

Hardverska implementacija Viola-Jones algoritma

Kandidat
Risto Pejašinović EE19/2015

Mentor

28. septembar 2019

Sadržaj

I	Viola-Jones algoritam	4
1	Uvod	4
1.1	Integralna slika	4
1.2	AdaBoost i HAAR obeležja	6
1.2.1	HAAR obeležja	6
1.2.2	AdaBoost	7
1.3	Kaskadni klasifikator	8
1.4	Skaliranje slike	9
1.5	Osetljivost na osvetljaj	11
1.6	Osetljivost na rotaciju objekta	11
2	OpenCV modeli	12
2.1	OpenCV model za frontalna lica	12
II	Hardverska implementacija	13
3	Sažetak	13
4	Specifikacije za izvršavanje	14
5	Arhitektura hardvera	16
5.1	Uvod	16
5.2	Interfejsi IP jezgra	16
5.3	Modul IMG RAM	17
5.4	Modul rd_addrngen	17
5.4.1	Scale_counter	18
5.4.2	Boundaries	18
5.4.3	Scale_ratio	18
5.4.4	Hopper	18
5.4.5	Sweeper	20
5.4.6	Skaliranje adrese	21
5.4.7	Modul addr_trans	21
5.5	Modul ii_gen i sii_gen	22
5.5.1	Odabir algoritma	22
5.5.2	Sekvencijalna implementacija generatora integralne slike	22
5.5.3	Generator kvadratne integralne slike	23
5.6	Modul frame_buffer	24
5.7	Modul stddev	25
5.8	Modul features_mem	26
5.9	Modul classifier	29
5.10	Interfejsi	31

6	PyGears metodologija	32
6.1	Uvod	32
6.2	Poređenje sa RTL metodologijom	32
6.3	Jezici za opis hardvera	32
6.4	Gears metodologija	33
6.4.1	DTI interfejs	33
6.5	Tipovi podataka	34
6.5.1	UInt	35
6.5.2	Int	35
6.5.3	Tuple	35
6.5.4	Array	35
6.5.5	Queue	35
6.5.6	Union	37
6.5.7	Unit	37
6.6	Čistoća Gear-ova	38
6.7	Definicija Gear komponenti	38
6.7.1	Gear implementiran pomoću SystemVerilog-a	38
6.7.2	Gear implementiram kompozicijom	39
6.7.3	Gear implementiram Python to HDL kompajlerom	39
6.8	Python kao jezik za verifikaciju	39

Mali rečník

Akronim	Opis
AdaBoost	= Adaptive Boosting

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) .

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

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

1	1	1
1	1	1
1	1	1

Ulazna slika

1	2	3
2	4	6
3	6	9

Integralna 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 oduzimanja i jednim sabiranjem.

Originalna	Integralna
5 2 3 4 1	5 7 10 14 15
1 5 4 2 3	6 13 20 26 30
2 2 1 3 4	8 17 25 34 42
3 5 6 4 5	11 25 39 52 65
4 1 3 2 6	15 30 47 62 81

$$5 + 4 + 2 + 2 + 1 + 3 = 17$$

$$(D) - (B) - (C) + (A) = S$$

$$34 - 14 - 8 + 5 = 17$$

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.

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

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.

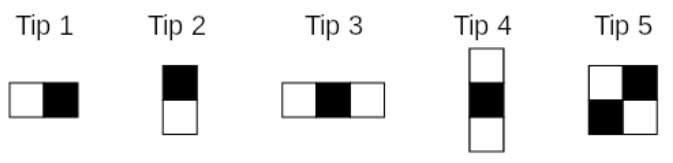
Integralnu sliku je moguće računati u paraleli ili sekvencijalno. Izbor algoritma za računanje integralne slike značajno utiče na performanse i potrebne hardverske resurse konačne implementacije.

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 zbroja 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 dobiti 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*

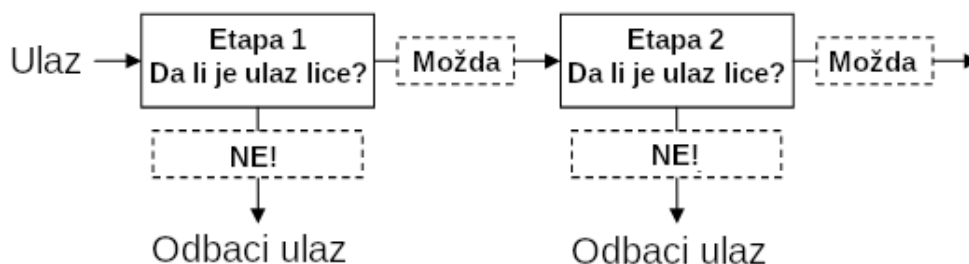
Kao rezultat AdaBoost algoritma dobijamo skup optimalnih obeležja, u ovom radu se koristi model sa skupom od 2913 obeležja.

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.

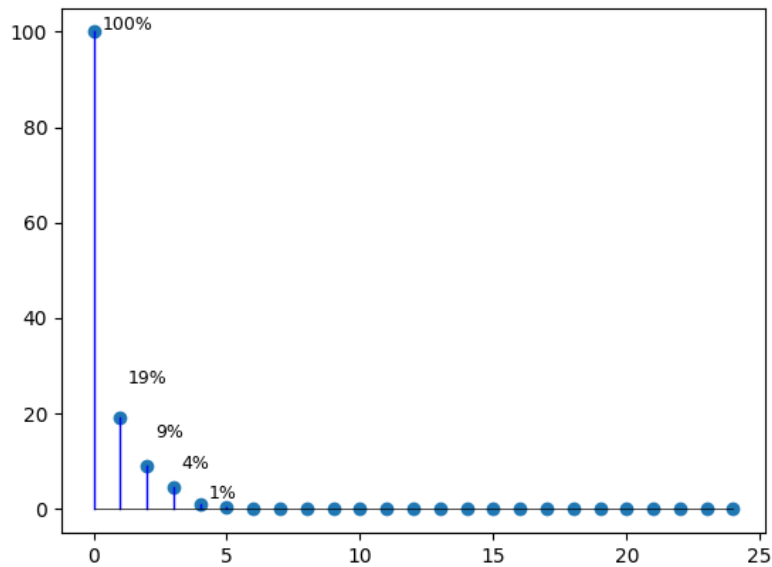
Konačno ukoliko sve etape u klasifikatoru na analiziranom prozoru daju rezultat da se na njemu možda nalazi lice može se zaključiti da se na toj poziciji zaista nalazi lice.

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...

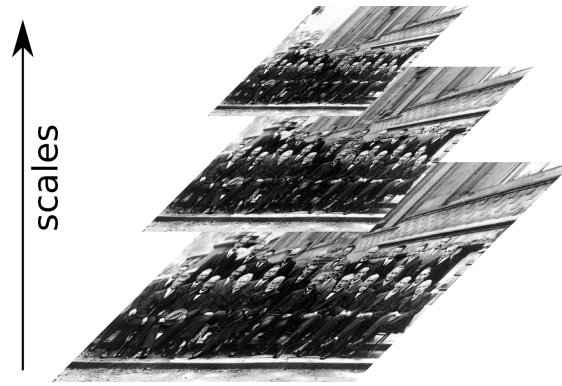
Vidimo da je posle pete etape procenat izvršavanja manji od 1%.

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.



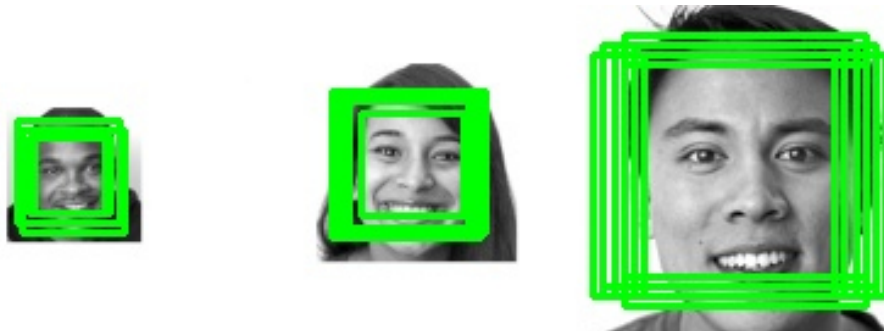
Slika 1.7: Piramida slike[6]

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]

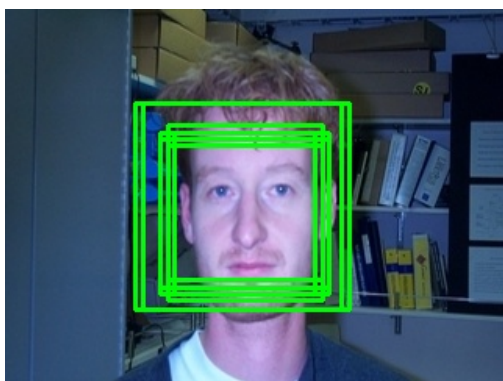
Na slikama(1.9, 1.10) su prikazana dva lica koja zbog nepovoljnog osvetljenja nisu detektovane od strane klasifikatora. Slika(1.9) nije detektovana zbog pristustva prejakog osvetljenja lica, dok slika(1.10) nije detektovana zbog slabog osvetljenja.

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 klasifikatora¹. [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>

²https://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 PyGears³ metodologiji.
- Poređenje dve metodologije i analiza prednosti i mana obe metodologije.
- Implementacija projektovanog IP jezgra na MYIR Z-Turn Board⁴ 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>

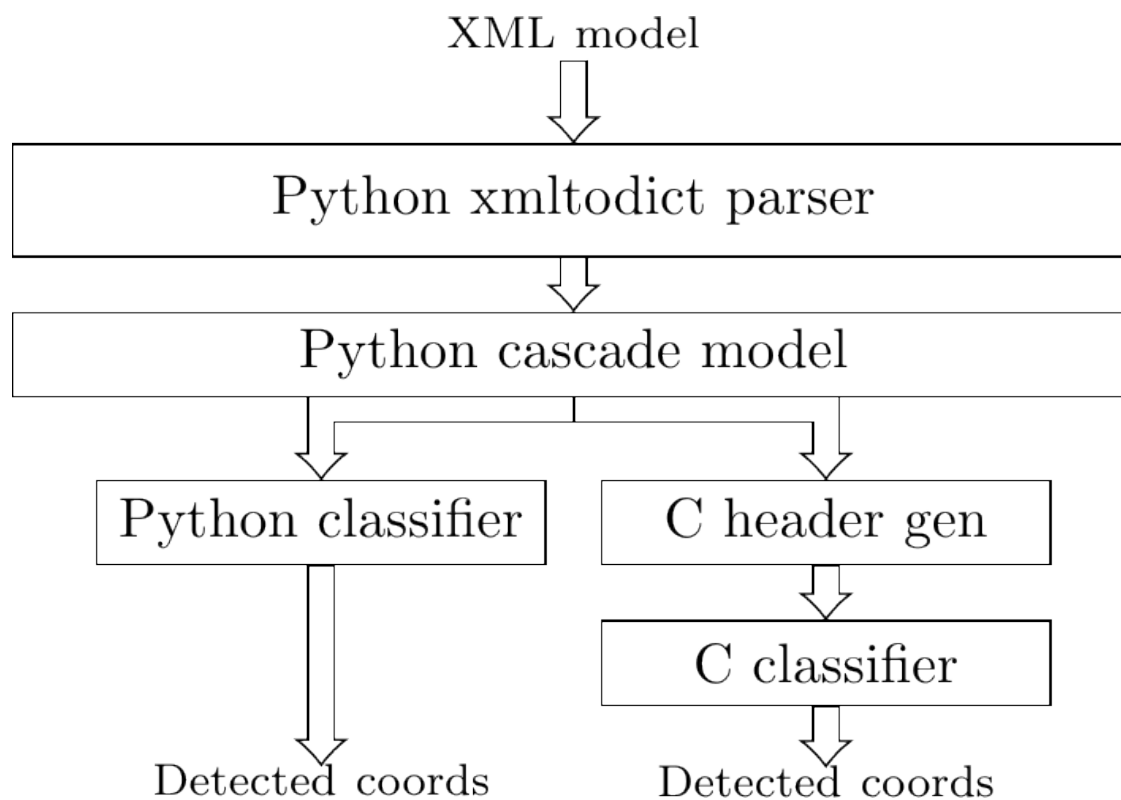
⁴<http://www.myirtech.com/list.asp?id=502>

4 Specifikacije za izvršavanje

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 paralelna 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 xmldict⁵.

Xmldict parsira XML fajl i skladišti ga u Python dictionary.

⁵<https://pypi.org/project/xmldict/>

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`

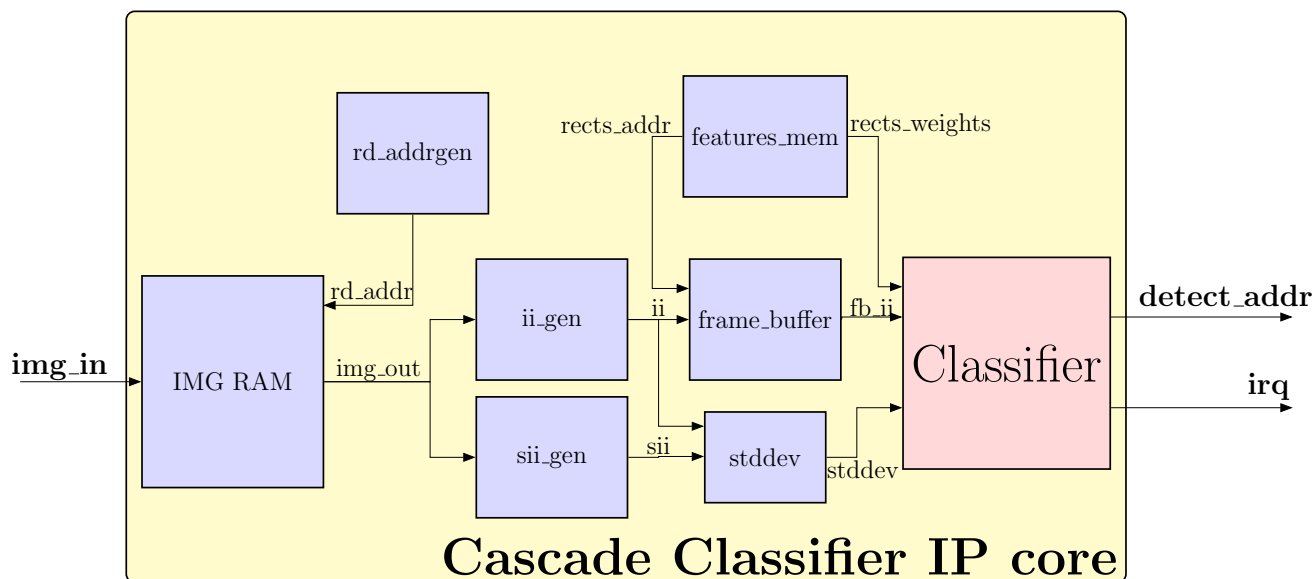
5 Arhitektura hardvera

5.1 Uvod

U ovom poglavlju biće opisana 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 systemske 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. Namijenjen 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 povećava za 1.

Skladištena slika je u 8-bitnom formatu i predstavlja grayscale vrednost piksela originalne slike.

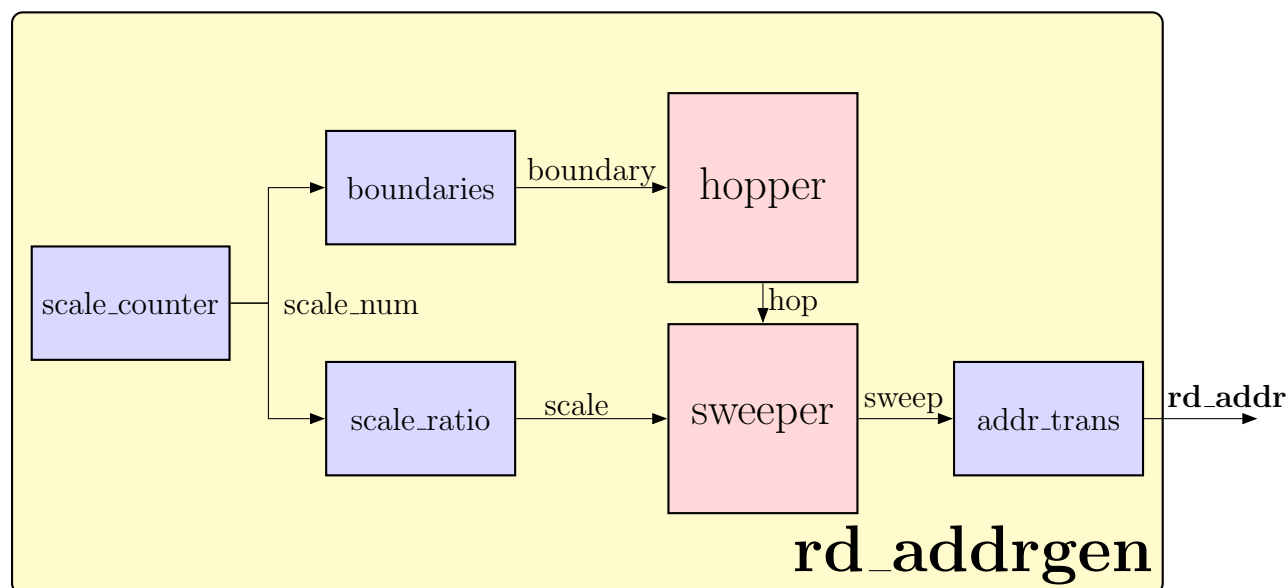
Veličina memorije je $width * height * 8$ bit. Za dimenziju slike 240x320 veličina memorije je 614400 bita.

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 ugnježena 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

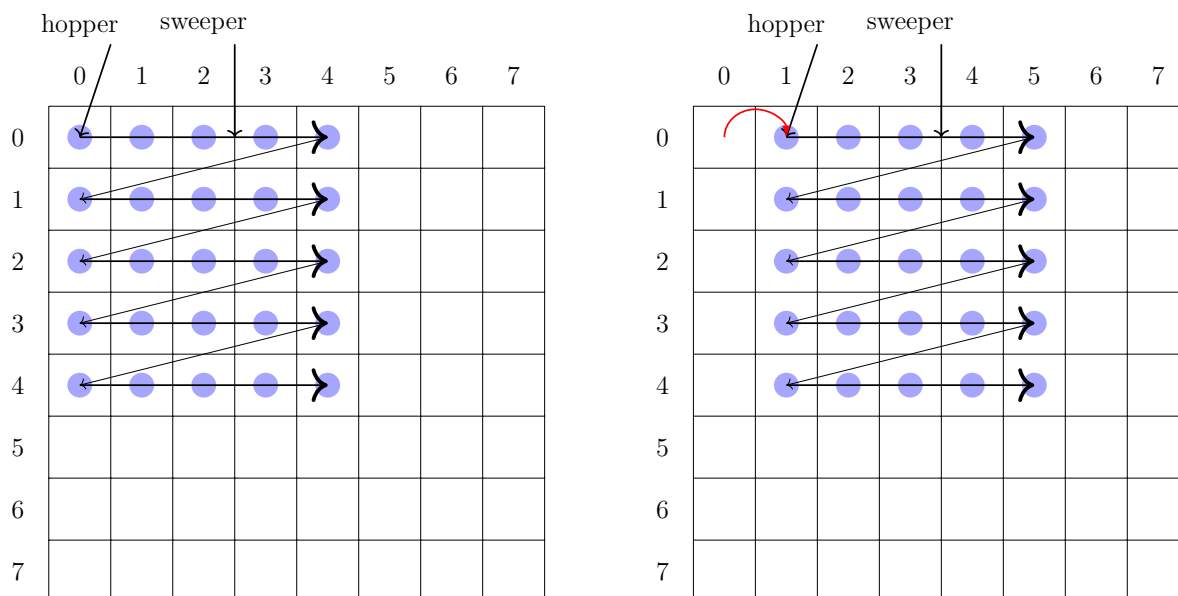
Na primeru (5.1) prikazana je implementacija **hopper**-a u C-u. Može se videti da granice **hopper**-a zavise od promenljivih **boundary_x** i **boundary_y** koji se indeksiraju preko **scale_num** promenljive opisane u sekcijama (5.4.2, 5.4.1).

Na slikama (5.3, 5.4) je prikazan rad **hopper**-a i **sweeper**-a. Plavi kružići predstavljaju piksele za koje će **rd_addrngen** 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.

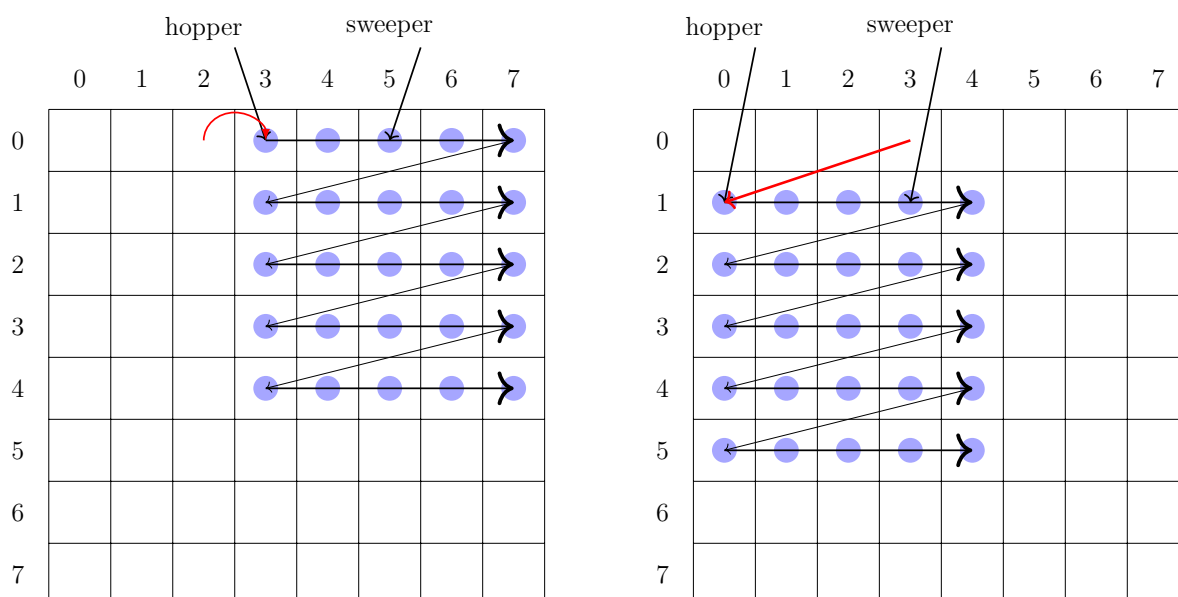
Na slici (5.4) može se videti trenutak kada **hopper** dostiže granicu **boundary_x** nakon čega

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.

	0	1	2	3	4	5	6	7	8	9
0	●	●	●		●	●	●		●	●
1	●	●	●		●	●	●		●	●
2	●	●	●		●	●	●		●	●
3										
4	●	●	●		●	●	●		●	●
5	●	●	●		●	●	●		●	●
6	●	●	●		●	●	●		●	●
7										
8	●	●	●		●	●	●		●	●
9	●	●	●		●	●	●		●	●

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 sledeće formule.

$$lin_addr = (y * img_width) + x \quad (5.1)$$

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.

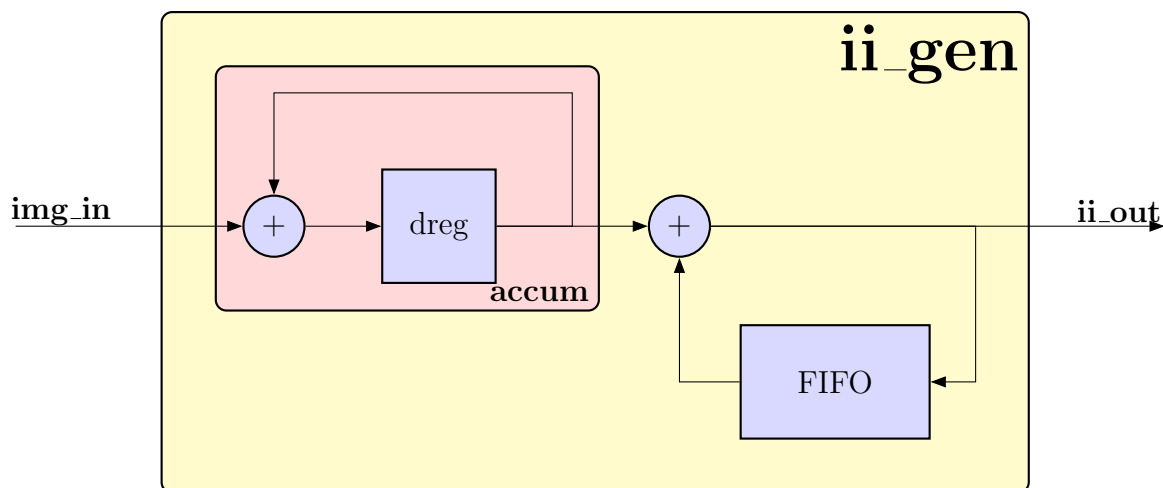
Isto tako se ispostavlja da i u hardverskoj implementaciji generisanje integralne slike ima veliki uticaj na performanse i potrebne hardverske resurse sistema.

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.

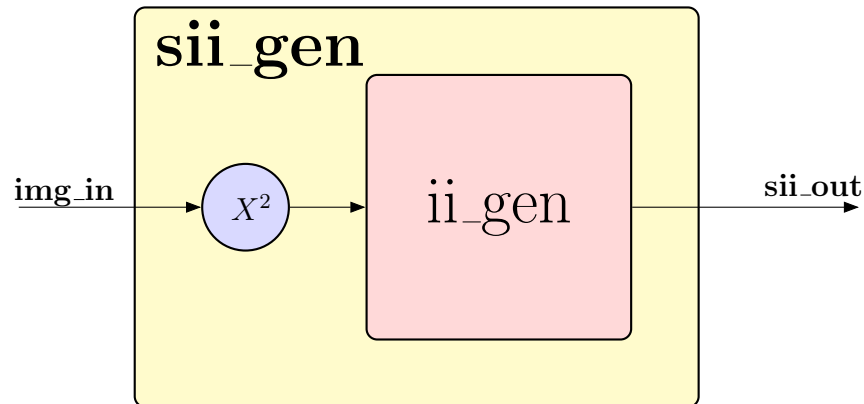
Zbog potreba algoritma FIFO bafer je modifikovan na sledeći način:

- 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.

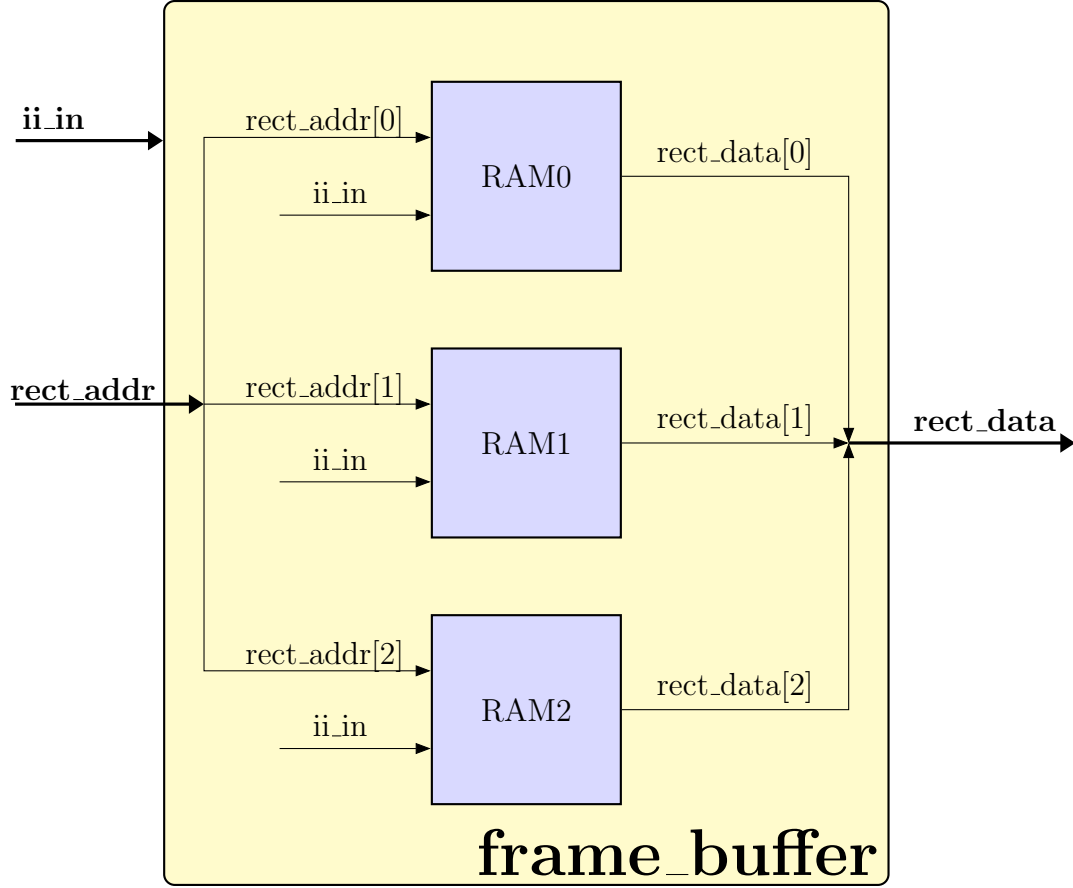
Generisanje kvadratne integralne slike je jednostavno uz gotov generator integralne slike. Potrebno je kvadrirati ulazne piksele i dovesti ih na generator integralne slike kao na slici(5.7).



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 inkrementuje za jedan nakon svakog primljenog podatka.

Potrebna veličina RAM memorije je data sledećim vezama:

$$size(bit) = frame_width * frame_height * w_ii \quad (5.2)$$

$$w_ii = \lceil \log_2(frame_width * frame_height * 2^{w_img}) \rceil \quad (5.3)$$

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).

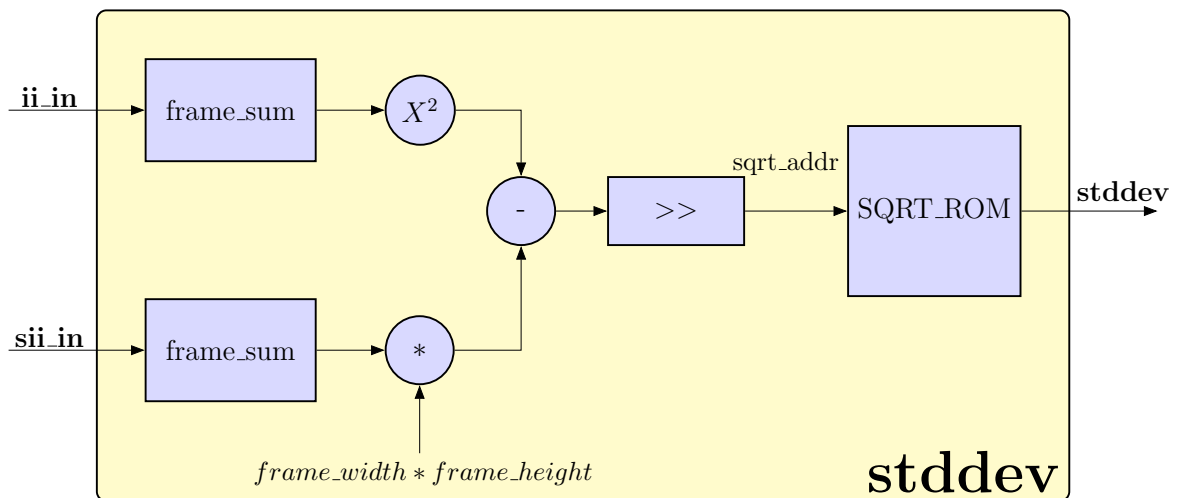
Sabiranjem gornjeg levog i donjeg desnog piksela zatim oduzimanje gornjeg desnog i donjeg levog integralne slike dobijamo sumu piksela celog prozora.

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 obavljaју 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.

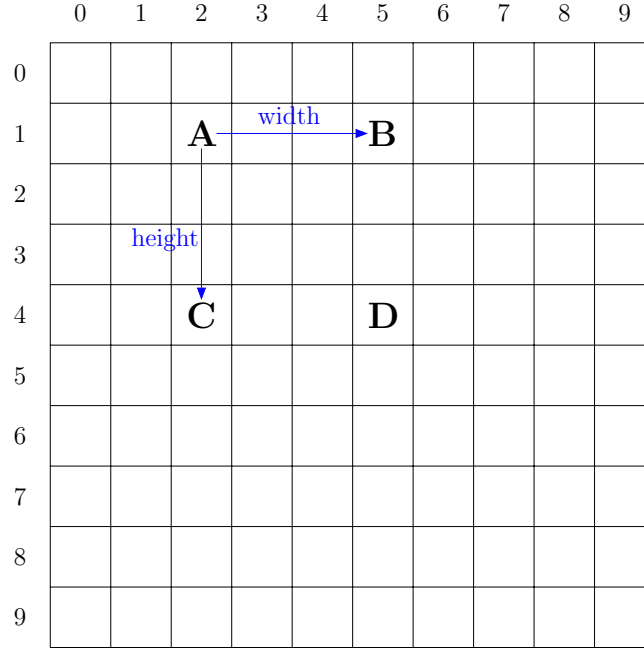
5.8 Modul features_mem

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:

$$B = A * width \quad (5.4)$$

$$D = (A + width) + (height * FRAME_WIDTH) \quad (5.5)$$

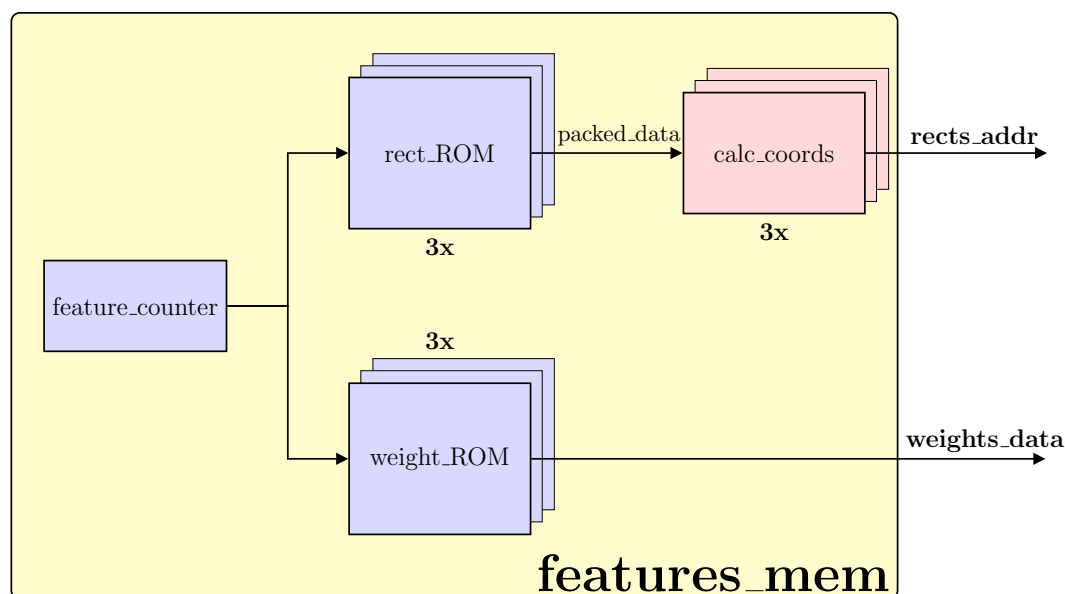
$$C = (A + width) + (height * FRAME_WIDTH) - width \quad (5.6)$$

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 struktura 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).

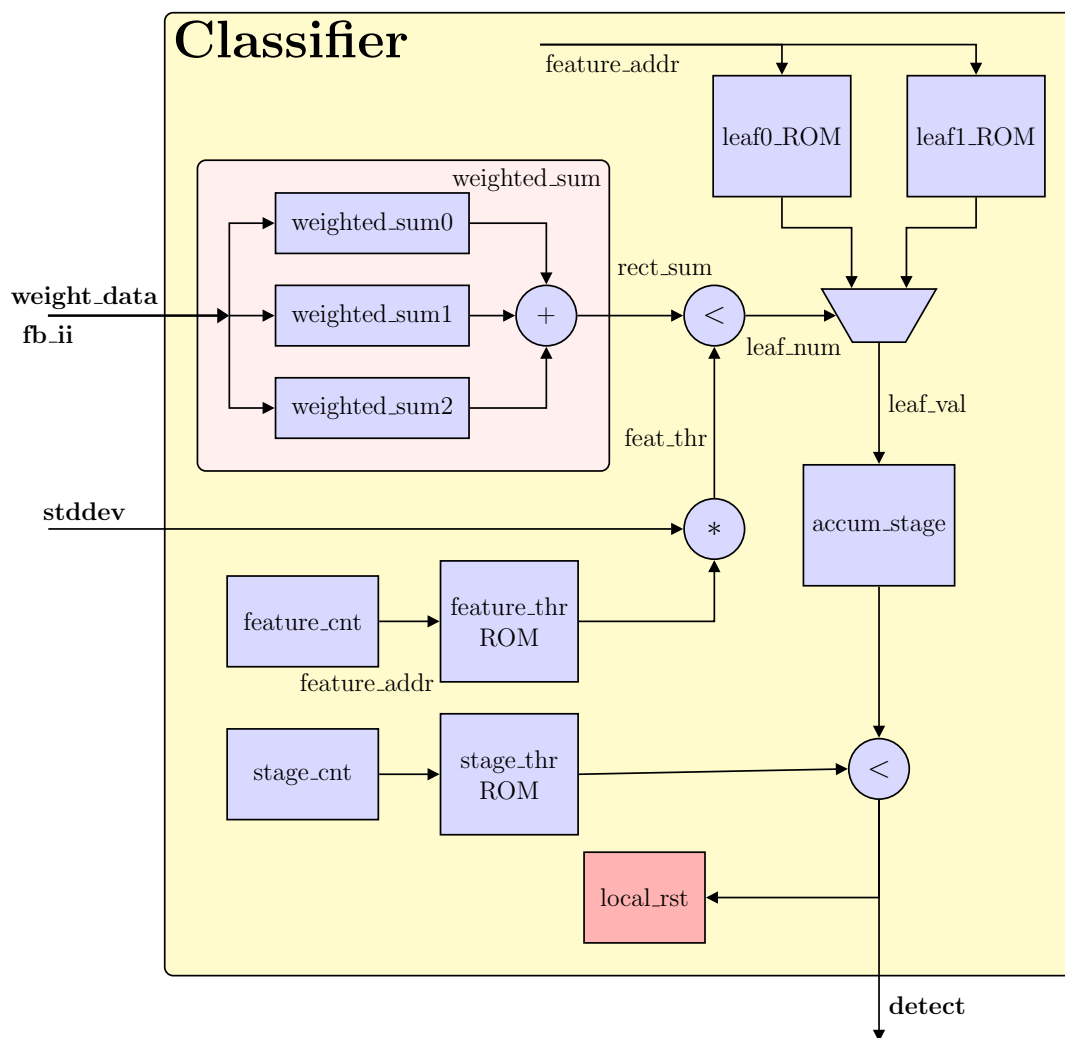
Konačno može se izračunati potrebna memorija u slučaju modela opisanog u sekciji(2.1).

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 od-uzeti 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 memorije leaf_val0, u suprotnom povratna vrednost će biti iz memorije leaf_val1.

5.10 Interfejsi

Komunikacija svih unutrašnjih modula i komunikacija IP jezgra sa okolinom realizovana je pomoću **DTI**(data transfer interface)[9] interfejsa. Korišćenjem 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] je open-source projekat koji uvodi Gears metodologiju za projektovanje hardvera i Python Framework koji implementira Gears metodologiju.

Gears metodologija uvodi lako povezivanje i kompozabilnost sistema od manjih funkcionalnih jedinica pod nazivom gear-ovi.

Gear moduli su međusobno povezani DTI interfejsom koji implementira jednostavan handshake protokol. Na ovaj način je omogućeno je lako ponovno korišćenje implementiranih funkcija i jednostavno debugovanje dizajna.

Pisanjem dizajna u Python-u koji je dinamičan jezik i veoma visokog nivoa apstrakcije omogućena je dodatno bolje parametrizovanje, skalabilnost i kompozabilnost dizajna.

Takođe omogućeno je pisanje verifikacije u Python-u na veoma visokom nivou apstrakcije uz korišćenje ogromnog broja besplatno dostupnih Python paketa za većinu problema i algoritama, time je skraćeno vreme i olakšan posao prilikom pisanja referentnih modela.

PyGears konačno obavlja konverziju Python opisa u SystemVerilog opis koji je podržan od strane alata za sintezi i simulaciju.

6.2 Poređenje sa RTL metodologijom

PyGears[9] predstavlja alternativu RTL[13] metodologiji za opis hardvera. RTL metodologija pruža standardan način translacije sekvencijalnog algoritma u hardware opis. Struktura dobijena pomoću RTL metodologije uglavnom se sastoji od **FSMD**(Finite State Machine with Datapath).

Datapath deo koji sadrži funkcionalne i memorijske jedinice i mreže za rutiranje podataka[14] Mašina stanje koja sadrži registre stanja, logiku za izlazne signale i logiku za sledeće stanje.

Kako dizajn implementiran pomoću RTL metodologije postaje kompleksniji i mašina stanja postaje kompleksnija i sadrži više stanja. Pipeline-ovanje, debugovanje 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 u 1984. i 1983. godine i daleko su siromašniji po mogućnostima od današnjih jezika višeg nivoa kao što su Python, C++, itd.

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 detaljno istestirani.

Sve velike kompanije koje proizvode FPGA čipove trenutno ne objavljuju arhitekturu svojih FPGA čipova, pa alati za sintezu i implementaciju zavise od tih kompanija⁷. 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.

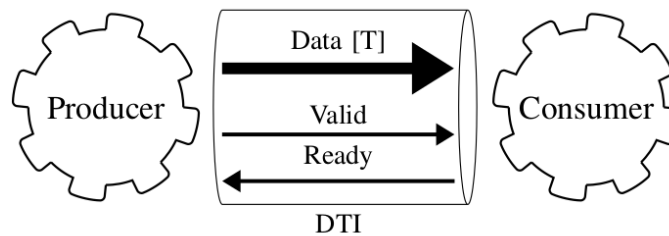
Dok se ovi jezici uglavnom baziraju na RTL metodologiji i uvode jezik višeg nivoa kao zamenu za standardan HDL. PyGears dodatno izdvaja i uvođenje nove metodologije zvane Gears.

6.4 Gears metodologija

Cilj Gears metodologije je da se obezbedi bolja kompozabilnost i ponovno korišćenje napisanih modula.

Kako bi se ovo postiglo uveden je standardan interfejs za komunikaciju između modula.

6.4.1 DTI interfejs



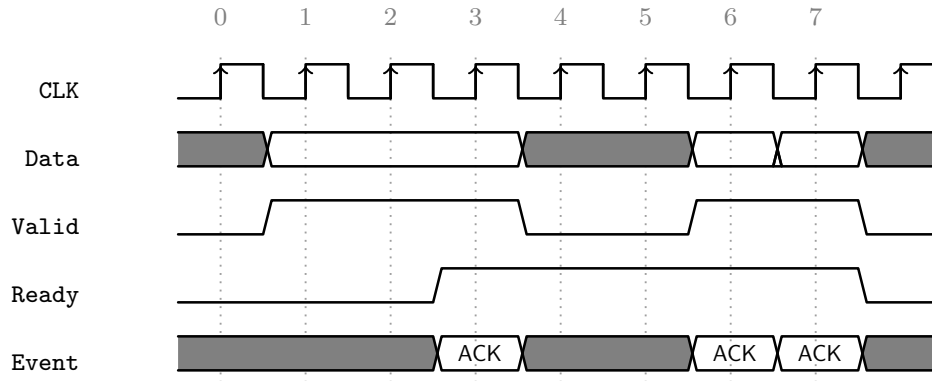
Slika 6.1: DTI interfejs[9]

Uveden je interfejs **DTI**(data transfer interface). Podaci se šalju od strane Producer-a prema Consumer-u. Sastoji se od:

- **Data** linije za podatke, proizvoljne širine i proizvoljnog tipa.
- **Valid** jednobitnog signala kojim Producer signalizira validnost podatka na Data liniji.
- **Ready** jednobitnog signala kojim Consumer signalizira da je pročitao podatak na Data liniji.

Pomoću Valid i Ready signala realizovan je handshaking protokol.

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



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 na magistrali 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 handshake obavljen.
Nakon handshake-a Producer može postaviti novi podatak na magistralu.
5. Producer ne sme menjati podatak na magistrali sve do pojave ACK od strane Consumer-a.
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 Producer-a. Odnosno Producer ne sme odlučivati o postavljanju podatka na magistralu 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.

Neki od podržanih tipova su:

- **UInt[W]** 6.5.1

- **Int**[W] 6.5.2
- **Tuple**[T1, T2, ..., TN] 6.5.3
- **Array**[T, N] 6.5.4
- **Queue**[T, LVL] 6.5.5
- **Union**[T1, T2, ..., TN] 6.5.6
- **Unit** 6.5.7

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.

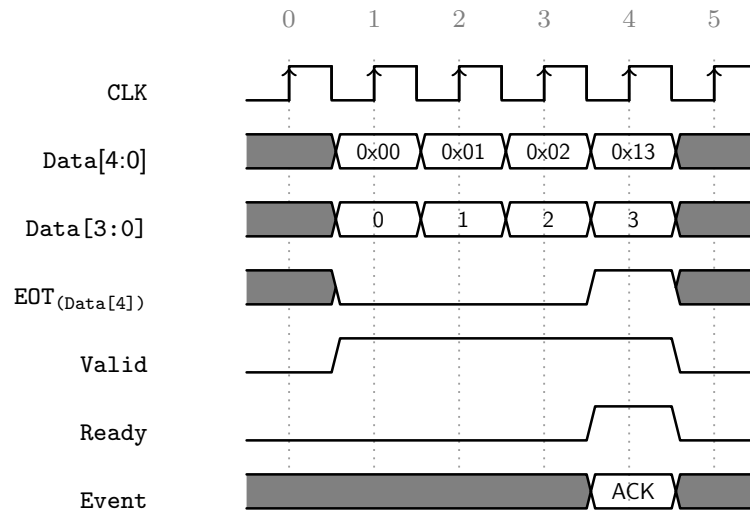
6.5.4 Array

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

6.5.5 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. Rezultujuća Data magistrala biće širine 5 bita, 4 bita za podatak i 1 bit za EOT.

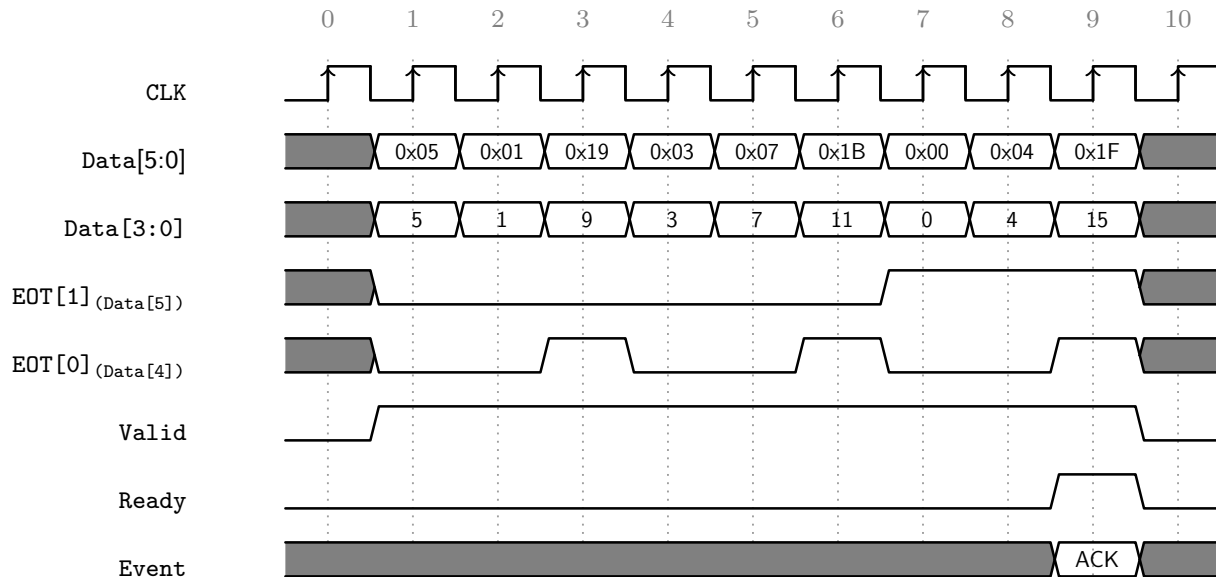


Slika 6.3: Primer transakcije tipa

Kao primer dvodimenzionalna matrica se može predstaviti kao `Queue[UInt[4], 2]`, odnosno red nivoa 2 sa proizvoljnim podatkom u ovom slučaju četvorobitnim neoznačnim brojem.

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

Slika 6.4: Matrice 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.

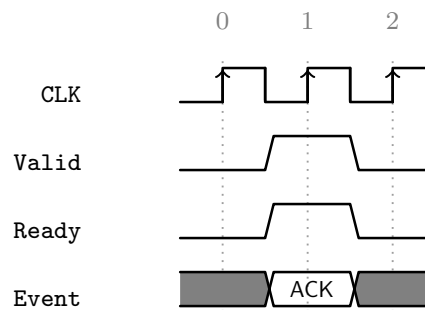
Na slici(6.4) u taktovima(3, 6, 9) niži bit EOT-a je na visokom nivou što predstavlja podatak u poslednjoj koloni matrice. U taktovima(7, 8, 9) viši bit EOT-a je na visokom nivou što označava podatke u poslednjoj liniji matrice. Konačno u taktu 9 oba EOT bita su na visokom nivou što označava poslednji podatak u matrici.

6.5.6 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.

6.5.7 Unit

Unit je tip koji ne prenosi 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 dobijena 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

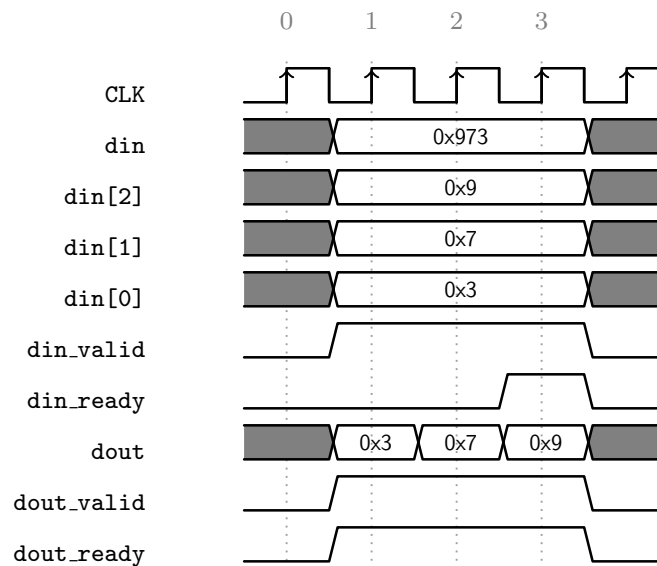
Moguće je i pisati opis gear-a u Python-u pomoću HDL kompajlera.

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 primeru 6.7.



Slika 6.7: Serialize gear primer

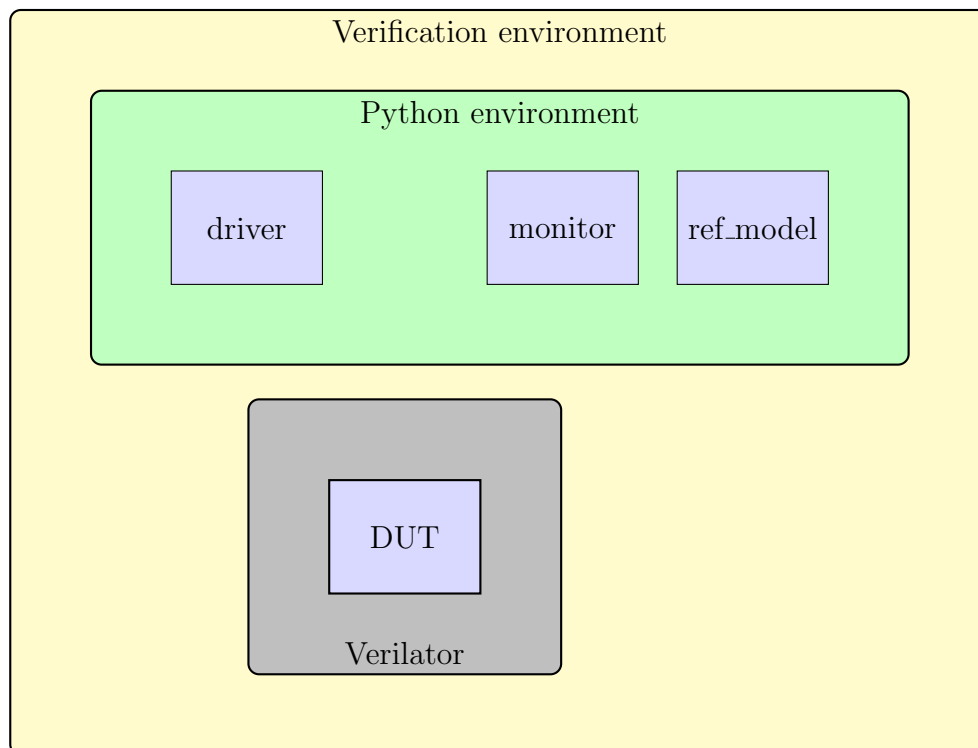
6.8 Python kao jezik za verifikaciju

Pored alata za pisanje opisa hardvera PyGears pruža mogućnost verifikacije i simulacije. Pomoću interfejsa sa besplatnim Verilator simulatorom i SystemVerilog DPI[20] interfejsom za komercijalne simulatore(Questa[21], NCSim[22]), moguće je pisanje verifikacionog

okruženja u Python-u koje komunicira sa ovim simulatorima.

Ovakvo verifikaciono okruženje ima prednosti u odnosu na SystemVerilog ili VHDL okruženje:

- Python je danas široko rasprostranjen i zajednica ima daleko više korisnika nego SystemVerilog ili VHDL. Kao posledica toga dostupno je mnoštvo paketa sa implementiranim algoritmima za većinu problema, tako da je pisanje referentnih modela značajno olakšano korišćenjem gotovih biblioteka.
- Manipulacija kompleksnih tipova podataka je daleko lakša u Python-u od VHDL-a i SystemVerilog-a, tako da je razvoj okruženja daleko brži.
- Verifikaciono okruženje se ne simulira zajedno sa HDL modelom time se dobijaju bolje performanse verifikacije.



Slika 6.8: Python verifikaciono okruženje sa Verilatorom

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 hardware design,” 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 slozenih 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.
- [20] Wikipedia, “Systemverilog dpi wikipedia.” [Online]. Available: https://en.wikipedia.org/wiki/SystemVerilog_DPI
- [21] MentorGraphics, “Questa simulator.” [Online]. Available: <https://www.mentor.com/products/fv/questa/>
- [22] Cadence, “Ncsim simulator.” [Online]. Available: https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html