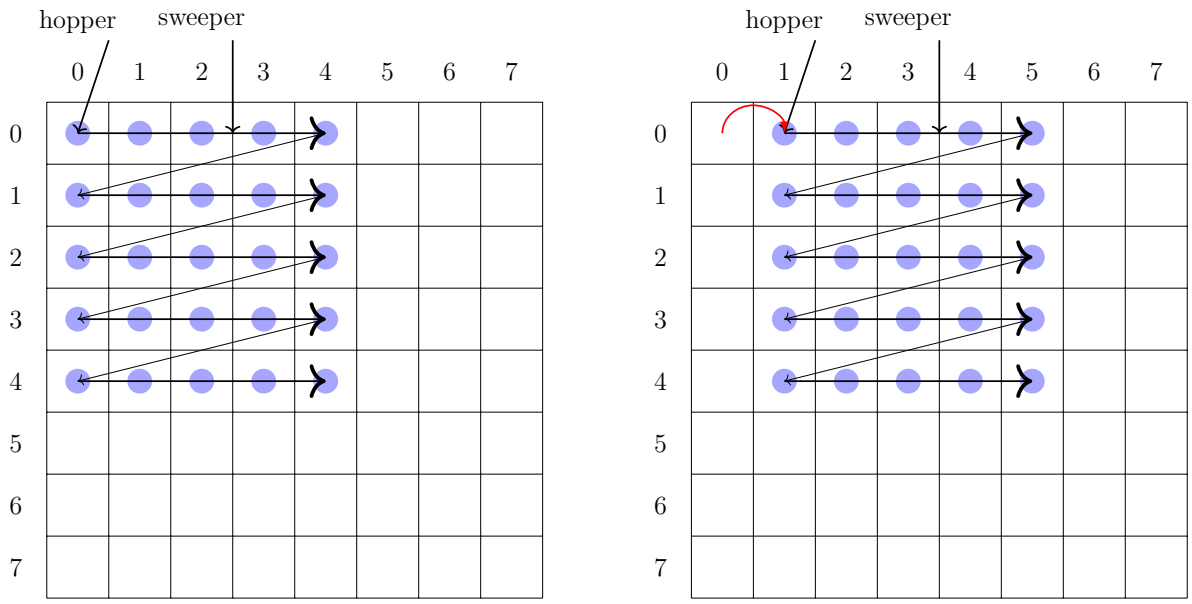
Primer 5.1: Primer **hopper**-a u **C**-u

Na primeru (5.1) prikazana je implementacija **hopper**-a u **C**-u. Može se videti da granice **hopper**-a zavise od promenljivih **boundary_x** i **boundary_y** koji se indeksiraju preko **scale_num** promenljive opisane u sekcijama (5.4.2, 5.4.1).

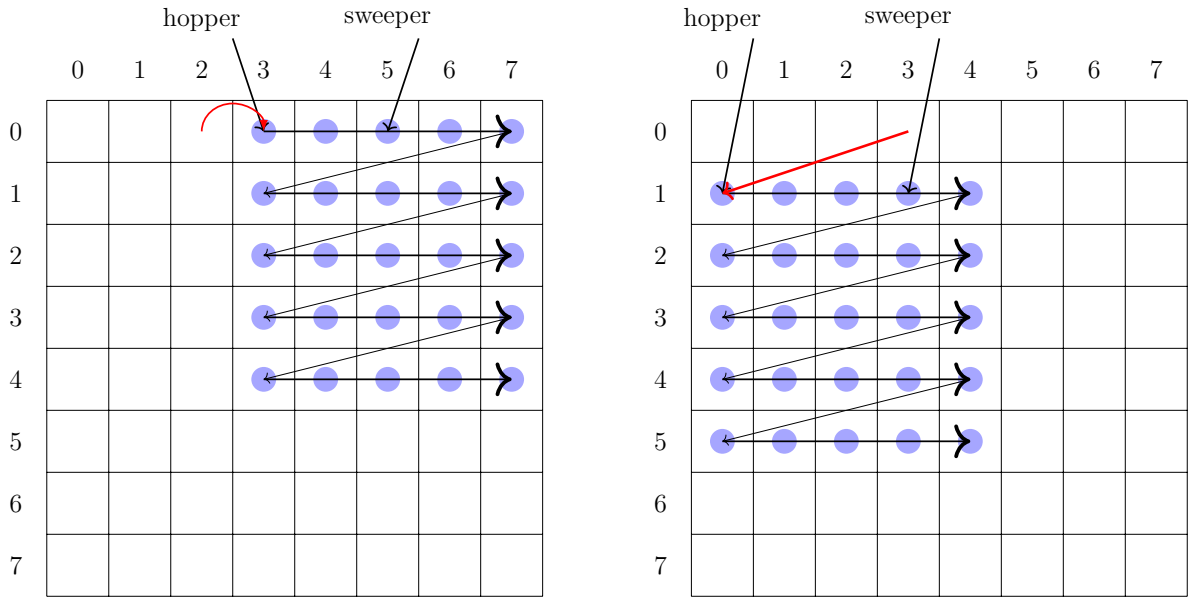
Na slikama (5.3, 5.4) je prikazan rad hopper-a i sweeper-a. Plavi kružići predstavljaju piksele za koje će rd_addrgen generisati adresu za trenutno stanje hopper-a.
Gornji levi kružić je koordinata trenutnog položaja hopper-a. Crevnom strelicom je obeleženo iteriranje hopper-a kroz sliku.
Na slici (5.4) može se videti trenutak kada hopper dostiže granicu boundary_x nakon čega

prelazi u novi red, uvećava Y koordinatu a X koordinatu postavlja na početni položaj.

Nakon što hopper dostigne obe granice boundary_x i boundary_y završen je trenutni stepen skaliranja slike i potrebno je uvećati scale_num za jedan.



Slika 5.3: Način rada **hopper** i **sweeper** komponenti.



Slika 5.4: Prelazak **hopper**-a u novi red.

5.4.5 Sweeper

Slično kao hopper i sweeper se može predstaviti kao dve ugnježdene for petlje. Ponovo se prvo iterira po X koordinati pa onda po Y.

```
1   int x, y;
2   // hopper
3   for(int hop_y = 0; hop_y < boundary_y[scale_num]; hop_y++){
4     for(int hop_x = 0; x < boundary_x[scale_num]; hop_x++){
5       // sweeper
6       for(y = hop_y; y < hop_y+feature_height; y++){
7         for(x = hop_x; x < hop_x+feature_width; x++){
8           // scale address
9           // translate to linear address
10        }
11      }
12    }
13  }
```

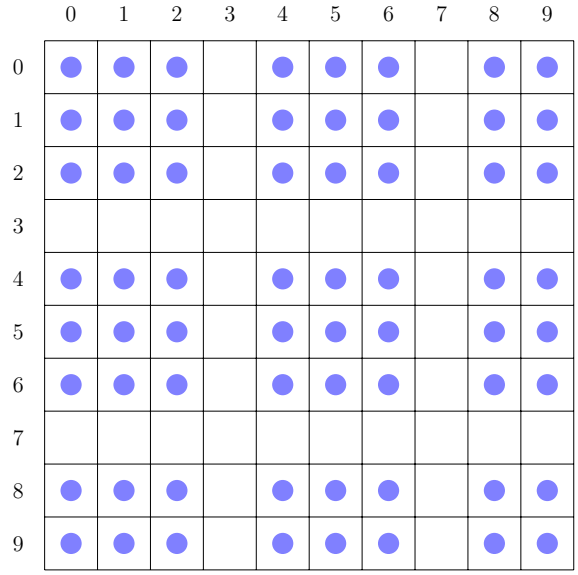
Primer 5.2: Primer **sweeper**-a u **C**-u

Kao što se vidi na primeru (5.2) hopper i sweeper se zajedno mogu predstaviti kao četiri ugnježdene for petlje.
Sweeper će početnu vrednost svojih X i Y promenljivih uzeti od trenutne vrednosti hopper koordinata, zatim će se iterirati feature_width i feature_height puta.
Na slikama (5.3, 5.4) prelazak sweeper-a po slici je prikazan horizontalnim i dijagonalnim strelicama. Dok je plavim kružićima predstavljeni pikseli koje sweeper prebriše za jedan položaj hopper-a.

5.4.6 Skaliranje adrese

Unutar sweeper-a je implementirano i skaliranje adrese u svrhu skaliranja slike objašnjeno u sekciji 1.4.
Potrebno je množiti adresu sa decimalnim faktorom (npr. 1.2, 1.33), kako je u hardveru množenje sa decimalnim brojevima sa pokretnom tačkom skupa operacija, efikasnije je odraditi množenje sa fiksnom tačkom.
To je odrađeno tako što celobrojni faktor unapred softverski izračunat i smešten u scale_ratio modul, isto tako je i broj pomeranja u desno dobijene binarne vrednosti unapred poznat. Pa je jednostavno odraditi množenje sa fiksnom tačkom.

Ovako skalirana adresa prikazana je na slici (5.5). Može se videti da će u ovom slučaju svaka četvrta tačka biti preskočena. Na taj način će se dobiti manja slika od originalne, u ovom primeru od početne slike $10 * 10$ dobija se slika $8 * 8$.
Na taj način objekti koji su izgledali veći na originalnoj slici će izgledati manji na skaliranoj slici, što nam je potrebno kako bi dobili invarijantnost veličine opisane u sekciji 1.4.



Slika 5.5: Posledica skaliranja adrese.

5.4.7 Modul addr_trans

Konačno je potrebno konvertovati adresu predstavljenu koordinatama (y, x) u linearnu adresu, pošto se RAM memorija adresira linearno. Ovo obavlja addr_trans u hardveru.
Translaciju je jednostavno uraditi pomoću seledeće formule.

$$lin\_addr = (y * img\_width) + x \tag{5.1}$$

Gde su y i x koordinate iz sweeper-a a img_width je parametar koji označava širinu slike.

5.5 Modul ii_gen i sii_gen

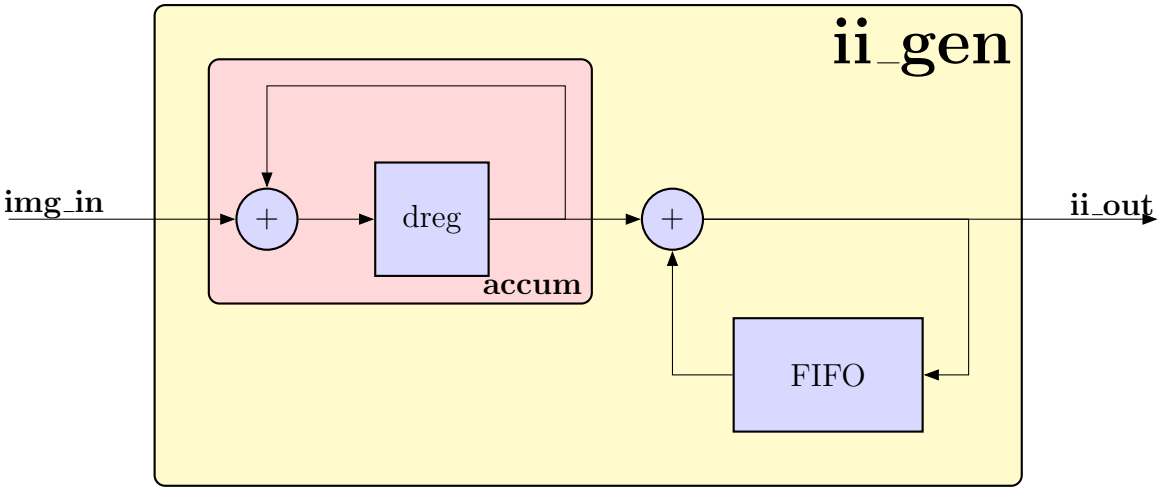
5.5.1 Odabir algoritma

Jedan od kritičnih delova Viola-Jones algoritma je generisanje integralne slike opisane u poglavlju 1.1.
Isto tako se ispostavlja da i u hardverskoj implementaciji generisanje integralne slike ima veliki uticaj na performanse i potrebne hardverske resurse sistema.

Kao izbor možemo izabrati sekvencijalni ili paralelni algoritam.
Ukoliko bi se odabrao paralelni algoritam koji može da računa više piksela u paraleli povećanje resursa bi se drastično odrazilo na img_ram i frame_buffer memorije. Ali bi dobili bolje performanse sistema.
Kako bi implementacija paralelnog algoritma povećala kompleksnost ne samo ovog modula nego i okolnih komponenti, u ovom projektu on neće biti razmatran.

5.5.2 Sekvencijalna implementacija generatora integralne slike

U ovoj arhitekturi odabran je sekvencijalni algoritam generisanja integralne slike koji odgovara jednačini(5.1) iz sekcije 1.1.
Prednosti ovog algoritma u odnosu na paralelni je manji memorijski zahtevi na ulazu i izlazu, potrebno manje funkcionalnih jedinica i unutrašnje memorije. A mana je manja brzina.
Konkretno ovaj algoritam može da izračuna jedan piksel svaki takt.



Slika 5.6: Blok dijagram ii_gen modula

Na slici(5.6) prikazana je uprošćena šema generatora integralne slike.

Na ulazu module je port img_in koji predstavlja piksele slike pročitane iz img_ram memorije.

Pikseli u redu se akumuliraju pomoću sabirača i registra unutar accum modula. Na slici je izostavljeno da se registar dreg resetuje posle dolaska poslednjeg piksela u redu slike.

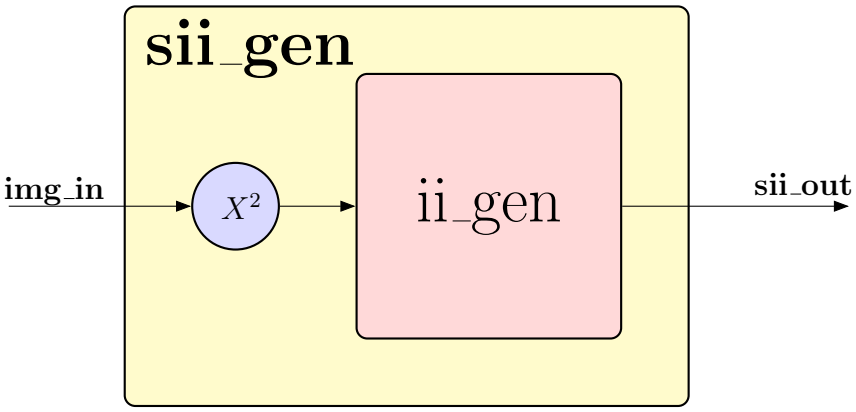
Nakon toga akumulirana vrednost se sabira sa vrednošću FIFO bafera koji sadrži vrednost piksela integralne slike iz predhodnog reda. Zatim prosleđuje na izlaz i ponovo upisuje u FIFO bafer kao vrednost izračunate integralne slike.

Zbog potreba algoritma FIFO bafer je modifikovan na sledeći način:

- Dodat je PRELOAD parametar koji pomera pokazivač za upis na vrednost širine prozora (feature_width) prilikom reseta. Ovo je potrebno da bi se obezbedilo čitanje nula iz bafera kada se obrađuje prvi red slike.
- FIFO se resetuje kada je završeno računanje celog prozora.

5.5.3 Generator kvadratne integralne slike

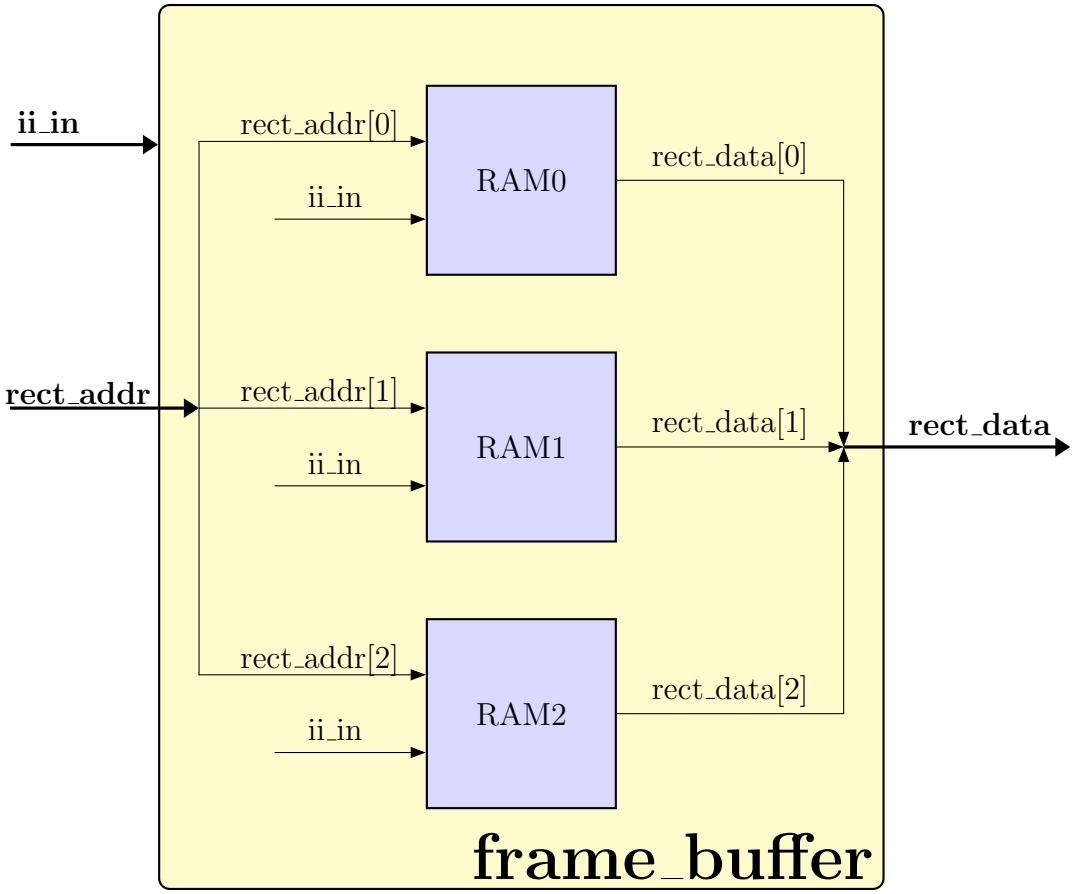
Generator kvadratne integralne slike je potreban za računanje standardne devijacije prozora što je opisano u sekciji 1.5.
Generisanje kvadratne integralne slike je jednostavno uz gotov generator integralne slike. Potrebno je kvadrirati ulazne piksele i dovesti ih na generator integralne slike kao na slici(5.7).



Slika 5.7: Blok dijagram ii_gen modula

5.6 Modul frame_buffer

Potrebno je obezbediti da se generisana integralna slika može pročitati od strane klasifikatora u nasumičnom maniru. Kako bi se to obezbedilo potrebno je skladištiti integralnu sliku u lokalnu RAM memoriju.



Slika 5.8: Blok dijagram frame_buffer-a realizovanog sa 3 jedno-portne RAM memorije.

U ovu svrhu je projektovana komponenta frame_buffer. Sastoji se od brojača za adresu upisa i RAM memorije. Brojač adrese upisa se inkremetuje za jedan nakon svakog primljenog podatka.
Potrebna veličina RAM memorije je data sledećim vezama:

$$size(bit) = frame\_width * frame\_height * w\_ii \tag{5.2}$$

$$w\_ii = ceil(log_2(frame\_width * frame\_height * 2^{w\_img})) \tag{5.3}$$

Gde je frame_width i frame_height širina i visina obeležja modela, w_ii je širina magistrale integralne slike, a w_img ulazna širina magistrale piksela slike.

Radi ubrzavanja rada klasifikatora možemo računati sva tri pravougaonika (1.2.1) u paraleli. Da bi se to obezbedilo potrebno je čitati u istom taktu tri vrednosti iz frame_buffer memorije. Kako je više-portna memorija skupa i retko se nalazi u FPGA čipovima moguće rešenje je koristiti tri jedno-portne memorije. Ovo će kao rezultat zauzeti tri puta više RAM memorije na čipu.

Alat za sintezu uglavnom može da odradi transformaciju i da od jedne tro-portne memorije napravi tri jedno-portne. Ali se preporučuje da se ekslicitno instancioniraju tri memorije i pridržava Synthesis Guideline-a npr. Xilinx [8].
Primer prikazan na slici(5.8).

5.7 Modul stddev

Računanje standardne devijacije prozora je potrebno kako bi se smanjio uticaj različitog osvetljenja lica na slikama.
Za ovo je zadužen modul stddev. U sledećem primeru prikazano je računanje standardne devijacije u C-u.

```
1   long calcStddev(long sii[FRAME_HEIGHT][FRAME_WIDTH],
2                   long ii[FRAME_HEIGHT][FRAME_WIDTH]){
3
4     long mean, stddev;
5
6     mean = ii[0][0] + ii[FRAME_HEIGHT-1][FRAME_WIDTH-1] - ii
         [0][FRAME_WIDTH-1] - ii[FRAME_HEIGHT-1][0];
7
8     stddev = sii[0][0] + sii[FRAME_HEIGHT-1][FRAME_WIDTH-1] -
          sii[0][FRAME_WIDTH-1] - sii[FRAME_HEIGHT-1][0];
9
10    stddev = (stddev * (FRAME_WIDTH-1)*(FRAME_HEIGHT-1));
11    stddev = stddev - (mean*mean);
12    stddev = getSqrt(stddev);
13    return stddev;
14  }
```

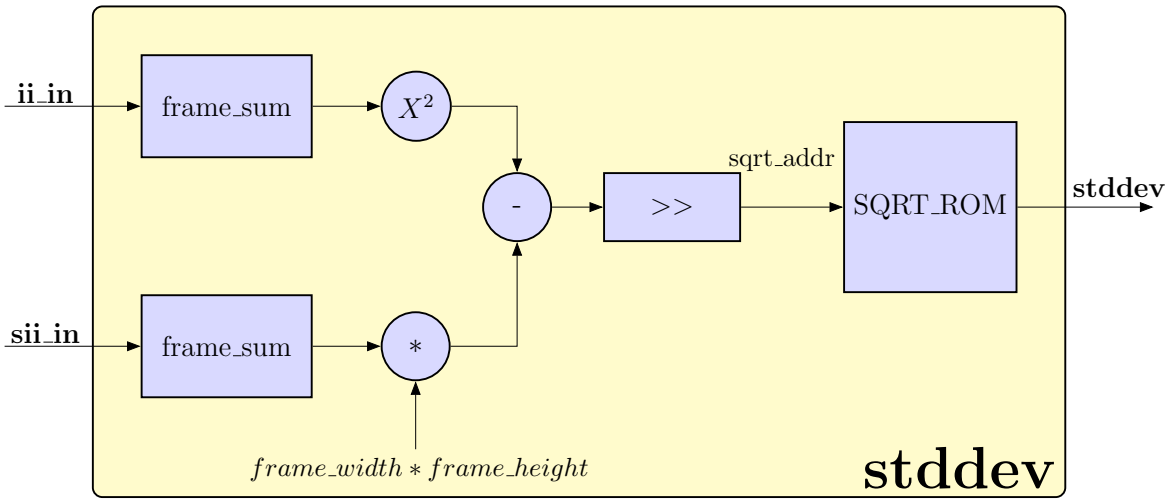
Primer 5.3: Primer računanja standardne devijacije u **C**-u

Za računanje standardne devijacije potrebni su nam ivice prozora integralne i kvadratne integralne slike, što se vidi u liniji 6 i 8 primera(5.3).
Sabiranjem gornjeg levog i donjeg desnog piksela zatim oduzimanje gornjeg desnog i donjeg levog integralne slike dobijamo sumu piksela celog prozora.
Ako pogledamo primer u C-u vidimo da imamo operacije sabiranja, oduzimanja, kvadriranja promenljivih, zatim množenje sa konstantom.
Sa ovim operacijama većina alata za sintezu nema problem prilikom mapiranja i većina

FPGA čipova ima potrebne funkcionalne jedinice.

Konačno potrebno je odraditi kvadratni koren u liniji 12.
Ovo je operacija koju većina alata za automatsku sintezu ne mogu implementirati na FPGA čipovima, pa je potrebno pronaći dobru aproksimaciju.

U ovom projektu je odlučeno koristiti lookup tabelu.
Za unapred definisani opseg vrednosti operanda za korenovanje softverski su izračunate vrednosti kvadratnog korena i smeštene u listu.
Prilikom softverske analize odlučeno je da je 256 vrednosti korena dovoljno za ispravan rad celog sistema, uz minimalan gubitak pouzdanosti.



Slika 5.9: Blok dijagram stddev modula.

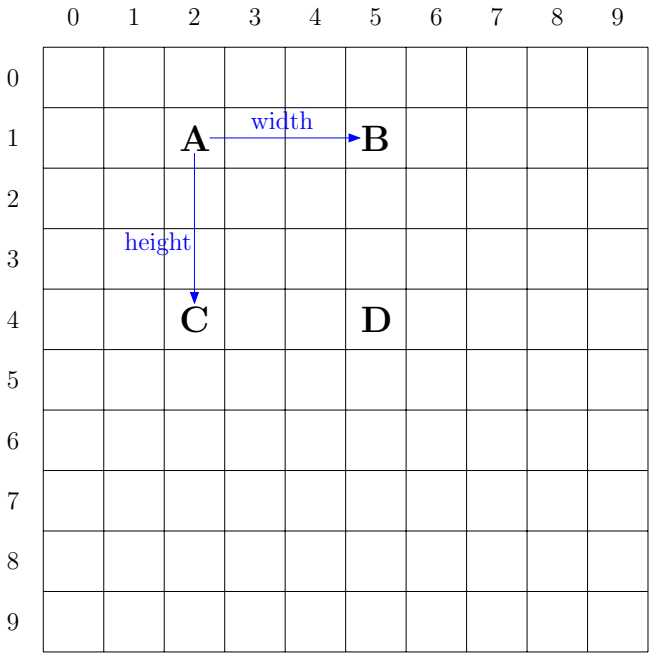
Na slici(5.9) prikazan je blok dijagram projektovanog hardverskog modula na osnovu primera(5.3). Operaciju sabiranja piksela unutar prozora obavljaju frame_sum moduli. Mogu se videti i operator kvadriranja, množenja zatim i oduzimanja kao u prethodnom primeru.
Lookup tabela je implementirana kao ROM memorija, nazvana SQRT_ROM sa 256 lokacija.

5.8 Modul features_mem

Svako obeležje u modelu se sastoji od najviše tri pravougaonika što je objašnjeno u sekciji 1.2.1.
Svaki pravougaonik je definisan sa četiri koordinate (A, B, C, D) i sa težinom njegove površine.

Zbog uštede memorije u hardverskoj implementaciji, moguće je svaki pravougaonik predstaviti koordinatom jedne njegove tačke zatim širinom i visinom pravougaonika. Ostale tačke je moguće izračunati pomoću ovih podataka.
Na ovaj način se vrši ušteda memorije, ali se uvodi potreba za množačima i sabiračima u

hardveru. U ovom slučaju zbog velikog broja obeležja (2913 u testiranom modelu 2.1) ušteda memorije je značajna, a dodatni sabirači i množači ne unose veliku cenu u sistem.



Slika 5.10: Reprezentacija pravougaonika u obeležju.

Sa slike(5.10) mogu se videti potrebni parametri za reprezentaciju pravougaonika.
Kao referentu koordinatu izabrana je tačka A od koje se meri širina i visina pravougaonika kao na slici.
Kako je za adresiranje podatka iz RAM-a potrebna linearna adresa kao što je objašnjeno u sekciji(5.4.7) koordinata tačke A je linearizovana u softveru.
Ostale koordinate moguće je dobiti na sledeći način:

$$B = A * width \tag{5.4}$$

$$D = (A + width) + (height * FRAME\_WIDTH) \tag{5.5}$$

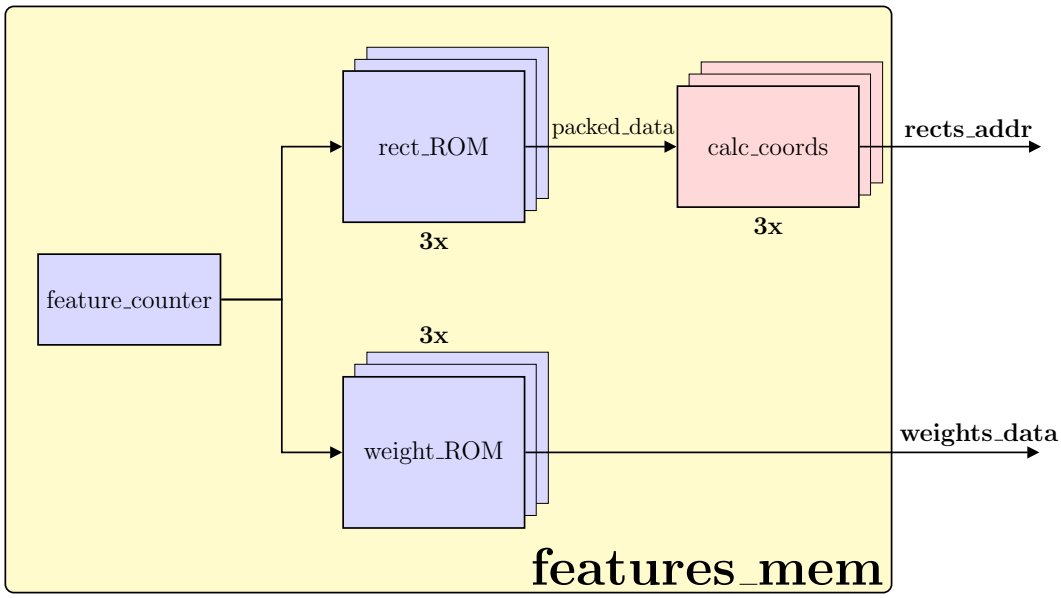
$$C = (A + width) + (height * FRAME\_WIDTH) - width \tag{5.6}$$

N	Packed value(20 bit)	Height (5bit)	Width (5bit)	A linear coord (10bit)
0	0x1A989	9	12	106
1	0x1A987	7	12	106
2	0x39249	9	18	228
3	0x72926	19	9	458
. . .	. . .	. . .	. . .	. . .
feature num - 1	0x088d6	22	6	34

Tabela 5.1: Struktura memorije rect_ROM0 za model 2.1

Na tabeli(5.1) je prikazana strukura zapakovane memorije rect_ROM
U prvoj koloni **N** su adrese lokacija, kojih ima features_num (2913 u modelu 2.1).
U drugoj koloni **Packed value(20 bit)** je zapakovana vrednost memorijske lokacije na adresi u heksadecimalnom zapisu.
U trećoj i četvrtoj koloni se nalaze **height** i **width** raspakovane vrednosti visine i širine pravougaonika.
U poslednjoj koloni se nalazi **A linear coord** raspakovana vrednost linearne koordinate A.

Pored rect_ROM memorije potrebna je weight_ROM memorija koja će čuvati vrednosti težina za svaki pravougaonik.



Slika 5.11: Blok dijagram features_mem modula.

Na slici(5.11) je prikazan uprošćen blok dijagram features_mem modula.
Komponenta **feature_counter** generiše adresu za čitanje iz ROM memorija.

Postoje po 3 **rect_ROM** i **weight_ROM** memorije prethodno opisane, po jedna za svaki pravougaonik u obeležju.
Komponenta **calc_coords** vrši raspakivanje memorije opisane u tablici(5.1) na način opisan formulama(5.4, 5.5, 5.6).

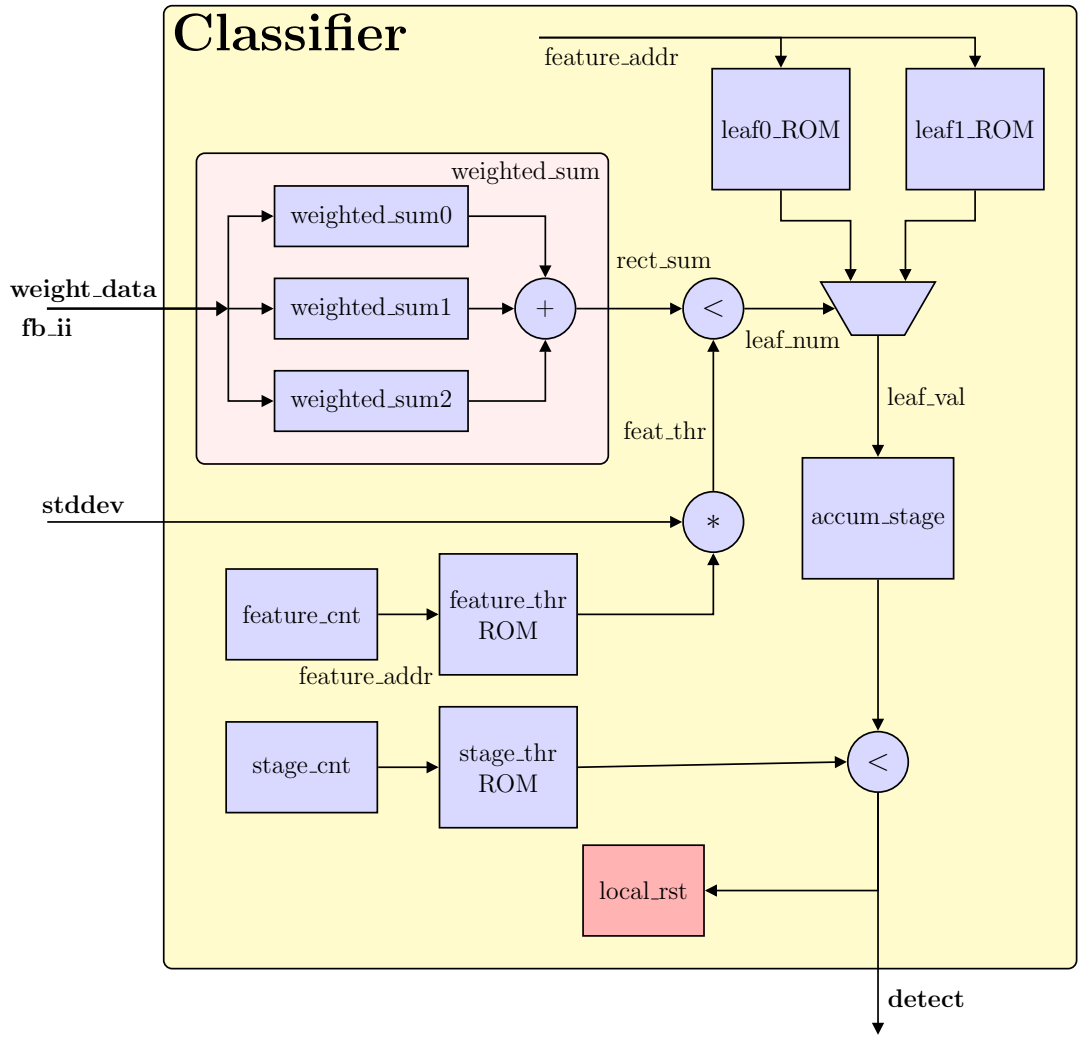
Konačno može se izračunati potrebna memorija u slučaju modela opisanog u sekciji(2.1).

feature_num	2913
rect_num	3
w_weight	3 bits
w_rect	20 bits
TOTAL	524340 bits

Tabela 5.2: Veličina memorije features_mem za model 2.1

5.9 Modul classifier

Classifier modul obavlja klasifikaciju prozora, odnosno signalizira da li se na trenutnom prozoru nalazi traženi objekat. Zauzima najviše hardverskih resursa u sistemu i pored ii_gen(5.5) modula najviše utiče na performanse sistema.



Slika 5.12: Blok dijagram classifier modula.

Sledeći primeri u C-u približno opisuje algoritam rada klasifikatora.

```
1   int weights[RECT_NUM][FEATURE_NUM]; //feature weights
2   int rects[RECT_NUM][FEATURE_NUM][4];//unpacked rect_coords
3
4   long weighted_sum_i(int ii[FRAME_HEIGHT][FRAME_WIDTH],
5                       int f,       // feature_num
6                       int r){      // rect_num
7
8     long sum = ii[rects[r][f][0][1]][rects[r][f][0][0]] +
9                ii[rects[r][f][3][1]][rects[r][f][3][0]] -
10               ii[rects[r][f][2][1]][rects[r][f][2][0]] -
11               ii[rects[r][f][1][1]][rects[r][f][1][0]];
12    sum *= weights[r][f];
13
14    return sum;
15  }
16  long weighted_sum(int ii[FRAME_HEIGHT][FRAME_WIDTH],
17                    int feature_num){
18    long sum0 = weighted_sum_i(ii, feature_num, 0);
19    long sum1 = weighted_sum_i(ii, feature_num, 1);
20    long sum2 = weighted_sum_i(ii, feature_num, 2);
21
22    return sum0 + sum1 + sum2;
23  }
```

Primer 5.4: Weighted_sum u **C**-u

Primer 5.4 približno prikazuje algoritam rada weighted_sum komponente sa slike(5.12). Memorije weights i rects se nalaze u features_mem komponenti u hardveskoj implementaciji i objašnjene su u sekciji(5.8).

Funkcija weighted_sum_i() računa sumu piksela pravougaonika na integralnoj slici. Kao na slici(1.2) na integralnoj slici potrebno je sabrati gornji levi i donji desni piksel zatim oduzeti gornji desni i donji levi piksel.
Pošto svaki pravougaonik ulazi u konačan zbir sa nekom težinom potrebno je pomnožiti sumu pravougaonika sa težinom kao u liniji 12.

Funkcija weighted_sum() dodatno sabira težinske sume sva tri pravougaonika.

```
1   int feature_thresholds[FEATURE_NUM];
2   int leaf_val0[FEATURE_NUM];
3   int leaf_val1[FEATURE_NUM];
4
5   int leaf_val(long stddev,
6                long sum,
7                int feature_num){
8
9     if(sum <= feature_thresholds[feature_num] * stddev)
10        return leaf_val0[feature_num];
11    else
12        return leaf_val1[feature_num];
13  }
```

Primer 5.5: Leaf_val u **C**-u

Primer(5.5) prikazuje algoritam računanja leaf_val vrednosti.
Memorija feature_thresholds se nalazi na slici(5.12) pod nazivom feature_thr ROM.
Memorije leaf_val0 i leaf_val1 su leafVal0 i leafVal1 na slici(5.12).

Funkcija leaf_val kao ulaz prima standardnu devijaciju prozora, težinsku sumu sva tri pravougaonika i broj trenutnog obeležja.
Povratna vrednost ove funkcije je sadržaj jedne od memorija leaf_val0 ili leaf_val1 za to obeležje.

Ukoliko je težinska suma manja od praga obeležja feature_threshold pomnoženog sa standardnom devijacijom, povratna vrednost će biti iz memeorije leaf_val0, u suprotnom povratna vrednost će biti iz memorije leaf_val1.

Literatura

[1] P. A. Viola and M. J. Jones, "Rapid object detection using a boosted cascade of simple features," in *CVPR*, 2001.

[2] A. Jain, "Computer vision – face detection," 2016. [Online]. Available: https://vinsol.com/blog/2016/06/28/computer-vision-face-detection/

[3] K. Cen, "Study of viola-jones real time face detector," 2016.

[4] O. Jensen, "Implementing the viola-jones face detection algorithm," 2008.

[5] M. Weber, "Frontal face dataset," 1999. [Online]. Available: www.vision.caltech.edu/Image_Datasets/faces/faces.tar

[6] Z. Ye, "5kk73 gpu assignment 2012," 2012. [Online]. Available: https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection

[7] OpenCV, "Opencv docs." [Online]. Available: https://docs.opencv.org/3.4.3/d7/d8b/tutorial_py_face_detection.html

[8] Xilinx, "Xst user guide." [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx10/books/docs/xst/xst.pdf