Hardverska implementacija Viola-Jones algoritma

Kandidat
Risto Pejašinović EE19/2015

Mentor

29. septembar 2019

Sadržaj

Mali rečnik

Akronim		**Opis**
AdaBoost	=	Adaptive Boosting
AXI	=	Advanced eXtensible Interface
DDR	=	Double Data Rate
DTI	=	Data Transfer Interface
EMIO	=	Extended multiplexed I/O
ESL	=	Electronic System Level Design and Verification
FPGA	=	Field Programmable Gate Array
FSMD	=	Finite State Machine with Datapath
HDL	=	Hardware Description Language
PL	=	Programmable Logic
PS	=	Processing System
RAM	=	Random Access Memory
ROM	=	Read Only Memory
RTL	=	Register Transfer Methodology
SoC	=	System on Chip
SV	=	System Verilog
XML	=	eXtensible Markup Language

Deo I
Viola-Jones algoritam

1 Uvod

Viola-Jones algoritam za detekciju i lokalizaciju objekata na slici razvijen je od strane Paul Viola i Michael Jones 2001. godine [1].
Zbog brze i pouzdane detekcije jedan je od nakorišćenijih algoritama za detekciju lica na slici. Iako danas neuronske mreže polako zamenjuju tradicionalne algoritme za detekciju objekata, Viola-Jones algoritam je i danas prisutan u velikom broju mobilnih telefona i digitalnih kamera.

Pouzdanost i brzina su postignuti uvođenjem tri ključna doprinosa:

- **Integralna slika** omogućava brzo izračunavanje obeležja.
- **Adaptive Boosting (AdaBoost)** je algoritam za mašinsko učenje, odabiranjem obeležja povećava se brzina i pouzdanost detekcije.
- **Kaskadni klasifikator**, organizovanjem obeležja u kaskade omogućava brzo odbacivanje regiona sa malom verovatnoćom pojave traženog objekta, npr. pozadine slike.

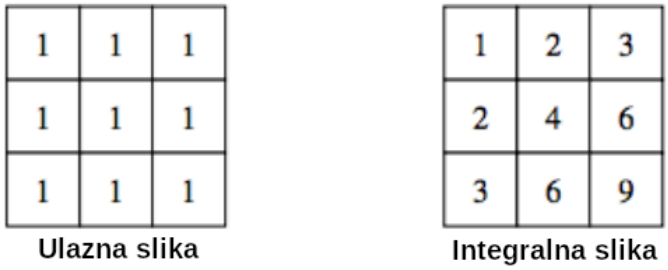
1.1 Integralna slika

Kao jedan od ključnih delova algoritma, integralna slika je uveliko zaslužna za brzinu detekcije. Integralna slika predstavlja transformaciju originalne slike, takvu da se zbir proizvoljnog broja piksela originalne slike može izračunati u konstantnom vremenu.

Vrednost piksela integralne slike na poziciji (x, y) je zbir svih piksela koji se nalaze u pravougaoniku gore i levo od pozicije (x, y).

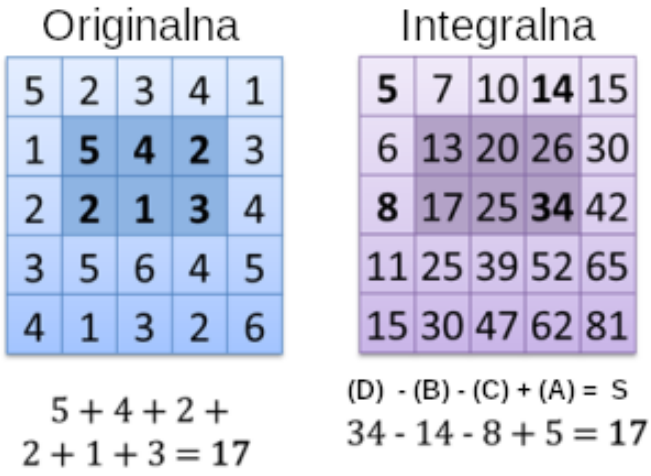
$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y') \tag{1.1}$$

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



Slika 1.1: Primer integralne slike

Kako je u Viola-Jones algoritmu potrebno izračunavanje površine nad pravougaonim obeležjem originalne slike. Integralna slika donosi veliko ubrzanje prilikom ove operacije.
U integralnoj slici za proizvoljno pravougaono obeležje moguće je izračunati površinu sa samo 2 oduizamanja i jednim sabiranjem.



Slika 1.2: Primer računanja površine pravougaonika [2]

Na slici (1.2) je prikazano računanje površine pravougaonika na originalnoj slici i iste površine 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.

Formula za računanje površine pravougaonika originalne slike pomoću integralne slike prikazan je u nastavku.

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

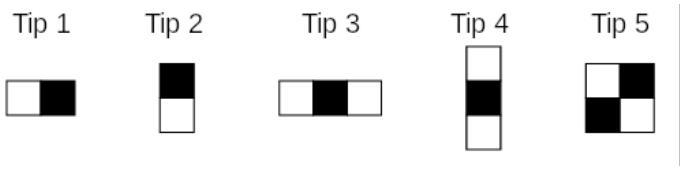
Treba napomenuti da je za transformaciju slike u integralnu potrebno značajno računsko vreme. Zbog toga je korišćenje integralne slike isplativo jedino ukoliko je potreban veliki broj operacija računanja površine pravougaonika na željenoj slici, što je slučaj u Viola-Jones algoritmu.

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

1.2 AdaBoost i HAAR obeležja

1.2.1 HAAR obeležja

Viola-Jones modeli za detekciju objekata se sastoje od mnogobrojnih HAAR obeležja. HAAR obeležja se sastoje od dva ili više pravougaonika. Ovi pravougaonici mogu imati svoje dimenzije i težinu. Na slici(1.3) prikazani su neki mogući tipovi HAAR obeležja, crni i beli pravougaonici imaju različite težine.



Slika 1.3: HAAR obeležja [4]

HAAR obeležja imaju dimenzije manje od dimenzije slike, tipične dimenzije ovih obeležja su oko 24x24 piksela.
Prilikom detekcije pomoću Viola-Jones algoritma pravougaoni prozor dimenzija istih kao HAAR obeležja prolazi preko slike.
Za trenutnu poziciju prozora potrebno je izračunati površinu piksela obuhvaćenu pravougaonicima HAAR obeležja zatim ih pomnožiti sa težinom i sabrati.
Akumulacijom zbirova svih HAAR obeležja u modelu moguće je proceniti da li se na trenutnom položaju prozora nalazi traženi objekat.

Kako se u modelu tipično nalaze preko hiljadu HAAR obeležja preprocesiranjem slike u integralnu sliku dobija se veliko ubrzanje prilikom računanja ovih površina.

Slika 1.4: Primer obeležja za detekciju lica [1]

Oblik obeležja zavisi 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 osvetljaja regiona čela i očiju. Ovo obeležje koristi činjenicu da je oblast čela svetlija od očiju.
Drugo obeležje poredi intenzitet regiona mosta nosa sa očima, ovde se koristi činjenica da je region mosta svetliji od očiju.

Za dimenziju prozora 24x24 kombinacije svih mogućih varijacija oblika i pozicija datih obeležja čini skup od oko 160.000 različitih obeležja. Neki od ovih obeležja neće biti korisni prilikom detekcije željenog objekta. Veliki broj obeležja će ciljati istu osobinu detektovanog objekta tako da neće dobrineti povećanoj preciznošću detekcije. Iz tih razloga moguće je drastično smanjiti i pronaći optimalan broj obeležja korišćenjem algoritma za mašinsko učenje poput AdaBoost.

1.2.2 AdaBoost

Kako je već rečeno može se dobiti oko 160.000 obeležja za prozor dimenzije 24x24. Potrebno je pronaći optimalan broj obeležja za detekciju željenog objekta. Ukoliko je broj obeležja previše mali detekcija neće biti pouzdana, a ukoliko je broj obeležja preveliki vreme detekcije je veće.
Kako je prikazano na slici(1.4) u primeru detektora lica, neka obeležja naglašavaju osobine objekta bolje od drugih.

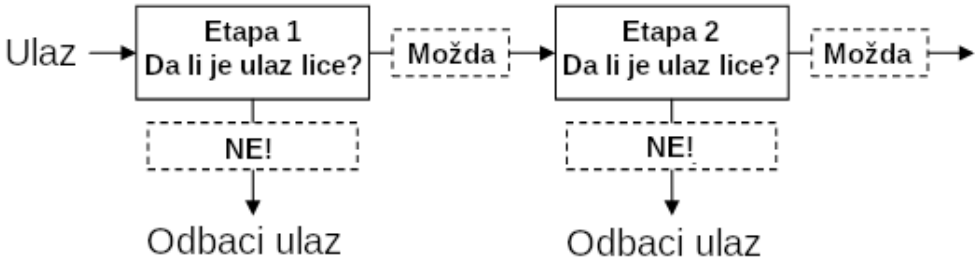
Kako bi se odabrao skup korisnih obeležja može se koristiti neki od algoritama mašinskog učenja. Viola i Jones predlažu modifikovani AdaBoost algoritam.

Ideja AdaBoost-a je da se kombinacijom više *weak learner*-a dobija pouzdana detekcija. *Weak learner* je klasifikator koji ima pouzdanost pogađanja malo bolju nego nasumično. Odnosno pouzdanost *weak learner*-a mora biti bar malo iznad 50%.
Kaskadiranjem ovako dobijenih *weak learner*-a može se dobiti *strong classifier*
Kao rezultat AdaBoost algoritma dobijamo skup optimalnih obeležja, u ovom radu se koristi model sa skupom od 2913 obeležja.

1.3 Kaskadni klasifikator

Osnovni cilj Viola-Jones algoritma je da se na osnovu svih HAAR 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 modela na svakoj poziciji bi bilo suvišno. Tako da je korišćenje jednog jakog klasifikatora računski neefikasno.

Ukoliko bi se za region koji sigurno ne sadrži lice ranije zaključilo da ga je moguće odbaciti, dobila bi se značajna ušteda u vremenu detekcije.
Organizovanjem obeležja u kaskade i obrazovanjem kaskadnog klasifikatora postiže se upravo to. Regioni koji sadrže pozadinu tipično se odbacuju posle jedne ili dve etape kaskade.

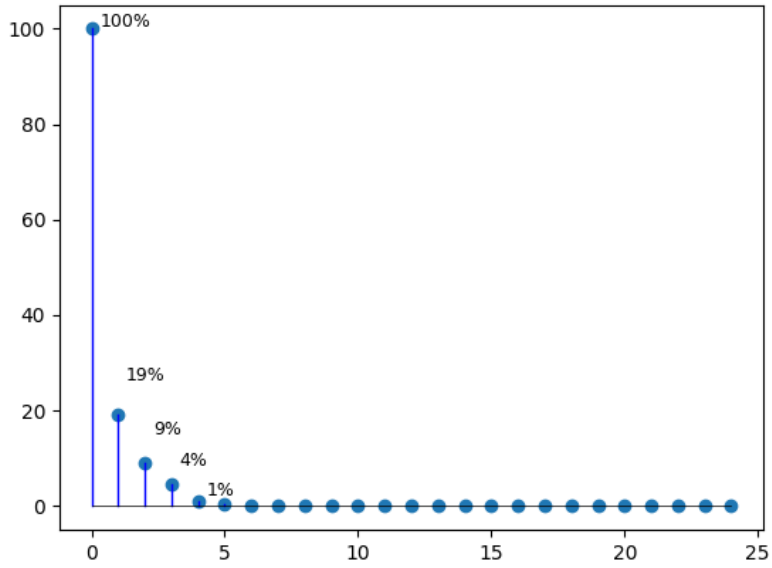


Slika 1.5: Kaskadni klasifikator [4]

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

Konačno ukoliko sve etape u klasifikatoru na analiziranom prozoru daju rezultat da se na njemu 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.
Takođe vreme detekcije je značajno skraćeno zbog brzog odbacivanja pozadine slike.

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



Slika 1.6: Procenat izvršavanja etapa na svim regionima slike

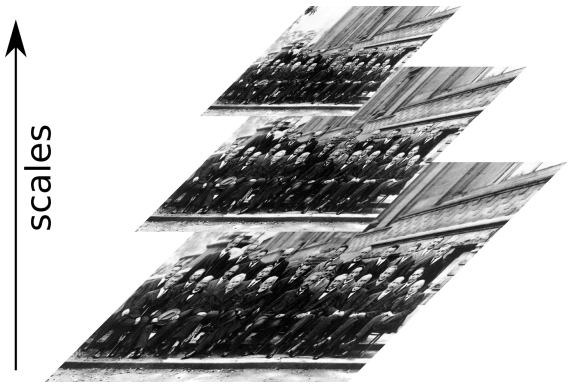
Na slici(1.6) je prikazana statistika izvršavanja etapa na *Caltech Dataset-u*[5].
Caltech Dataset sadrži 450 slika 27 različitih ljudi pod različitim osvetljenjima, izrazima lica i pozadinama.

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 Skaliranje slike

Objekti na slikama mogu biti bliži ili dalji kameri odnosno mogu biti različitih dimenzija, potrebno je obezbediti da se ona detektuju nezavisno od veličine.

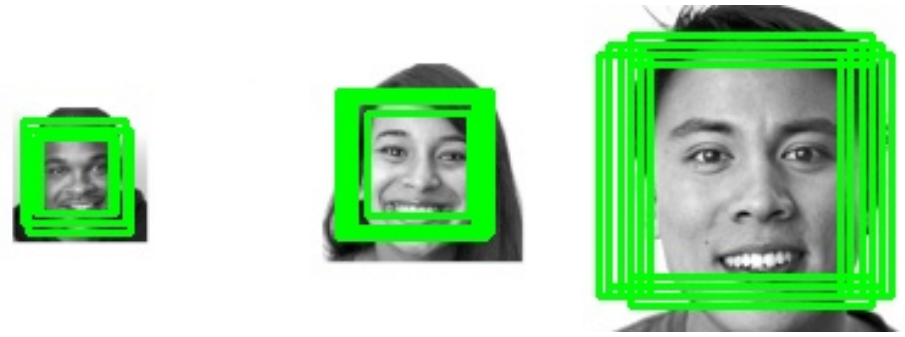
Pošto su obeležja istrenirana da detektuju samo lica koja su iste dimenzije kao i veličina obeležja potrebno je skalirati sliku ili obeležja kako bi mogli da se detektuju objekti veći od dimenzija obeležja.
Usled toga najmanja dimenzija objekta kojeg je moguće detektovati jednaka je dimenziji obeležja.



Slika 1.7: Piramida slike[6]

Ovaj problem je rešen skaliranjem slike. Na slici(1.7) prikazana je piramida skaliranih slika.
Detekcija se prvo vrši na slici originalnih dimenzija, nakon toga se dimenzije slike smanjuju sa nekim faktorom. Skaliranje se ponavlja sve dok je dimenzija skalirane slike veća od dimenzija obeležja.

Biranjem velikog faktora skaliranja dobija se veća brzina detekcije, ali se može izgubiti na preciznosti tako što će objekti određenih dimenzija biti ignorisani.
Potrebno je naći optimalan odnos brzine i preciznosti.



Slika 1.8: Različite veličine lica

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

1.5 Osetljivost na osvetljaj

Objekti se mogu naći pod raznim profilima osvetljenja što uzrokuje značajan problem ovom algoritmu.

Slika 1.9: Previše osvetljaja[5]

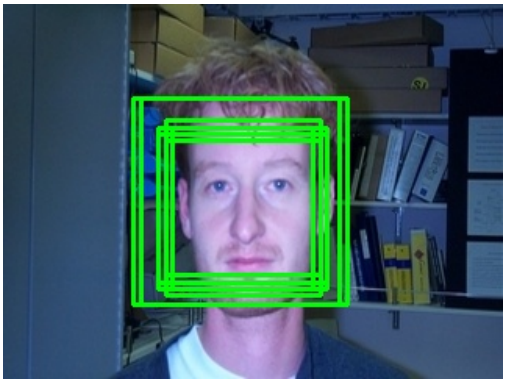
Slika 1.10: Premalo osvetljaja[5]

Na slikama(1.9, 1.10) su prikazana dva lica koja zbog nepovoljnog osvetljenja nisu detektovane od strane klasifikatora. Slika(1.9) nije detektovana zbog pristustva prejakog 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 Osetljivost na rotaciju objekta

Dodatna mana ovog algoritma je nemogućnost detektovanja objekata koji su rotirani. Odnosno moguće je detektovati objekte samo u onom položaju u kojem su prikazani prilikom treniranja modela.
Teoretski moguće je trenirati model za različite uglove rotacije objekta, ali to bi dodatno povećalo dimenzije modela i smanjilo pouzdanost detekcije.



Slika 1.11: Originalna[5]

Slika 1.12: Rotirana[5]

2 OpenCV modeli

OpenCV je biblioteka koja sadrži implementacije velikog broja algoritama za obradu slike. OpenCV sadrži alat za AdaBoost treniranje kaskadnog klasifikatora kao i implementaciju detektora objekata pomoću Viola-Jones algoritma.
Pored toga OpenCV sadrži istrenirane i testirane modele klasifikatora[1]. [7]

OpenCV istrenirane modele skladišti u .xml fajlove koji sadrže sledeće informacije o modelu:

- 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.
    Prve 2 vrednosti liste rects su x i y koordinate gornje leve tačke,
    treća i četvrta vrednost širina i visina pravougaonika,
    poslednja vrednost je 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
Hardverska implementacija

3 Sažetak

Cilj ovog rada je hardverska implementacija akceleratora Viola-Jones algoritma. Pored toga hardverska implementacija je urađena pomoću dve metodologije kako bi se utvrdile prednosti i mane ovih metodologija.

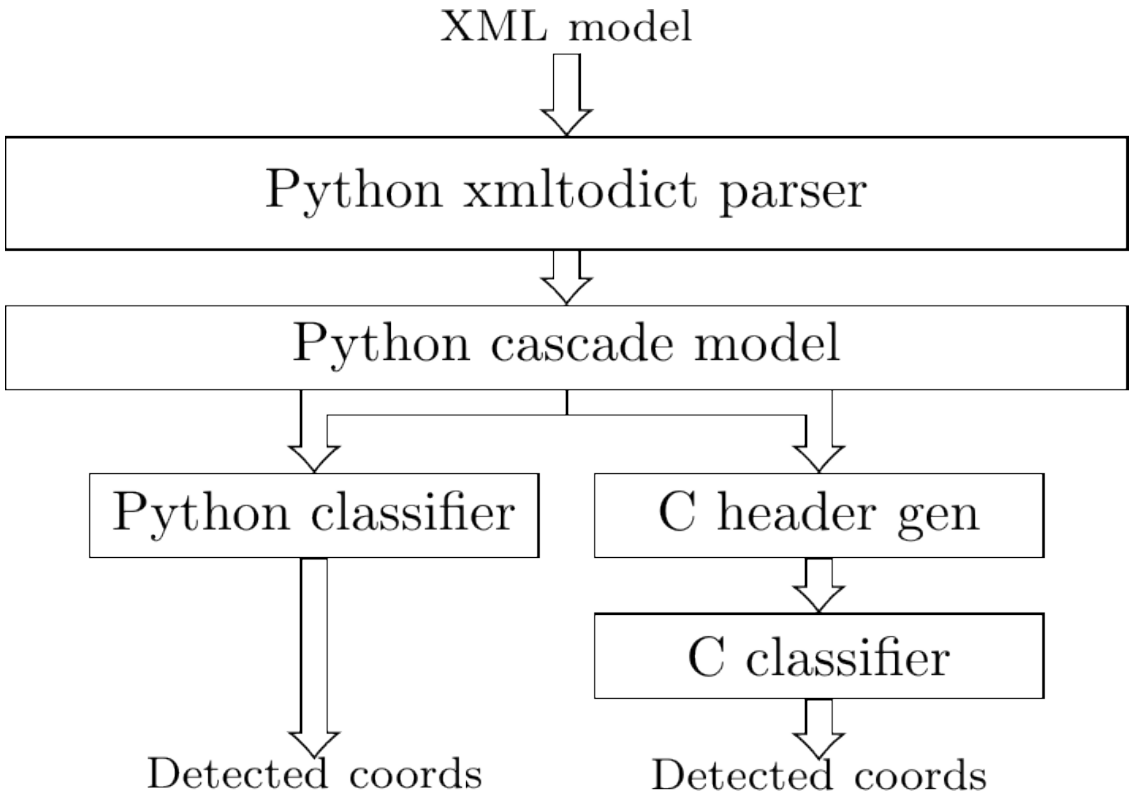
Kako bi se projektovao željeni hardverski akcelerator bilo je potrebno odraditi sledeće stvari:

- Pisanje implementacija u Python i C++ programskim jezicima kao specifikacija za izvršavanje, koja pritom pomaže prilikom particionisanja i projektovanja hardvera.
- Projektovanje arhitekture hardverskog akceleratora za Viola-Jones algoritam.
- Pisanje HDL modela za sintezu u SystemVerilog RTL metodologiji i PyGears[3] metodologiji.
- Poređenje dve metodologije i analiza prednosti i mana obe metodologije.
- Implementacija projektovanog IP jezgra na MYIR Z-Turn Board[4] ploči 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] `http://www.myirtech.com/list.asp?id=502`

4 Specifikacije za izršavanje

Prilikom projektovanja hardverske arhitekture određenog algoritma preporučljivo je prvo implementirati algoritam u softveru kako bi se algoritam bolje shvatio.
Postoje metodologije koje definišu potrebne korake prilikom projektovanja digitalnih sistema, jedna takva metodologija je Electronic System Level Design and Verification (ESL). U ovom radu nije korišćena ova metodologija već je softverska specifikacija napisana u C++ i Python jeziku.
Iz ovih specifikacija dobijeno je bolje razumevanje algoritma i naznake o mogućnosti paralelizna određenih delova algoritma i particionisanja komponenti sistema.



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

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

Na ulazu se nalazi eXtensible Markup Language (XML) model dobijen treniranjem pomoću OpenCV biblioteke opisan u sekciji 2.
Parsiranje XML modela se rešava u Python-u. Zbog velikog broja Python paketa dostupnih sa gotovim rešenjima za većinu softverskih problema, problem parsiranja XML fajla se može rešiti korišćenjem paketa xmltodict[5].
Xmltodict parsira XML fajl i skladišti ga u Python dictionary.

[5] `https://pypi.org/project/xmltodict/`

Implementirane su klase *CascadeClass*, *StageClass*, *FeatureClass* i *RectClass* [6] koje predstavljaju abstraktni Python model modela kaskadnog klasifikatora.

Napisana je i Python implementacija Viola-Jones algoritma koja koristi Python model klasifikatora i koristi se kao specifikacija za izvršavanje.

Python model klasifikatora može da generiše *C++* reprezentaciju modela klasifikatora i sačuva ih u Header fajlove.
Ovim je izbegnuto parsiranje XML fajlova C++ jezikom, pošto je ovaj zadatak mnogo jednostavnije odraditi u Python-u.
C++ implementacija Viola-Jones algoritma koristi ovako generisan model klasifikatora.

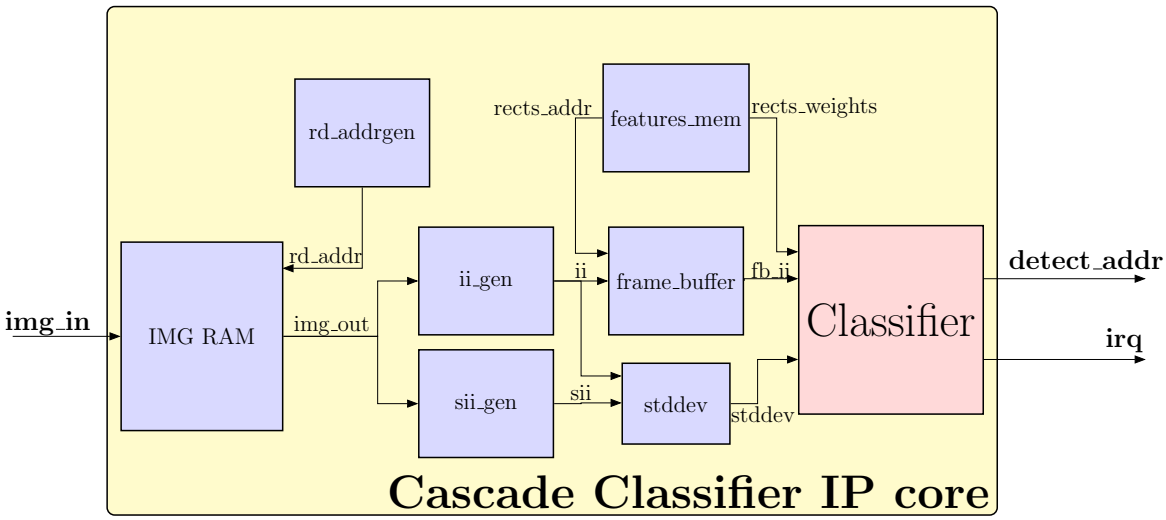
[6] `cascade_classifier/python_model/cascade.py`

5 Arhitektura hardvera

5.1 Uvod

U ovom poglavlju biće opisana projektovana arhitektura hardvera.
Predložena hardverska arhitektura će biti opisani uz što manje detalja implementacije. PyGears i SystemVerilog implementacije će pratiti predloženu arhitekturu, ali će se razlikovati u detaljima.

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



Slika 5.1: Arhitektura hardvera kaskadnog klasifikatora

U nastavku će biti opisana arhitektura. Prvo će biti opisani spoljni interfejsi, zatim će detaljnije biti opisane interne komponente.

5.2 Interfejsi IP jezgra

IP jezgro se povezuje ostatkom sistema pomoću 3 spoljna interfejsa:

- Ulazni interfejs **img_in** je 8-bitni podatak i predstavlja vrednosti piksela slike u *grayscale* formatu (nijanse sive).
- Izlazni interfejs **detect_addr** ukoliko jezgro detektuje objekat na slici, na ovom interfejsu će se pojaviti X i Y koordinate detektovanog objekta.
- Izlazni interfejs ili signal **irq** signalizira završetak obrade slike i daje do znanja ostatku sistema da je jezgro spremno za obradu nove slike. Namenjen da se koristi kao prekidni signal za procesor.

5.3 Modul IMG RAM

U toku rada IP jezgra postoji potreba za ponovnim čitanjem istih piksela slike. Čitanje iz eksterne memorije može biti sporo i energetski neefikasno, kako bi se smanjila potreba za višestrukim čitanjem istog podatka iz spoljne memorije, odlučeno je da se cela slika se čuva u manju memoriju unutar IP jezgra.

Modul **IMG RAM** je zadužen za skladištenje slike koja se obrađuje. Sastoji se od Random Access Memory (RAM) memorije i brojača za generisanje adrese za upis.
Korišćena RAM memorija treba da ima jedan port za upis i jedan za čitanje.

Prilikom upisivanja slike u memoriju, posle svakog primljenog podatka na ulaznom interfejsu brojač adrese upisa se poveća za 1.

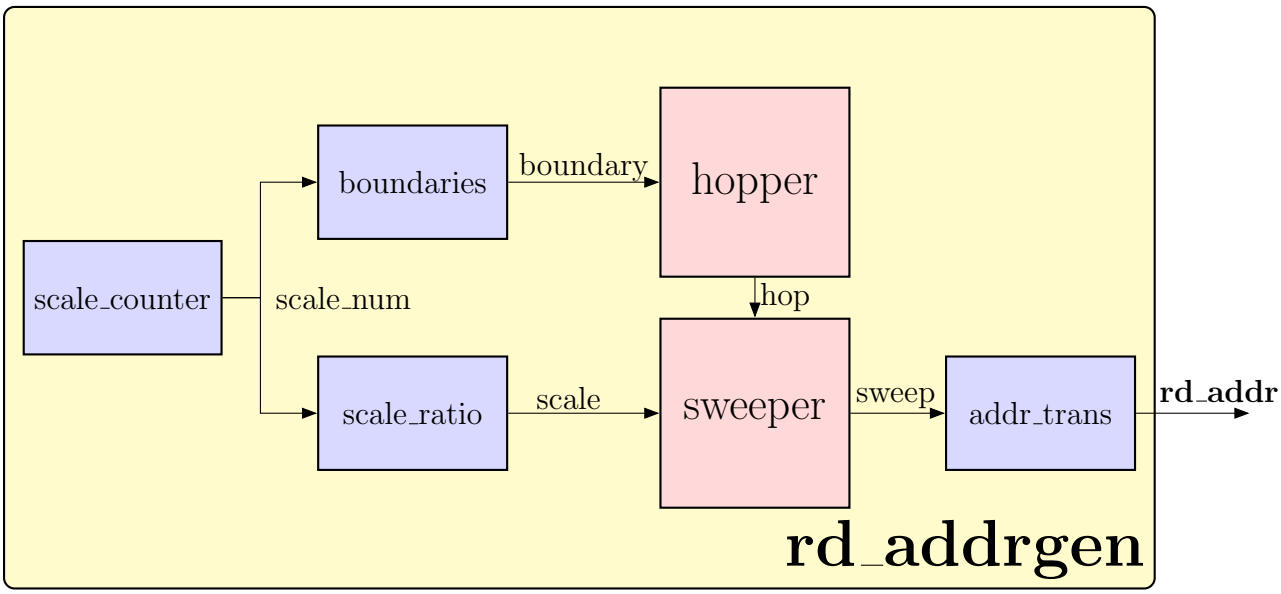
Skladištena slika je u 8-bitnom formatu i predstavlja grayscale vrednost piksela originalne slike.
Veličina memorije je $width * height * 8$ bit. Za dimenziju slike 240x320 veličina memorije je 614400 bita.

IMG RAM se povezuje sa ostatkom sistema preko 3 interfejsa:

- Ulazni interfejs **img_in** ekvivalentan je sa istoimenim interfejsom na višem nivou hijerarhije.
- Ulazni interfejs **rd_addr** prihvata linearizovanu adresu za čitanje piksela slike iz ove memorije. Ova adrese je generisana od strane **rd_addrgen** modula.
- Izlazni interfejs **img_out** šalje pročitane vrednosti piksela iz RAM memorije ostatku sistema.

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

U sklopu ovog modula rešeno je i skaliranje slike, opisano u sekciji 1.4.

5.4.1 Scale_counter

Scale_counter je brojač koji predstavlja trenutni nivo piramide skaliranja prikazane na slici(1.7). Nakom završetka obrade slike na trenutnom nivou piramide ovaj brojač se uveća za 1.

5.4.2 Boundaries

Ulazni interfejs **Boundaries** modula je povezan sa izlazom **scale_counter** modula, odnosno dobija trenutni stepen skaliranja slike kao ulaz. Na osnovu stepena skaliranje ovaj modul na izlazu daje granične vrednosti **X** i **Y** koordinata za **hopper** modul.
Granice osiguravaju ispravno generisanje adrese, odnosno obezbeđuju da adresa za čitanje iz **IMG RAM** modula nikad ne iskoči iz opsega memorije.
Ove vrednosti su fiksne i moguće ih je izračunati pre sinteze hardvera, pa su zato softverski izračunate i prosleđene kao parametri.
Ovaj modul se sastoji od dva multipleksera i liste unapred poznatih konstanti.

5.4.3 Scale_ratio

Scale_ratio je sličan **boundaries** modulu i prihvata isti ulazni podata.
Razlika u odnosu na **boundaries** modul je u vrednosti parametara. Parametri u ovom modulu se prosleđuju **sweeper** modulu i omogućavaju računanje skalirane adrese pomoću aritmetike sa fiksnom tačkom.

5.4.4 Hopper i Sweeper

Hopper se može zamisliti kao dvostruka ugnježdena for petlja gde iteratori petlje predstavljaju koordinate **X** i **Y**.
Prvo se iterira po **X** kordinati pa zatim 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 gornje granice **hopper**-a predstavljaju konstante **boundary_x[scale_num]** i **boundary_y[scale_num]**, kao što je opisano u sekcijama (5.4.2, 5.4.1).

Iteratori petlje se prosleđuju na izlazni interfejs **hopper**-a i služe kao početne tačke za brojače **sweeper**-a koji će biti opisan u nastavku.

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

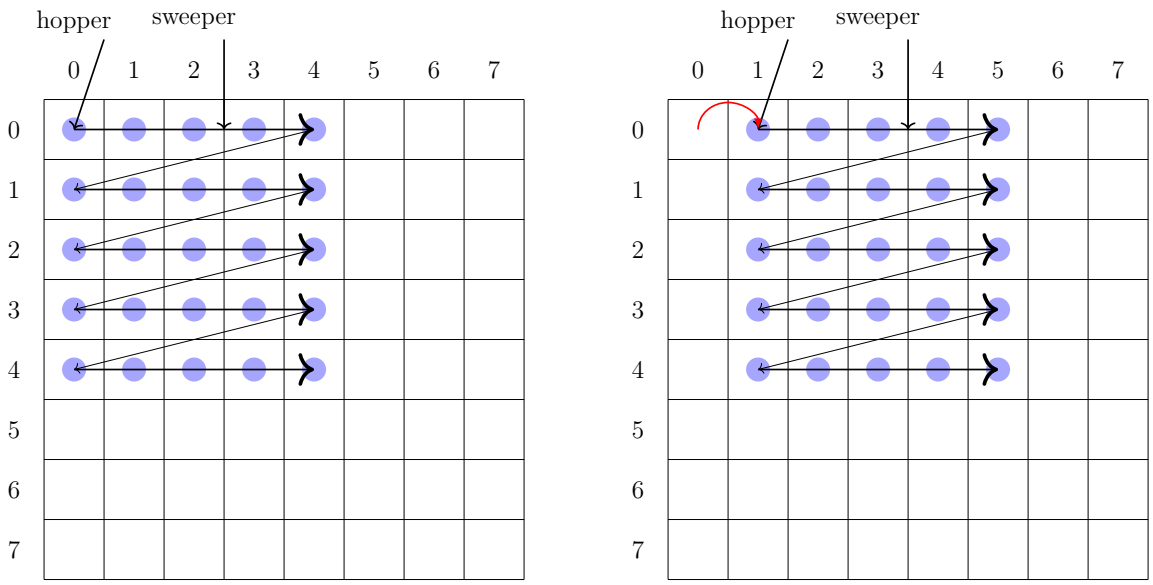
```
1   int x, y;
2   // hopper
3   for(int hop_y = 0; hop_y < boundary_y[scale_num]; hop_y++){
4     for(int hop_x = 0; x < boundary_x[scale_num]; hop_x++){
5       // sweeper
6       for(y = hop_y; y < hop_y+feature_height; y++){
7         for(x = hop_x; x < hop_x+feature_width; x++){
8           //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

Primer 5.2: Primer **hopper**-a i **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 kao donje granicu petlji uzeti trenutne vrednosti **hopper** koordinata, zatim će se iterirati **feature_width** i **feature_height** puta.

Rad **hopper**-a i **sweeper**-a je nalakše opisati slikama, što je i učinjeno na slikama(5.3, 5.4). Na slikama (5.3, 5.4) prelazak **sweeper**-a po slici je prikazan horizontalnim i dijagonalnim strelicama. Plavim kružićima predstavljeni su pikseli koje sweeper prebriše za jedan položaj hopper-a.

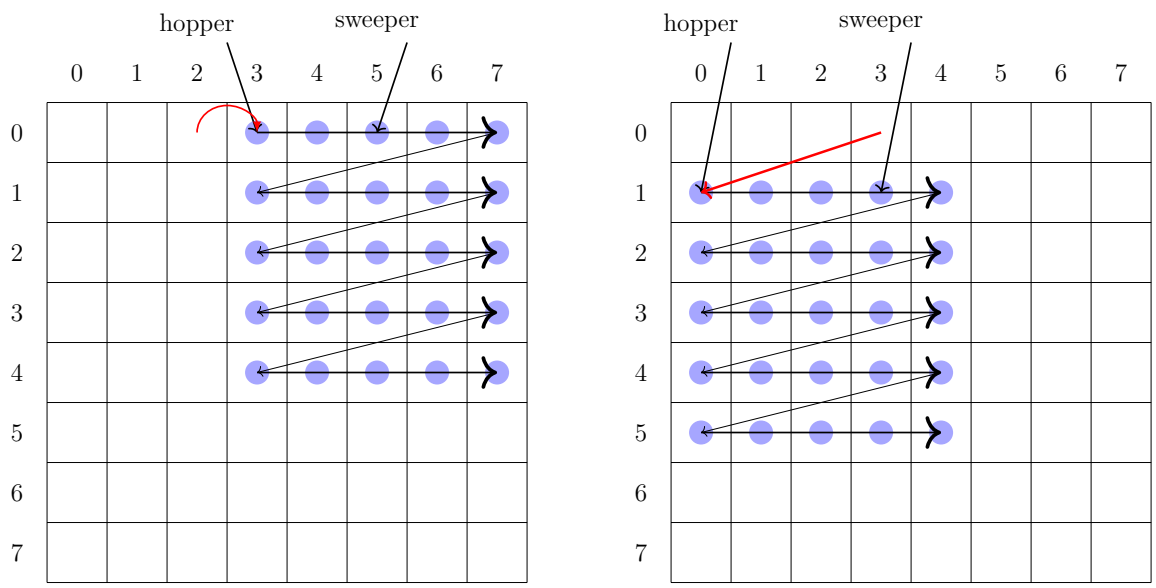
Gornji levi kružić predstavlja koordinatu trenutnog položaja hopper-a. Crevnom strelicom je obeleženo iteriranje hopper-a kroz sliku.



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

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 nulu.

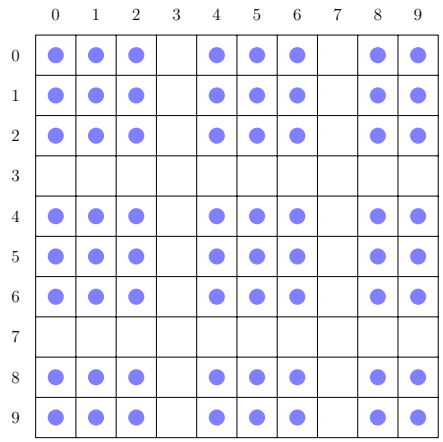
Nakon što hopper dostigne obe granice boundary_x i boundary_y završen je trenutni stepen skaliranja slike i potrebno je uvećati scale_num za jedan.



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

5.4.5 Skaliranje adrese

Unutar sweeper-a je implementirano i skaliranje adrese u svrhu skaliranja slike objašnjeno u sekciji 1.4.
Kako bi se adresa skalirala 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, množenje sa fiksnom tačkom je efikasnije.
To je odrađeno tako što je celobrojni faktor za množenje unapred softverski izračunat i smešten u scale_ratio modul opisan ranije, isto tako je i broj pomeranja u desno dobijene binarne vrednosti unapred poznat. Na ovaj način je uštređeno dosta hardverskih resursa.



Slika 5.5: Posledica skaliranja adrese.

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 nezavisnost detekcije od veličine objekta, opisano u sekciji 1.4.

5.4.6 Modul addr_trans

Zamišljanje slike kao dvodimenzionalne matrice je intuitivno. Međutim podaci u RAM memoriji su složeni u jednodimenzionalnom nizu i tako se trebaju adresirati.
Kako bi se ova neusklađenost rešila potrebrebno je linearizovati adresu dobijenu iz prethodnih modula.
Odnosno adresu u formatu (y, x) treba pretvoriti u linearnu pomoću sledeće 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.
Operaciju skaliranja obavlja modul **addr_trans**.

5.5 Modul ii_gen i sii_gen

5.5.1 Odabir algoritma

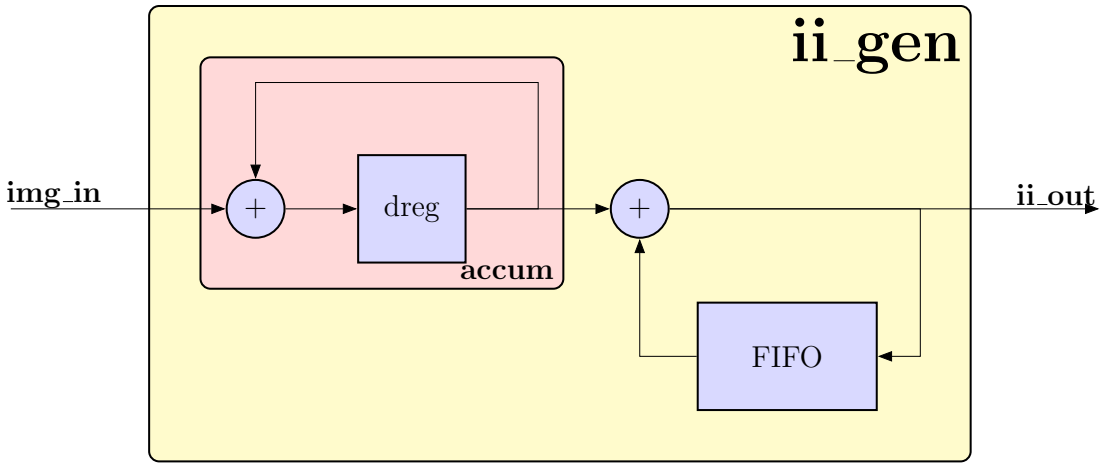
Jedan od kritičnih delova Viola-Jones algoritma je generisanje integralne slike opisane u poglavlju 1.1.
Ispostavlja da i u hardverskoj implementaciji generisanje integralne slike ima veliki uticaj na performanse i potrebne hardverske resurse sistema.

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 paralelni algoritam neće biti razmatran.

5.5.2 Sekvencijalna implementacija generatora integralne slike

Zbog jednostavnosti i potrebom za malim brojem resursa za ovu arhitekturu odabran je sekvencijalni algoritam generisanja integralne slike koji odgovara jednačini(5.1) iz sekcije 1.1.

Prednosti ovog algoritma u odnosu na paralelni su manji memorijski zahtevi na ulaznom i izlaznom interfejsu, potrebno je manje funkcionalnih jedinica i unutrašnje memorije. Dok je mana je manja brzina računanja.
Ovaj algoritam izračunava po jedan piksel svaki takt.



Slika 5.6: Blok dijagram ii_gen modula

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

Ulazni interfejs ovom modula se povezuje izlaznim interfejsom IMG RAM memorije.

Pikseli u istom 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 prozora. Ovo je moguće dodavanjem dodatnih polja uz vrednost ulaznog piksela, koji predstavljaju kraj transakcije za red i kolonu. O ovom konceptu će biti više reči u sekciji posvećenoj PyGears tipovima.

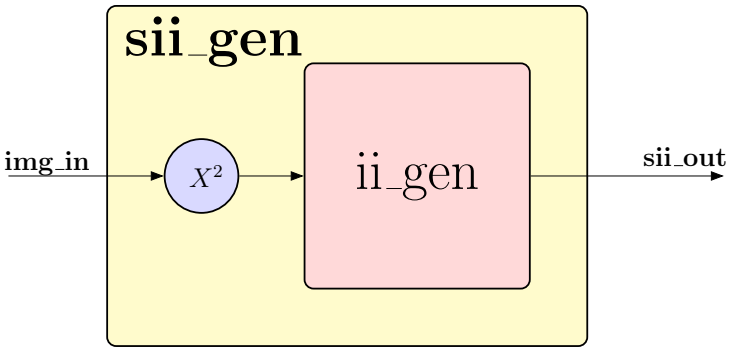
Nakon toga akumulirana vrednost se sabira sa sadržajem FIFO bafera koji sadrži vrednost piksela integralne slike iz predhodnog reda. Zatim se vrednost prosleđuje na izlaz i ponovo upisuje u FIFO bafer. Iz FIFO bafer-a će na ovaj način izlaziti vrednosti piksela iz prethodnog reda, a bafer će se puniti vrednostima trenutnog reda.

U ovom slučaju standardan 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

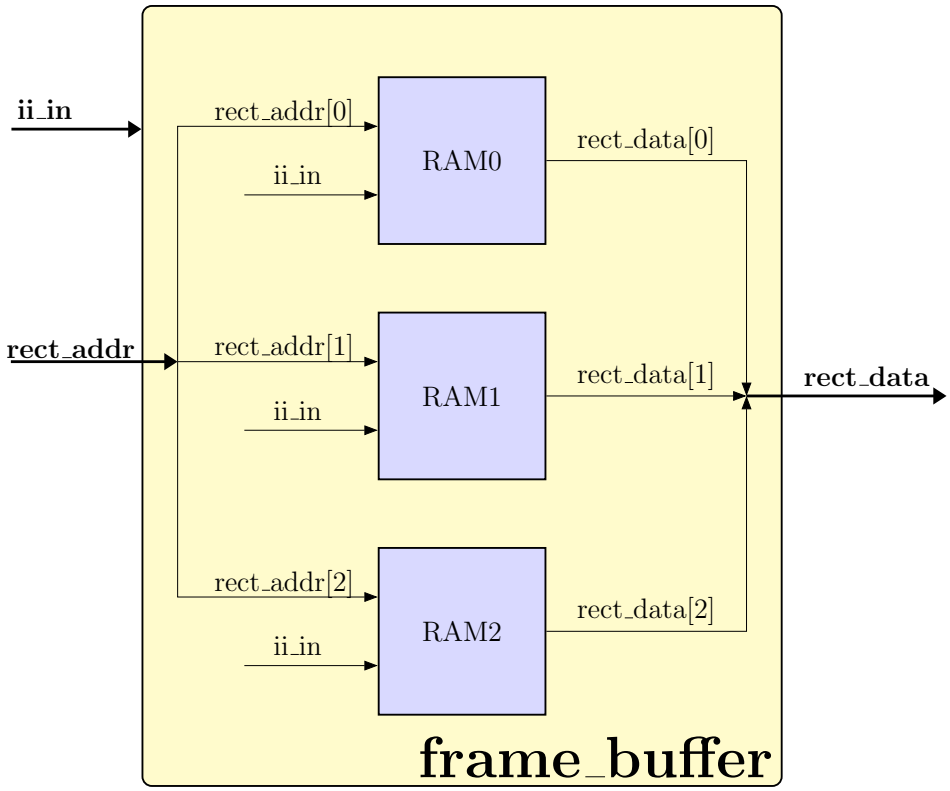
Generator kvadratne integralne slike je potreban za računanje standardne devijacije prozora što je opisano u sekciji 1.5.
Generisanje kvadratne integralne slike je jednostavno uz korišćenje generatora integralne slike. Potrebno je kvadrirati ulazne piksele i dovesti ih na generator integralne slike kao 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 i da vrednost bude dostupna u roku od jednog takta. 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 memorija. Brojač adrese upisa se inkremetuje za jedan nakon svakog primljenog podatka kao i kod IMG RAM modula. Brojač nije prikazan na blok dijagramu iznad.

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 su frame_width i frame_height širina i visina obeležja modela, w_ii je širina magistrale integralne slike, w_img je 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 dvo-portne memorije. Ovo će kao rezultat zauzeti tri puta više RAM memorije na čipu, od koje će svaka imati isti sadržaj.

Alat za sintezu uglavnom može da odradi transformaciju i da od jedne šesto-portne memorije instancionira tri dvo-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 objekata 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);
```

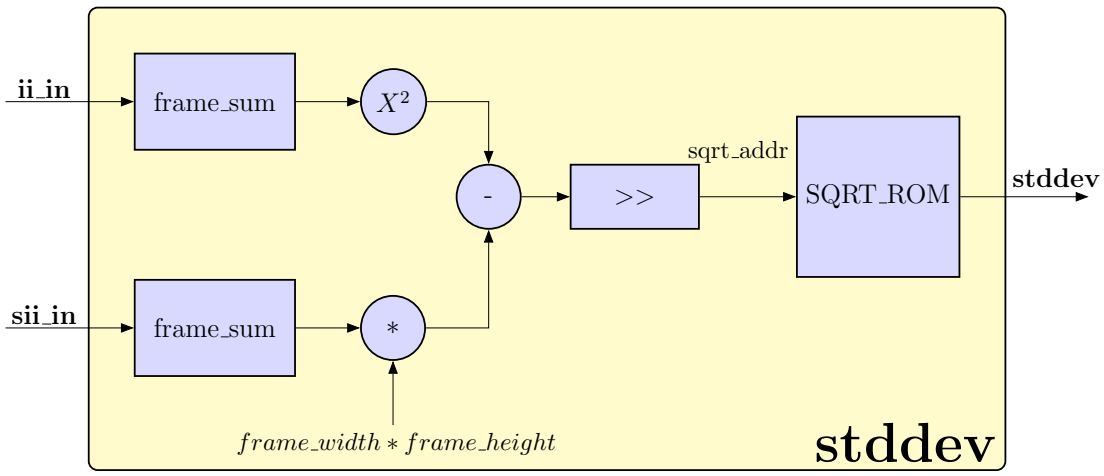
```
13      return stddev;
14    }
```

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

Za računanje standardne devijacije potrebne su pikseli 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 alati 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.



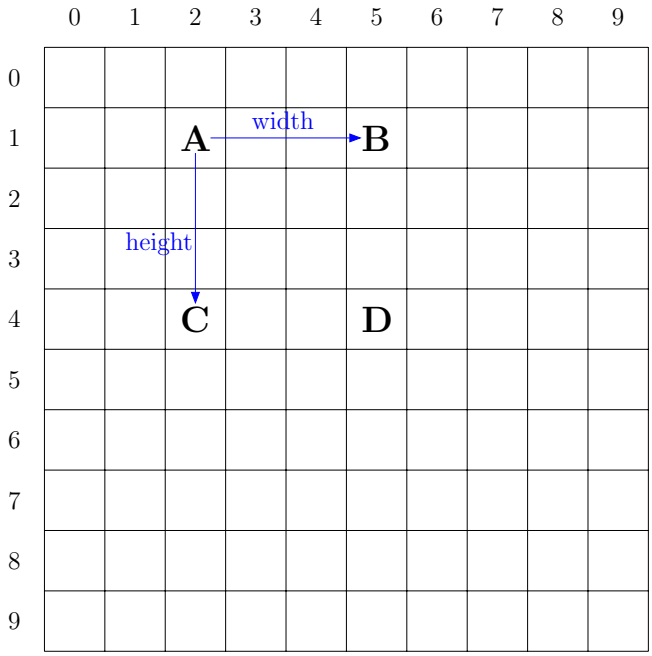
Slika 5.9: Blok dijagram stddev modula.

Na slici(7.1) 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 Read Only Memory (ROM) memorija, pod nazivom SQRT_ROM i sadrži 256 memorijskih 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 čestog i ponovljenog pristupa ovim obeležjima potrebno ih je skladištiti u lokalnoj memoriji.

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 za računanje ostalih koordinata. 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(5.4.6) 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)
0	0x1A989
1	0x1A987
2	0x39249
3	0x72926
. . .	. . .
feature num - 1	0x088d6

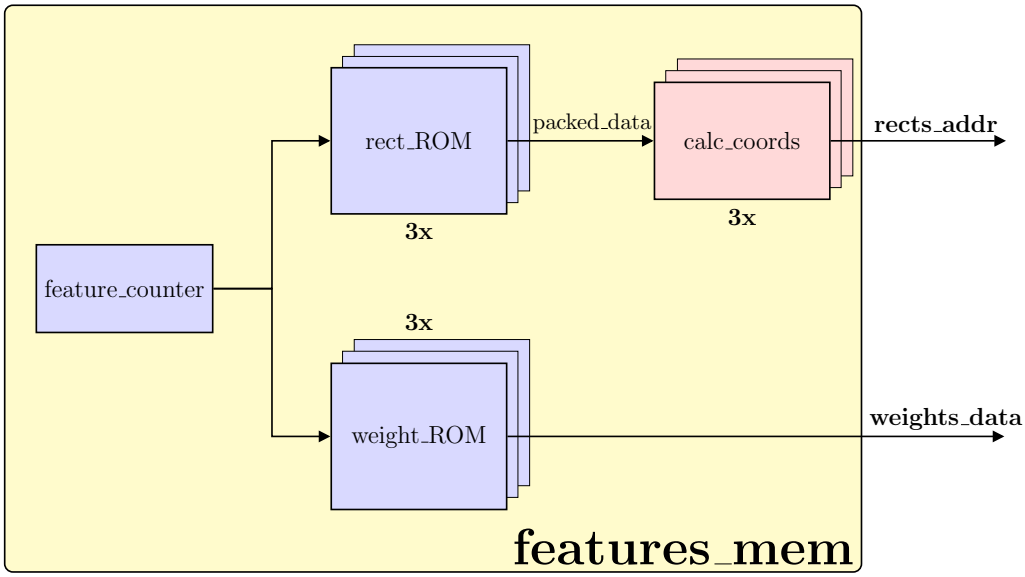
Tabela 5.1: Struktura zapakovane memorije rect_ROM0 za model 2.1

N	Height (5bit)	Width (5bit)	A linear coord (10bit)
0	9	12	106
1	7	12	106
2	9	18	228
3	19	9	458
. . .	. . .	. . .	. . .
feature num - 1	22	6	34

Tabela 5.2: Struktura raspakovane memorije rect_ROM0 za model 2.1

Na tabeli(5.1) je prikazana strukura zapakovane memorije rect_ROM
Dok je tabeli(5.2) prikazana polja raspakovane memorije rect_ROM
U prvoj koloni **N** su adrese lokacija, kojih ima features_num (2913 u modelu 2.1).
Na prvoj tabeli drugoj koloni **Packed value(20 bit)** je zapakovana vrednost memorijske lokacije na adresi u heksadecimalnom zapisu.
Na drugoj tabeli na drugoj i trećoj koloni se nalaze **height** i **width** raspakovane vrednosti visine i širine pravougaonika.
Na drugoj tabeli i 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).

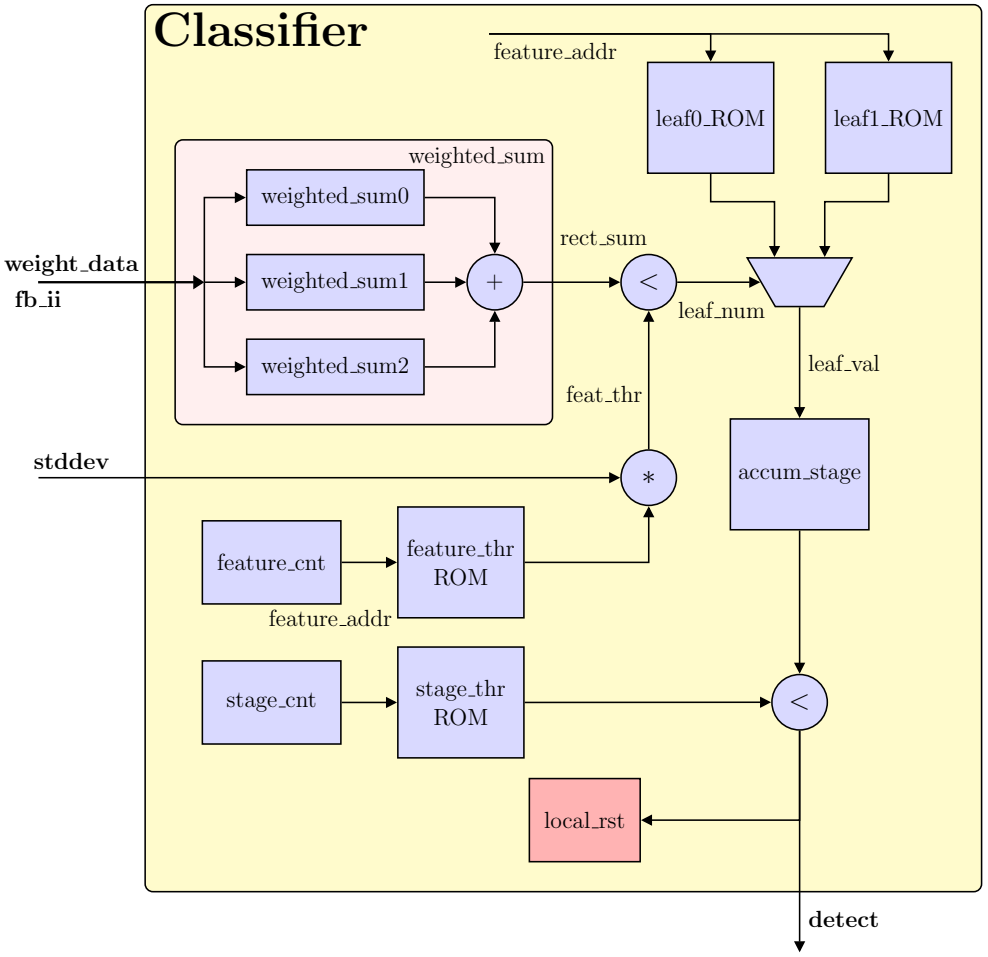
feature_num	2913
rect_num	3
w_weight	3 bits
w_rect	20 bits
TOTAL	524,340 bits

Tabela 5.3: Veličina memorije features_mem za model 2.1

5.9 Modul classifier

Classifier modul obavlja samu klasifikaciju prozora. Pored ii_gen(5.5) modula najviše utiče na performanse sistema.

Na slici ispod je prikazan blok dijagram klasifikatora.



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.

```
int feature_thresholds[FEATURE_NUM];
int leaf_val0[FEATURE_NUM];
int leaf_val1[FEATURE_NUM];

int leaf_val(long stddev,
             long sum,
             int feature_num){

  if(sum <= feature_thresholds[feature_num] * stddev)
      return leaf_val0[feature_num];
  else
      return leaf_val1[feature_num];
}
```

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

Primer(5.6) 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.

Dalje je potrebno akumulirati vrednosti unutar etape i uporediti sa pragom etape. Ukoliko je akumulirana vrednost manja od praga etape može se zaključiti da se na posmatranom prozoru ne nalazi objekat. Ukoliko je akumulirana vrednost veća od praga etape na posmatranom prozoru se može naći objekat i potrebno je izvršiti sledeću etapu.

```
int stage_res(int leaf_val,
              int feature_start,
              int feature_end){
int64_t sum = 0;

for(int feature_num = feature_start;
    feature_num < feature_end;
    feature_num++){
  sum += leaf_val;
}
```

```
11
12    if(sum < stageThresholds[stage_num]) return -1;
13    else return 1;
14  }
```

Primer 5.6: Leaf_val u **C**-u

U hardverskoj implementaciji ukoliko je akumulirana vrednost etape manja od praga etape, aktiviraće se local_rst modul i resetovati **classifier** modul, ovo će signalizirati ostatku sistema da se na posmatranom prozoru ne nalazi objekat.

5.10 Interfejsi

Komunikacija svih unutrašnjih modula i komunikacija IP jezgra sa okolinom realizovana je pomoću **Data Transfer Interface (DTI)**[9] interfejsa.
Korišćenjenjem ovog interfejsa dobijena je robusnija komunikacija između modula zahvaljujući *handshaking*-u.

Dodatna prednost ovog interfejsa je kompatibilnost sa **AXI-Stream**[10], **Avalon-ST**[11] i sličnim streaming interfejsima. Zahvaljujući tome moguće je povezati ovaj interfejs sa AXI-Stream i Avalon-ST bez dodatnog adaptera.

Interfejs će biti detaljnije objašnjen u PyGears(6) sekciji.

6 PyGears metodologija

6.1 Uvod

PyGears[12] pozajmljuje koncepte iz funkcionalnog programiranja i primenjuje ih na dizajnu digitalnog hardvera
Sastoji se od Gears metodologije i Python Framework koji implementira Gears metodologiju. Gears metodologija omogućava lako povezivanje i kompozabilnost sistema pomoću manjih funkcionalnih jedinica pod nazivom gear-ovi.
Gear moduli su međusobno povezani DTI interfejsom koji implementira jednostavan handshake protokol. Korišćenjem standardnog generičkog interfejsa omogućeno je lako povezivanje ovih komponenti.

Opisom hardvera u Python-u koji je dinamičan jezik i veoma visokog nivoa mogu se napisati gear-ovi koji su veoma apstraktni i generički, time je omogućeno lako ponovno korišćenje modela.

PyGears takođe omogućava i verifikaciju u Python-u. Implementiran je interfejs ka popularnim komercijalnim simulatorima Questa, NCSim i open source Verilator, tako da je moguća kosimulacija između Python okruženja i Hardware Description Language (HDL) modela.

PyGears konačno obavlja konverziju Python opisa u System Verilog (SV) opis koji je podržan od strane alata za sintezi i simulaciju.

6.2 Poređenje sa RTL metodologijom

PyGears[9] predstavlja alternativu Register Transfer Methodology (RTL)[13] metodologiji za opis hardvera. RTL metodologija pruža standardan način translacije sekvencijalnog algoritma u hadverski opis. Struktura dobijena pomoću RTL metodologije uglavnom se sastoji od **Finite State Machine with Datapath (FSMD)**.
Datapath sadrži funkcionalne i memorijske jedinice i mreže za rutiranje podataka[14]
Controlpath diriguje podacima u Datapath delu i sačinjena je od mašine stanja.

Kako dizajn implementiran pomoću RTL metodologije postaje kompleksniji i mašina stanja postaje kompleksnija i sadrži više stanja. Pipeline-ovanje, debagovanje ovakvog dizajna može biti veoma teško.
Prednost PyGears metodologije u odnosu na RTL metodologiju najvidljivija je u sistemima koji su Dataflow orijentisani.

6.3 Jezici za opis hardvera

Jezici sa opis hardvera koji se danas daleko najviše koriste su Verilog i VHDL, nastali su 1984. i 1983. godine i daleko su siromašniji po mogućnostima od današnjih jezika višeg nivoa kao što su Python, C++.

Najveća prednost ovih jezika je podrška od strane alata za sintezu i simulaciju, kako se ovi jezici koriste za opis i verifikaciju hardvera preko 30 godina, alati su zreli i dobro istestirani.

Sve kompanije koje proizvode FPGA čipove trenutno ne objavljuju arhitekturu svojih FPGA čipova, pa alati za sintezu i implementaciju zavise od tih kompanija[7] . Zbog toga direktna podrška alata za sintezu i implementaciju za neki novi HDL jezik je nemoguća.

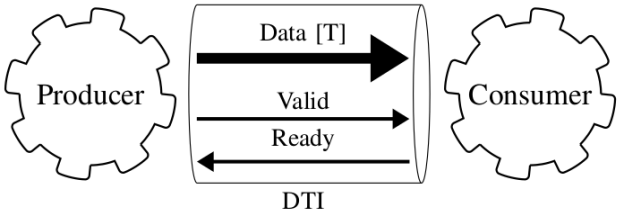
Iz tog razloga većina novih HDL jezika baziranih na Python[16, 9], Scala [17, 18], Haskell[19] generišu konačan HDL kod u Verilog-u, SystemVerilog-u ili VHDL-u pogodnom za simulatore i alate za sintezu.

Dok se ovi jezici uglavnom baziraju na RTL metodologiji i uvode jezik višeg nivoa kao zamenu za standardan HDL. PyGears dodatno izdvaja i uvođenje nove metodologije zvane Gears.

6.4 Gears metodologija

Gears metodologija obezbeđuje bolju kompozabilnost i ponovno korišćenje napisanih hardverskih modela.
Kako bi se ovo postiglo uveden je standardan generički handshaking interfejs za komunikaciju između modula.

6.4.1 DTI interfejs



Slika 6.1: DTI interfejs[9]

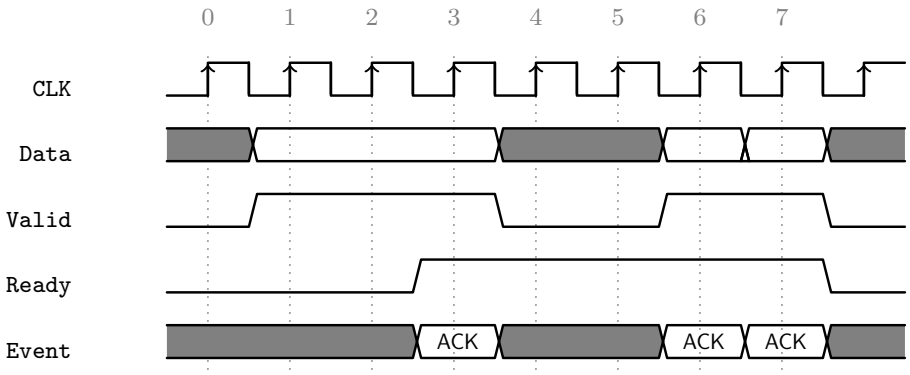
Jednostavan **DTI** interfejs se koristi za komunikaciju između gear-ova. Modul koji šalje podatke se naziva Producer, a modul koji prima podatke se naziva Consumer.
DTI se sastoji od sledećih signala:

- **Data** linije za podatke, proizvolje širine i proizvoljnog tipa.
- **Valid** jednobitan signal kojim upravlja Producer i signalizira da je podataka na Data liniji validan.

[7]Postoji inicijativa da se dokumentuju FPGA čipovi svih velikih proizvođača u projektu zvanom SymbiFlow[15], kao i alati za sintezu i implementaciju.

- **Ready** jednobitan signal kojim upravlja Consumer i signalizira da je konzumirao podatak sa Data linije i spreman za novi podatak.

Pomoću Valid i Ready signala realizovan je handshaking protokol.



Slika 6.2: DTI primer transakcije

Na slici(6.2) prikazan je talasni oblik protokola:

1. Producer je postavio podatak na Data liniju i podigao Valid signal na jedinicu. Što označava validan podatak na magistrali.

2. Istog trenutka nakon postavljanja Valid signala na jedinicu Consumer može da koristi podatak na magistrali.

3. Consumer može koristiti podatak potreban broj taktova.

4. Kada Consumer završi sa korišćenjem podatka, postavlja Ready signal na jedinicu čime označava ACK(acknowledgment) odnosno signalizira da je završio sa podatkom na magistrali i da je spreman za sledeći podatak.
   Nakon handshake-a Producer može postaviti novi podatak na magistralu.

5. Producer ne sme menjati podatak na magistrali sve do pojave ACK od strane Consumera.

6. Producer može držati Valid signal na logičkoj nuli i tada se neće obavljati transakcije na magistrali.

7. Ne sme postojati kombinaciona putanja od Ready do Valid signala na strani Producera. Odnosno Producer ne sme odlučivati o postavljanju podatka na magistrali na osnovu stanja Consumer-a.

8. Može postojati kombinaciona putanja od Valid do Ready signala na strani Consumer-a.

6.5 Tipovi podataka

Data linija u DTI interfejsu pored toga što može biti proizvoljne širine može predstavljati i različite kompleksne tipove podataka. Ovim se omogućava bolja kompozicija modula, lakša manipulacija podacima, pregledniji izvorni kod.

6.5.1 Uint

Uint[W] predstavlja neoznačenu celobrojnu vrednost proizvoljne širine W.

6.5.2 Int

Int[W] predstavlja označenu celobrojnu vrednost proizvoljne širine W.

6.5.3 Tuple

Tuple[T1, T2, ..., TN] predstavlja tip sličan strukturi gde polja Tn mogu biti bilo kog tipa.
Koordinate piksela slike mogu se prigodno predstaviti kao Tuple tip.
Na primer Tuple[Uint[W_Y], Uint[W_X]] ovaj Tuple sadrži dva Uint polja koji predstavljaju X i Y koordinate.

6.5.4 Array

Array[T, N] predstavlja niz od N podataka tipa T.

6.5.5 Float

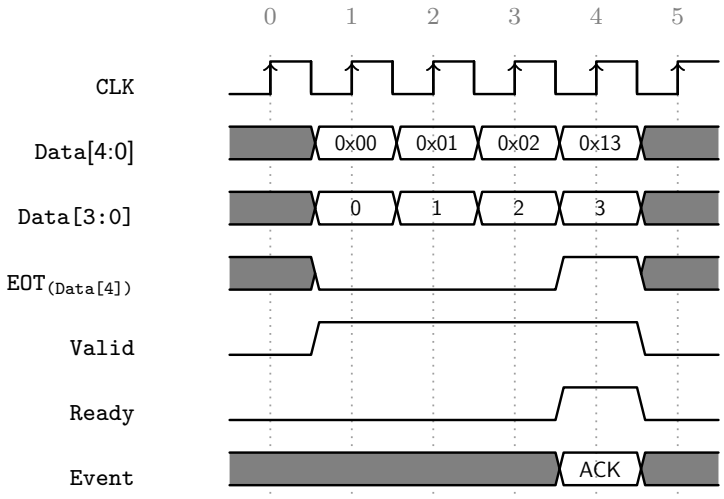
Float se koristi za predstavljanje decimalnih brojeva pomoću formata sa pokretnom tačkom. Ovaj tip se koristi isključivo za simulaciju i ne može se implementirati u hardveru.

6.5.6 Fixp

Fixp[WI, W] se koristi za predstavljanje decimalnih brojeva sa fiksnom tačkom.
Format Fixed point tipa je **WI** širina celobrojnog dela i **W** ukupna širina magistrale, širina decimalnog dela je razlika ove dve širine.
Ovaj tip se koristi za reprezentaciju decimalnih brojeva u hardveru.

6.5.7 Queue

Queue[T, LVL] ili red predstavlja transakciju koja sadrži više podataka proizvoljnog tipa T i informaciju o završetku transakcije EOT(end of transaction). Queue može biti proizvoljnog nivoa LVL.
Kao primer **Queue[Uint[4], 1](0, 1, 2, 3)** predstavlja red od četiri četvorobitna podatka, primer talasnih oblika na slici ispod. Rezultujuća Data magistrala biće širine 5 bita, 4 bita za podatak i 1 bit za EOT.

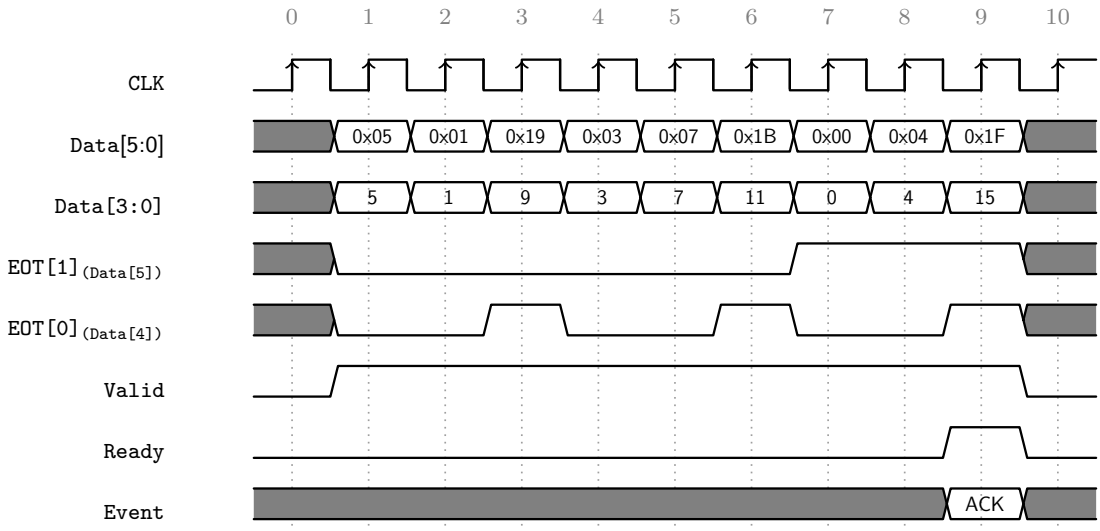


Slika 6.3: Primer transakcije tipa

Dvodimenzionalna matrica se može predstaviti kao Queue[Uint[4], 2], odnosno red nivoa 2 sa proizvoljnim podatkom u ovom slučajem četvorobitnim neoznačnim brojem.

	0	1	2
0	**5**	**1**	**9**
1	**3**	**7**	**11**
2	**0**	**4**	**15**

Slika 6.4: Matrica veličine 3x3



Slika 6.5: Transakcija matrice 6.4

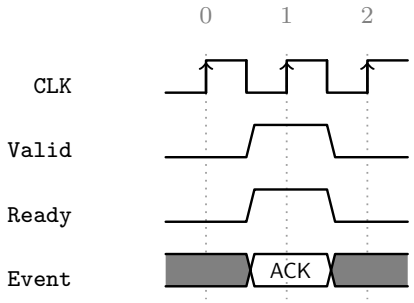
Kao što se može videti linija za podatke je u ovom slučaju širine 6 bita, od toga 2 bita su EOT.
Na slici(6.4) u taktovima(3, 6, 9) niži bit EOT-a je na visokom nivou što predstavlja podatak u poslednjoj koloni matrice. U taktovima(7, 8, 9) viši bit EOT-a je na visokom nivou što označava podatke u poslednjoj liniji matrice. Konačno u taktu 9 oba EOT bita su na visokom nivou što označava poslednji podatak u matrici.

6.5.8 Union

Union[T1, T2, ..., TN] je unija koja može prenositi samo jedan od podataka Tn u trenutku. Uz podatak prenosi se i informacija o aktivnom podatku na magistrali.

6.5.9 Unit

Unit je tip koji prenosi "prazan" podatak.



Slika 6.6: Unit tip

6.6 Čistoća Gear-ova

Kako bi se gear-ovi lakše povezivali i njihova funkcionalnost i ponašanje bilo razumljivo i predvidljivo dizajneru preporučuje se pisanje "Čistih" gear-ova.
"Čist" gear je onaj čije je inicijalno stanje dobro poznato i koji će se nakon izvršene funkcionalnosti potpuno vratiti u inicijalno stanje.
Čisto kombinacioni gear-ovi su uvek "čisti".

6.7 Definicija Gear komponenti

PyGears trenutno podržava tri načina za implementaciju Gear komponenti.

```
1 @gear
2 def gear_name(in1: T1, ..., inN: TN,
3                *,
4                p1=dflt, ..., pM=dfltM) -> ReturnType:
5
6     # Gear implementation
```

Primer 6.1: Primer definisanja Gear komponente

Primer(6.1) prikazuje definisanje gear komponente.
Dekorator **@gear** označava da je funkcija gear_name zapravo Gear komponenta, slično kao module ili entity kod Verilog-a i VHDL-a.
Kao parametri funkcije prosleđuju se DTI interfejsi i generički parametri. Delimiter "*" označava početak generičkih parametara.

Ulazni DTI interfejsi mogu imati proizvoljan naziv i tipove opisane u sekciji(6.5).
Parametar može biti bilo koji Python objekat i može imati inicijalnu vrednost dflt.

6.7.1 Gear implementiran pomoću SystemVerilog-a

Jedan od mogućih načina implementacije Gear komponente je pisanje SystemVerilog opisa pa zatim Python wrapper-a za komponentu.
Prilikom pisanja modula na ovaj način potrebno je pobrinuti se da napisana komponenta poštuje DTI protokol kao i da je "čist" Gear(6.6).

Prilikom pisanja wrapper-a za ovako implementiranu komponentu potrebno je unapred odrediti ReturnType koji će odgovarati izlaznom interfejsu SystemVerilog modula.
Takođe deo za implementaciju Gear-a u Python-u može ostati prazan.

Kako postoji velika verovatnoća unošenja bagova u dizajn prilikom ručnog pisanja modula koji poštuje DTI interfejs i koji je "čist", ovaj način implementacije modula nije preporučen.

6.7.2 Gear implementiram kompozicijom

Kako PyGears dolazi sa bibliotekom osnovnih funkcionalnih modula, većina potrebnih funkcionalnosti moguće je ostvariti njihovom kompozicijom.

6.7.3 Gear implementiram Python to HDL kompajlerom

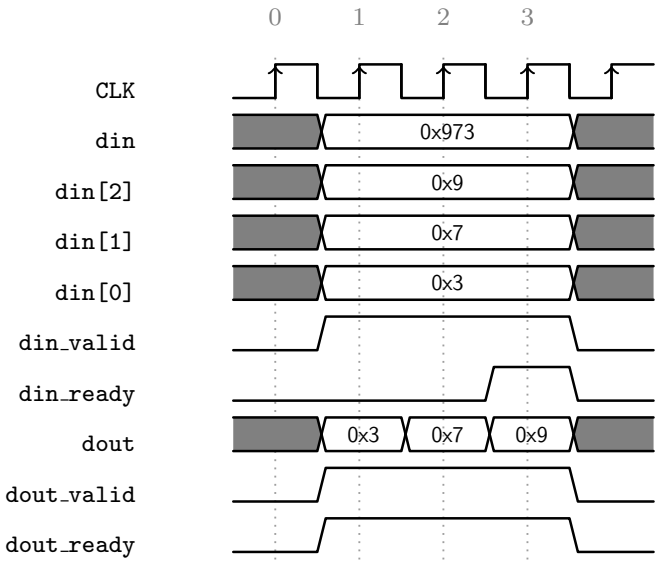
Moguće je i pisati opis gear-a u Python-u pomoću HDL kompajlera.

Kao primer prikazan je serialize gear koji kao ulaz prima niz podataka, a na izlazu daje jedan po jedan podatak ulaznog niza, odnosno serializuje podatke.

```
1 @gear(svgen={'compile': True})
2 async def serialize(din: Array) -> b'din.dtype':
3     async with din as val:
4         for i in range(len(din.dtype)):
5             yield val[i]
```

Primer 6.2: Primer kompajliranog serialize gear-a

Kao primer možemo uzeti niz **Array[Uint[4], 3](3, 7, 9)** za očekivati je da će se na izlazu pojaviti podaci redom 3, 7 pa 9 što je prikazano na slici 6.7.



Slika 6.7: Serialize gear primer

7 Integracija IP jezgra sa Zynq sistemom

PyGears i SystemVerilog implementacije predložene arhitekture su kompatibilne u pogledu spoljnih interfejsa. Pored toga ove implementacije su uveliko nezavisne od ciljane tehnologije za implementaciju.
U ovom radu IP jezgro će biti implementirano na Zynq System on Chip (SoC) sistemu.

Korišćeni DTI interfejs je kompatibilan sa AXI-Stream intefejsom, tako da je integracija sa sistemima koji koriste druge interfejse poput Avalon-a trebala biti jednostavna uz adaptere.

Preporučena je integracija IP jezgra u sistem sa procesorom, što će biti i prikazano u ovom radu. U ovom slučaju biće korišćen ARM Cortex-A9 procesor koji se može naći kao Hard IP jezgro unutar Zynq SoC platforme kompanije Xilinx.

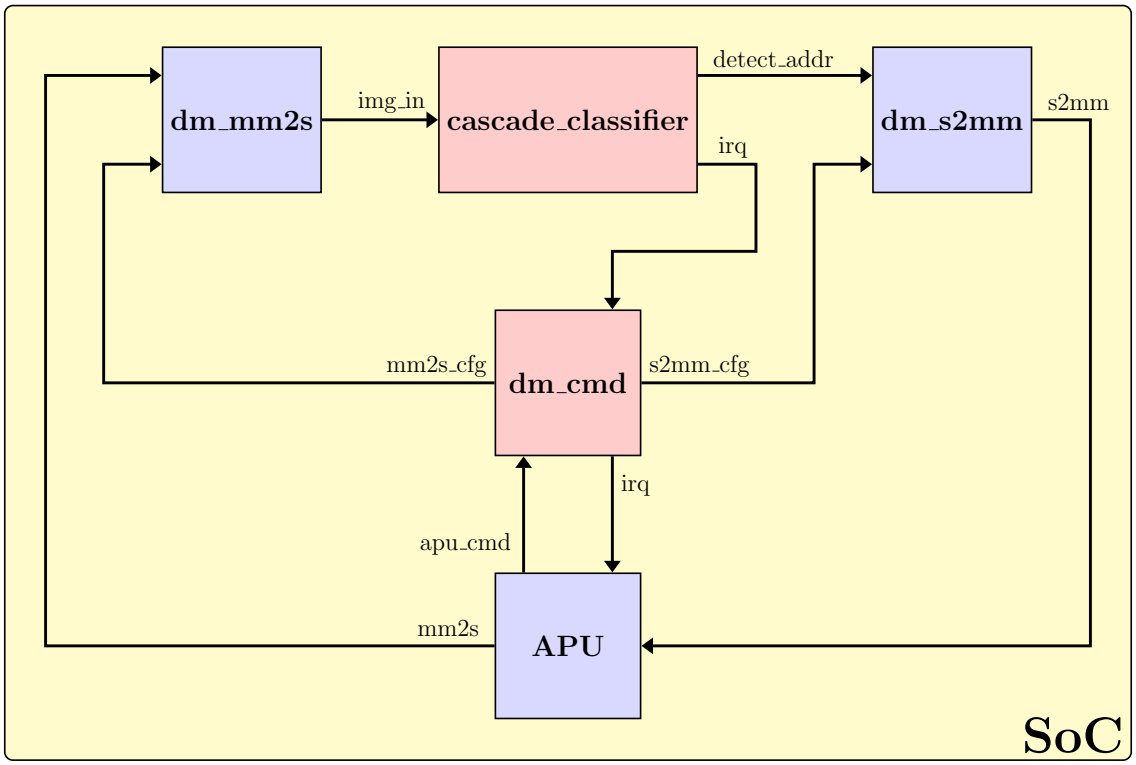
7.1 Zynq

Zynq se sastoji od Processing System (PS) sistem i Programmable Logic (PL) sistema. PS sistem je sačinjen od već pomenutog ARM Cortex-A9 procesora. Dok PL sistem predstavlja Field Programmable Gate Array (FPGA) programabilnu logiku Artix-7 familije.

Komunikacija između PL i PS sistema se obavlja preko Advanced eXtensible Interface (AXI) interfejsa, interapt pinova ili Extended multiplexed I/O (EMIO) pinova.

ARM procesor ima mogućnost povezivanja eksterne Double Data Rate (DDR) memorije preko internog DDR kontrolera. Preko ovog kontrolera pristup memoriji ima i PL sistem.
Zynq SoC ima podršku GNU/Linux operativnog sistema.

7.2 Predloženi blok dijagram sistema



Slika 7.1: Blok dijagram stddev modula.

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

[9] B. Vukobratović, A. Erdeljan, and D. Rakanović, "Pygears: A functional approach to hardwaredesign," 2019. [Online]. Available: https://osda.gitlab.io/19/1.3.pdf

[10] Xilinx, "Axi reference guide." [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[11] Intel, "Avalon® interface specifications." [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf

[12] B. Vukobratović, "Pygears." [Online]. Available: www.pygears.org

[13] P. P. Chu, *RTL hardware design using VHDL: coding for efficiency, portability, and scalability.* John Wiley & Sons, 2006.

[14] R. Struharik, "Rt metodologija projektovanja složenih digitalnih sistema." [Online]. Available: https://www.elektronika.ftn.uns.ac.rs/projektovanje-slozenih-digitalnih-sistema/wp-content/uploads/sites/120/2018/03/Predavanje-5-RT-Methodology.pdf

[15] "Symbiflow." [Online]. Available: https://symbiflow.github.io

[16] J. Decaluwe, "Myhdl: a python-based hardware description language." *Linux journal*, no. 127, pp. 84–87, 2004.

[17] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. IEEE, 2012, pp. 1212–1221.

[18] "Spinalhdl," https://github.com/SpinalHDL/SpinalHDL, accessed: 2018-08-12.

[19] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "C? ash: Structural descriptions of synchronous hardware using haskell," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*. IEEE, 2010, pp. 714–721.