



UNIVERZITET U NOVOM SADU  
**FAKULTET TEHNIČKIH NAUKA**  
KATEDRA ZA ELEKTRONIKU




Risto Pejašinović

# **Hardverska implementacija Viola-Jones algoritma**

ZAVRŠNI RAD  
-Osnovne akademske studije-

Novi Sad, 2019.

	UNIVERZITET U NOVOM SADU <b>FAKULTET TEHNIČKIH NAUKA</b> 21000 NOVI SAD , Trg Dositeja Obradovića 6	Broj:
	<b>ZADATAK ZA ZAVRŠNI (BACHELOR) RAD</b>	Datum:

(Podatke unosi predmetni nastavnik - mentor)

Vrsta studija:	Osnovne akademske studije
Studijski program:	Energetika, elektronika i telekomunikacije
Rukovodilac studijskog programa:	Dr Milan Sečujski, vanredni profesor

Student:	Risto Pejašinović	Broj indeksa:	EE19/2015
Oblast:	Projektovanje Složenih Digitalnih Sistema		
Mentor:	dr Vuk Vranković, docent		

NA OSNOVU PODNETE PRIJAVE, PRILOŽENE DOKUMENTACIJE I ODREDBI STATUTA FAKULTETA IZDAJE SE ZADATAK ZA ZAVRŠNI (Bachelor) RAD, SA SLEDEĆIM ELEMENTIMA:

- problem – tema rada;
- način rešavanja problema i način praktične provere rezultata rada, ako je takva provera neophodna;
- literatura

### NASLOV ZAVRŠNOG (BACHELOR) RADA:

Hardverska implementacija Viola-Jones algoritma.

### TEKST ZADATKA:

1. Teorijski uvod u Viola-Jones algoritam, njegove prednosti i mane.
2. Razvoj softverskih modela Viola-Jones algoritma u svrhu projektovanja hardverske arhitekture.
3. Projektovanje hardverske arhitekture akceleratora za Viola-Jones algoritam.
4. Implementacija projektovane arhitekture u SystemVerilog jeziku, kao i pomoću PyGears metodologije.
5. Integracija projektovanog IP jezgra sa Zynq 7020 SoC platformom.
6. Pisanje Linux Device Driver-a za komunikaciju sa projektovanim IP jezgrom, pisanje korisničkih aplikacija.
7. Analiza performansi i potrebnih hardverskih resursa za projektovano IP jezgro.

Rukovodilac studijskog programa:	Mentor rada:
dr Milan Sečujski	dr Vuk Vranković

Primerak za: O- Studenta; O- Studentsku službu fakulteta



UNIVERZITET U NOVOM SADU • FAKULTET TEHNIČKIH NAUKA  
21000 Novi Sad, Trg Dositeja Obradovića 6

### KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj, <b>RBR:</b>		
Identifikacioni broj, <b>IBR:</b>		
Tip dokumentacije, <b>TD:</b>		Monografska dokumentacija
Tip zapisa, <b>TZ:</b>		Tekstualni štampani materijal
Vrsta rada, <b>VR:</b>		Diplomski rad
Autor, <b>AU:</b>		Risto Pejašinović
Mentor, <b>MN:</b>		Prof. dr Vuk Vranković
Naslov rada, <b>NR:</b>		Hardverska implementacija Viola-Jones algoritma
Jezik publikacije, <b>JP:</b>		Srpski
Jezik izvoda, <b>Jl:</b>		Srpski
Zemlja publikovanja, <b>ZP:</b>		Srbija
Uže geografsko područje, <b>UGP:</b>		Vojvodina
Godina, <b>GO:</b>		2019
Izdavač, <b>IZ:</b>		Autorski reprint
Mesto i adresa, <b>MA:</b>		21000 Novi Sad, Trg Dositeja Obradovića 6
Fizički opis rada, <b>FO:</b> (poglavlja/strana/citata/tabela/slika/grafika/priloga)		(8/63/23/6/38/0/0)
Naučna oblast, <b>NO:</b>		Elektronika
Naučna disciplina, <b>ND:</b>		Embedded Sistemi
Predmetna odrednica/Ključne reči, <b>PO:</b>		FPGA, Hardverski akcelerator, Detekcija objekata, Obrada slike, Viola Jones
<b>UDK</b>		
Čuva se, <b>ČU:</b>		Biblioteka Fakulteta Tehničkih Nauka 21000 Novi Sad, Trg Dositeja Obradovića 6
Važna napomena, <b>VN:</b>		Nema
Izvod, <b>IZ:</b>		U ovom diplomskom radu projektovana je arhitektura hardverskog akceleratora Viola-Jones algoritma za detekciju objekata na slici. Akcelerator je implementiran na FPGA čipu.
Datum prihvatanja teme, <b>DP:</b>		01.09.2019.
Datum odbrane, <b>DO:</b>		23.9.2019.
Članovi komisije, <b>KO:</b>	Predsednik:	dr. Vuk Vranković, docent
	Član:	
	Član, mentor	
		<b>Potpis mentora</b>

Obrazac Q2.HA.04-05 - Izdanje 1



UNIVERSITY OF NOVI SAD • FACULTY OF TECHNICAL SCIENCES  
21000 Novi Sad, Trg Dositeja Obradovića 6

## KEY WORDS DOCUMENTATION

Accession number, <b>ANO:</b>		
Identification number, <b>INO:</b>		
Document type, <b>DT:</b>	Monographic publication	
Type of record, <b>T3:</b>	Textual material, printed	
Contents code, <b>CC:</b>	Graduate thesis	
Author, <b>AU:</b>	Risto Pejašinić	
Mentor, <b>MN:</b>		
Title, <b>TI:</b>	Hardware implementation of Viola-Jones algorithm	
Language of text, <b>LT:</b>	Serbian	
Language of abstract, <b>LA:</b>	Serbian	
Country of publication, <b>CP:</b>	Serbia	
Locality of publication, <b>LP:</b>	Vojvodina	
Publication year, <b>PY:</b>	2019	
Publisher, <b>PB:</b>	Author's reprint	
Publication place, <b>PP:</b>	21000 Novi Sad, Trg Dositeja Obradovića 6	
Physical description, <b>PD:</b> (chapters/ pages/ ref. / tables/ pictures/ graphs/ appendixes)	(8/63/23/6/38/0/0)	
Scientific field, <b>SF:</b>	Electrical engineering	
Scientific discipline, <b>SD:</b>	Embedded Systems	
Subject/ Key words, <b>S/KW:</b>	FPGA, Hardware accelerator, Object Detection, Image Processing, Viola Jones	
<b>UC</b>		
Holding data, <b>HD:</b>	Library of Faculty of Technical Sciences 21000 Novi Sad, Trg Dositeja Obradovića 6	
Note, <b>N:</b>	None	
Abstract, <b>AB:</b>	In this bachelor thesis architecture of digital hardware accelerator for Viola-Jones object detection algorithm is designed. <u>Accelerator is implemented on FPGA.</u>	
Accepted by the Scientific Board on, <b>ASB:</b>	01.09. 2019.	
Defended on, <b>DE:</b>	23.9.2019.	
Defended board, <b>DB:</b>	President:	Ph. D
	Member:	Ph. D assistant
	Member, Mentor	Ph.D Vuk Vranković, assistant professor
		<b>Mentor's signature</b>

Obrazac Q2.HA.04-05 - Izdanje 1

# Sadržaj

<b>I</b>	<b>Viola-Jones algoritam</b>	<b>10</b>
<b>1</b>	<b>Uvod</b>	<b>10</b>
1.1	Integralna slika . . . . .	10
1.2	AdaBoost i HAAR obeležja . . . . .	12
1.2.1	HAAR obeležja . . . . .	12
1.2.2	AdaBoost . . . . .	13
1.3	Kaskadni klasifikator . . . . .	14
1.4	Skaliranje slike . . . . .	15
1.5	Osetljivost na osvetljaj . . . . .	17
1.6	Osetljivost na rotaciju objekta . . . . .	17
<b>2</b>	<b>OpenCV modeli</b>	<b>18</b>
2.1	OpenCV model za frontalna lica . . . . .	18
<b>II</b>	<b>Hardverska implementacija</b>	<b>19</b>
<b>3</b>	<b>Sažetak</b>	<b>19</b>
<b>4</b>	<b>Specifikacije za izvršavanje</b>	<b>20</b>
<b>5</b>	<b>Arhitektura hardvera</b>	<b>22</b>
5.1	Uvod . . . . .	22
5.2	Interfejsi IP jezgra . . . . .	22
5.3	Modul IMG RAM . . . . .	23
5.4	Modul rd_addrngen . . . . .	24
5.4.1	Scale_counter . . . . .	24
5.4.2	Boundaries . . . . .	24
5.4.3	Scale_ratio . . . . .	25
5.4.4	Hopper i Sweeper . . . . .	25
5.4.5	Skaliranje adrese . . . . .	27
5.4.6	Modul addr_trans . . . . .	28
5.5	Modul ii_gen i sii_gen . . . . .	28
5.5.1	Odabir algoritma . . . . .	28
5.5.2	Sekvencijalna implementacija generatora integralne slike . . . . .	28
5.5.3	Generator kvadratne integralne slike . . . . .	29
5.6	Modul frame_buffer . . . . .	30
5.7	Modul stddev . . . . .	31
5.8	Modul features_mem . . . . .	32
5.9	Modul classifier . . . . .	36
5.10	Interfejsi . . . . .	39

<b>6</b>	<b>PyGears metodologija</b>	<b>40</b>
6.1	Uvod . . . . .	40
6.2	Poređenje sa RTL metodologijom . . . . .	40
6.3	Jezici za opis hardvera . . . . .	41
6.4	Gears metodologija . . . . .	41
6.4.1	DTI interfejs . . . . .	41
6.5	Tipovi podataka . . . . .	43
6.5.1	UInt . . . . .	43
6.5.2	Int . . . . .	43
6.5.3	Tuple . . . . .	43
6.5.4	Array . . . . .	43
6.5.5	Float . . . . .	43
6.5.6	Fixp . . . . .	43
6.5.7	Queue . . . . .	43
6.5.8	Union . . . . .	45
6.5.9	Unit . . . . .	45
6.6	Čistoća Gear-ova . . . . .	45
6.7	Definicija Gear komponenti . . . . .	46
6.7.1	Gear implementiran pomoću SystemVerilog-a . . . . .	46
6.7.2	Gear implementiram kompozicijom . . . . .	46
6.7.3	Gear implementiram Python to HDL kompajlerom . . . . .	47
<b>7</b>	<b>Integracija IP jezgra sa Zynq sistemom</b>	<b>48</b>
7.1	Zynq . . . . .	48
7.2	Predloženi blok dijagram sistema . . . . .	49
7.3	Implementirani sistem na Zynq SoC . . . . .	51
7.4	Rezultati implementacije sistema . . . . .	53
7.4.1	Sinteza i implementacija hardvera . . . . .	53
7.4.2	Analiza potrošnje hardverskih resursa . . . . .	54
7.4.3	Analiza vremenskog izveštaja i Timing Closure . . . . .	55
7.5	Softver procesora . . . . .	59
<b>8</b>	<b>Performanse i optimizacije</b>	<b>62</b>
8.1	Performanse . . . . .	62
8.2	Optimizacije . . . . .	63
8.2.1	Refaktorisanje generisanja integralne slike . . . . .	63
8.2.2	Generisanje integralne slike tokom rada klasifikatora . . . . .	63
8.2.3	Paralelno računanje prve etape . . . . .	64
8.2.4	Paralelni klasifikatori . . . . .	64
8.2.5	Računanje više obeležja u paraleli . . . . .	64
8.2.6	Računanje skaliranih slika u paraleli . . . . .	65
<b>9</b>	<b>Zaključak</b>	<b>66</b>

## Slike

1.1	Primer integralne slike . . . . .	11
1.2	Primer računanja površine pravougaonika [1] . . . . .	11
1.3	HAAR obeležja [2] . . . . .	12
1.4	Primer obeležja za detekciju lica [3] . . . . .	13
1.5	Kaskadni klasifikator [2] . . . . .	14
1.6	Procenat izvršavanja etapa na svim regionima slike . . . . .	15
1.7	Piramida slike[4] . . . . .	16
1.8	Različite veličine lica . . . . .	16
1.9	Previše osvetljaja[5] . . . . .	17
1.10	Premalo osvetljaja[5] . . . . .	17
1.11	Originalna[5] . . . . .	17
1.12	Rotirana[5] . . . . .	17
4.1	Veza Python modela sa XML modelom i C specifikacijom . . . . .	20
5.1	Arhitektura hardvera kaskadnog klasifikatora . . . . .	22
5.2	Blok dijagram <b>rd_addrngen</b> modula . . . . .	24
5.3	Način rada <b>hopper</b> i <b>sweeper</b> komponenti. . . . .	26
5.4	Prelazak <b>hopper</b> -a u novi red. . . . .	27
5.5	Posledica skaliranja adrese. . . . .	27
5.6	Blok dijagram <b>ii_gen</b> modula . . . . .	29
5.7	Blok dijagram <b>ii_gen</b> modula . . . . .	30
5.8	Blok dijagram <b>frame_buffer</b> -a realizovanog sa 3 jedno-portne RAM memorije. . . . .	30
5.9	Blok dijagram <b>stddev</b> modula. . . . .	32
5.10	Reprezentacija pravougaonika u obeležju. . . . .	33
5.11	Blok dijagram <b>features_mem</b> modula. . . . .	35
5.12	Blok dijagram <b>classifier</b> modula. . . . .	36
6.1	DTI interfejs[6] . . . . .	41
6.2	DTI primer transakcije . . . . .	42
6.3	Primer transakcije tipa . . . . .	44
6.4	Matrica veličine 3x3 . . . . .	44
6.5	Transakcija matrice 6.4 . . . . .	44
6.6	Unit tip . . . . .	45
6.7	Serialize gear primer . . . . .	47
7.1	Okviran blok dijagram sistema. . . . .	49
7.2	Zturn ploča[7] . . . . .	51
7.3	Kašnjenje kombinacione logike . . . . .	55
7.4	Vremenski izveštaj pre skraćivanja kombinacionih putanja . . . . .	56
7.5	Vremenski izveštaj posle skraćivanja kombinacionih putanja . . . . .	58
7.6	Softverski stek. . . . .	59
8.1	Performanse <b>frame_buffer</b> modula . . . . .	63

## Tabele

5.1	Struktura zapakovane memorije rect_ROM0 za model(2.1)	34
5.2	Struktura raspakovane memorije rect_ROM0 za model(2.1)	34
5.3	Veličina memorije features_mem za model 2.1	35
7.1	Hardverski resursi nakon sinteze PyGears implementacije	54
7.2	Hardverski resursi nakon sinteze SystemVerilog implementacije	54
7.3	Registarska mapa dm_cmd komponente	60
8.1	Poređenje performansi	62

## Kodni Segmenti

5.1	Primer <b>hopper</b> -a u <b>C</b> -u	25
5.2	Primer <b>hopper</b> -a i <b>sweeper</b> -a u <b>C</b> -u	25
5.3	Primer računanja standardne devijacije u <b>C</b> -u	31
5.4	Weighted_sum u <b>C</b> -u	37
5.5	Leaf_val u <b>C</b> -u	38
5.6	Stage_res u <b>C</b> -u	39
6.1	Primer definisanja Gear komponente	46
6.2	Primer kompajliranog serialize gear-a	47
7.1	PyGears implementacija stddev komponente	57



## Mali rečník

Akronim	Opis
AdaBoost	= Adaptive Boosting
ASIC	= Application Specific Integrated Circuit
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
HDMI	= High-Definition Multimedia Interface
LED	= Light Emitting Diode
MM	= Memory Mapped
PL	= Programmable Logic
PS	= Processing System
RAM	= Random Access Memory
ROM	= Read Only Memory
RTL	= Register Transfer Methodology
SO	= Shared Object
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 [3].

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 [1]

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.

$$\sum_{(x,y) \in ABCD} i(x,y) = ii(D) + ii(A) - ii(B) - ii(C) [8] \quad (1.2)$$

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.

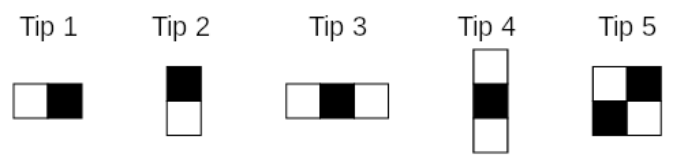
Integralnu sliku je moguće računati u paraleli ili sekvencijalno. Izbor algoritma za računanje integralne slike značajno utiče na performanse i potrebne hardverske resurse konačne implementacije.

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 [2]

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 [3]

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 minimalan broj obeležja za pouzdanu 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%.

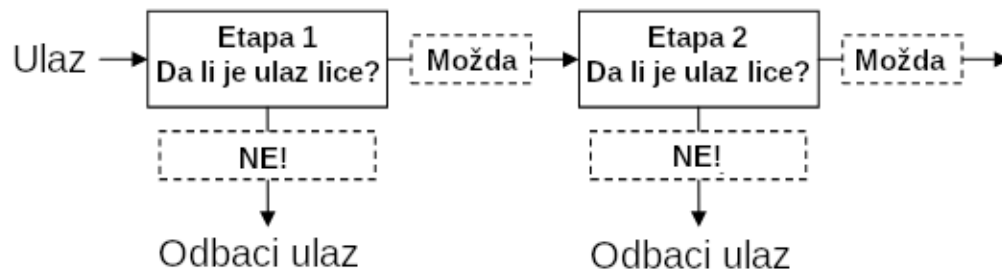
Kaskadiranjem ovako dobijenih *weak learner*-a može se dobiti *strong classifier*

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.



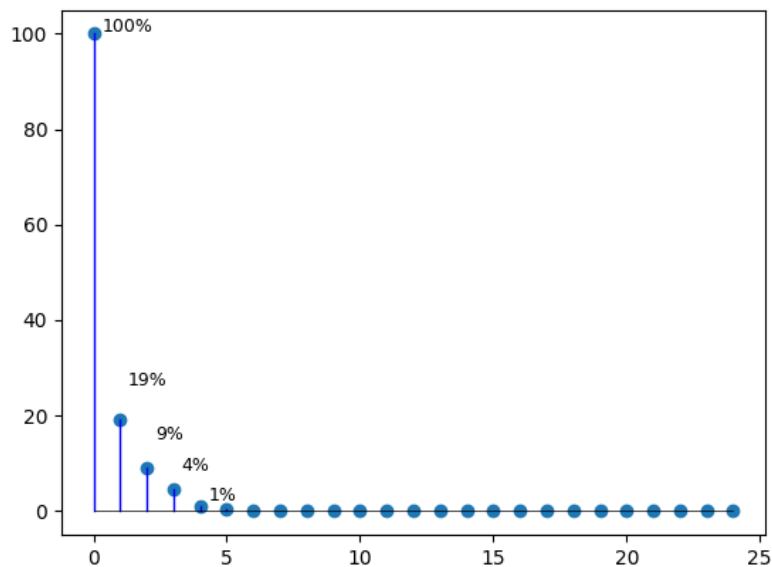
Slika 1.5: Kaskadni klasifikator [2]

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...

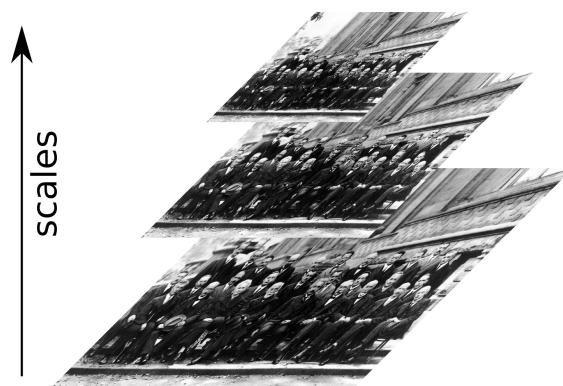
Vidimo da je posle pete etape procenat izvršavanja manji od 1%.

## 1.4 Skaliranje slike

Objekti na slikama mogu biti bliži ili dalji kameri odnosno mogu biti različitih dimenzija, potrebno je obezbediti da se objekti 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.

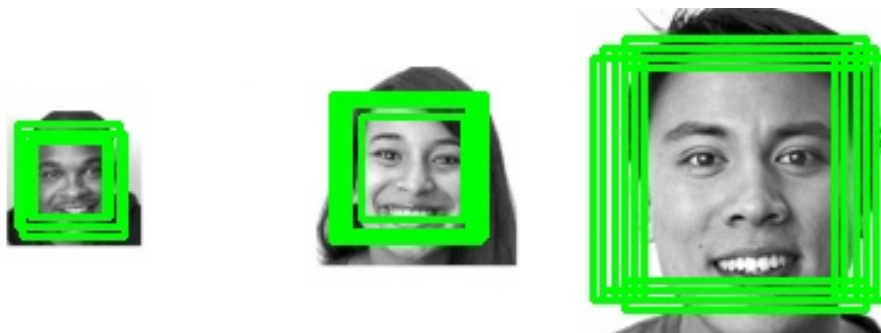


Slika 1.7: Piramida slike[4]

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 zadovoljavajući 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]

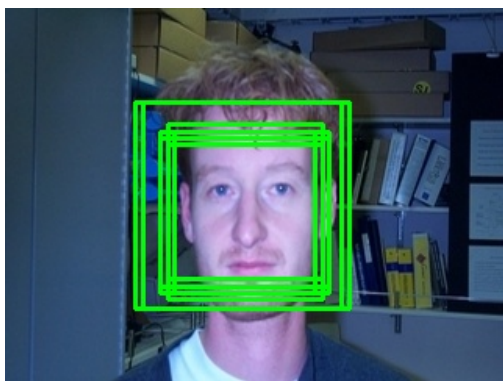
Na slikama(1.9, 1.10) su prikazana dva lica koja zbog nepovoljnog osvetljenja nisu detektovane od strane klasifikatora. Slika(1.9) nije detektovana zbog pristustva prejakog osvetljenja lica, dok slika(1.10) nije detektovana zbog slabog osvetljenja.

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<sup>1</sup>. [9]

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`<sup>2</sup>. 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.

---

<sup>1</sup><https://github.com/opencv/opencv/tree/master/data/haarcascades>

<sup>2</sup>[https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade\\_frontalface\\_default.xml](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<sup>3</sup> metodologiji.
- Poređenje dve metodologije i analiza prednosti i mana obe metodologije.
- Implementacija projektovanog IP jezgra na MYIR Z-Turn Board<sup>4</sup> ploči sa Zynq 7020 SoC.
- Pisanje Linux Kernel drajvera i korisničke aplikacije za korišćenje jezgra za detekciju lica na Xilinx Zynq platformi.

---

<sup>3</sup><https://github.com/bogdanvuk/pygears>

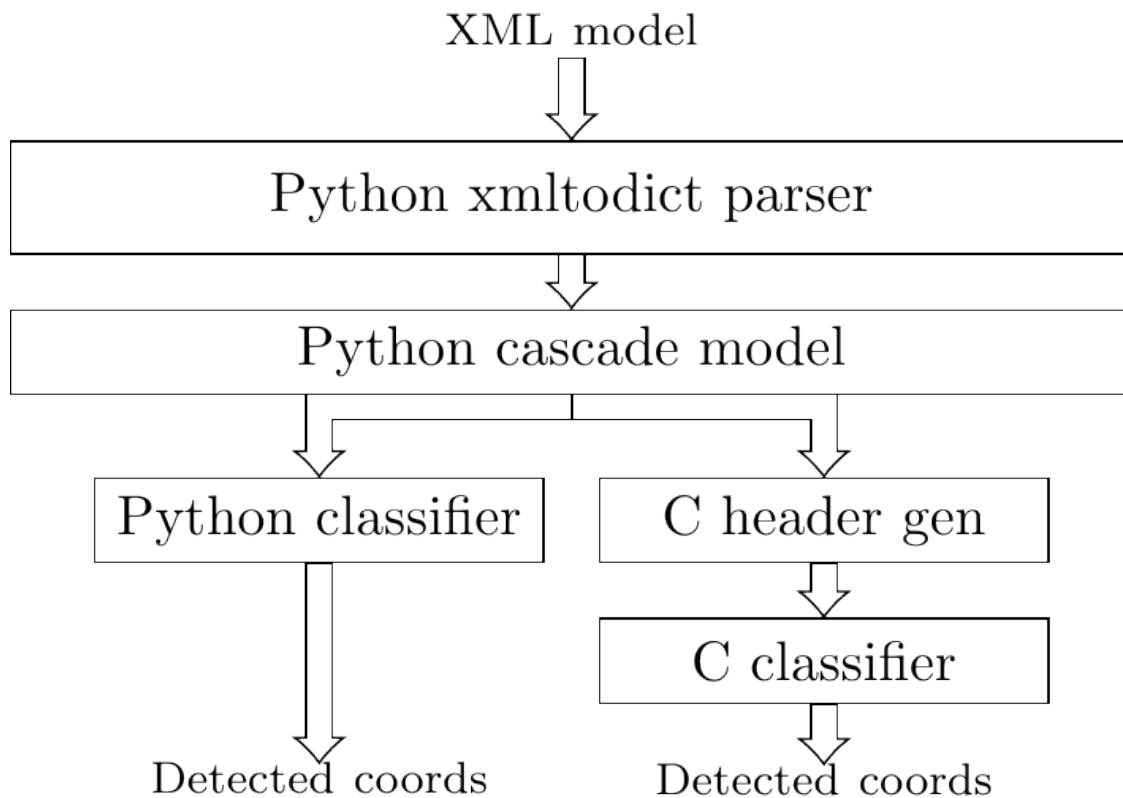
<sup>4</sup><http://www.myirtech.com/list.asp?id=502>

## 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 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 xmltodict<sup>5</sup>.

*Xmltodict* parsira XML fajl i skladišti ga u Python dictionary.

<sup>5</sup><https://pypi.org/project/xmltodict/>

Implementirane su klase *CascadeClass*, *StageClass*, *FeatureClass* i *RectClass* <sup>6</sup> 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.

---

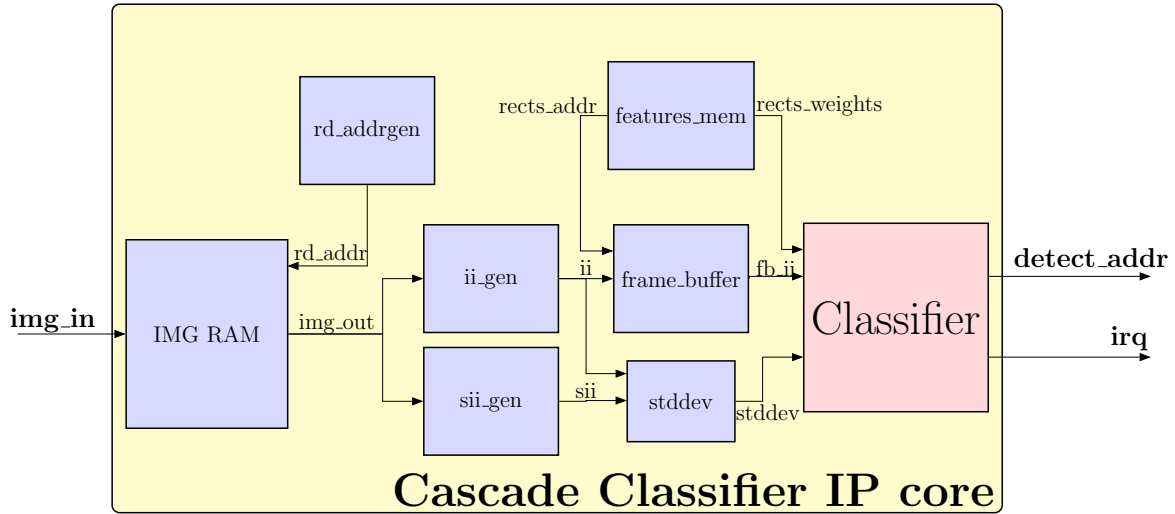
<sup>6</sup>`cascade_classifier/python_model/cascade.py`

## 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.

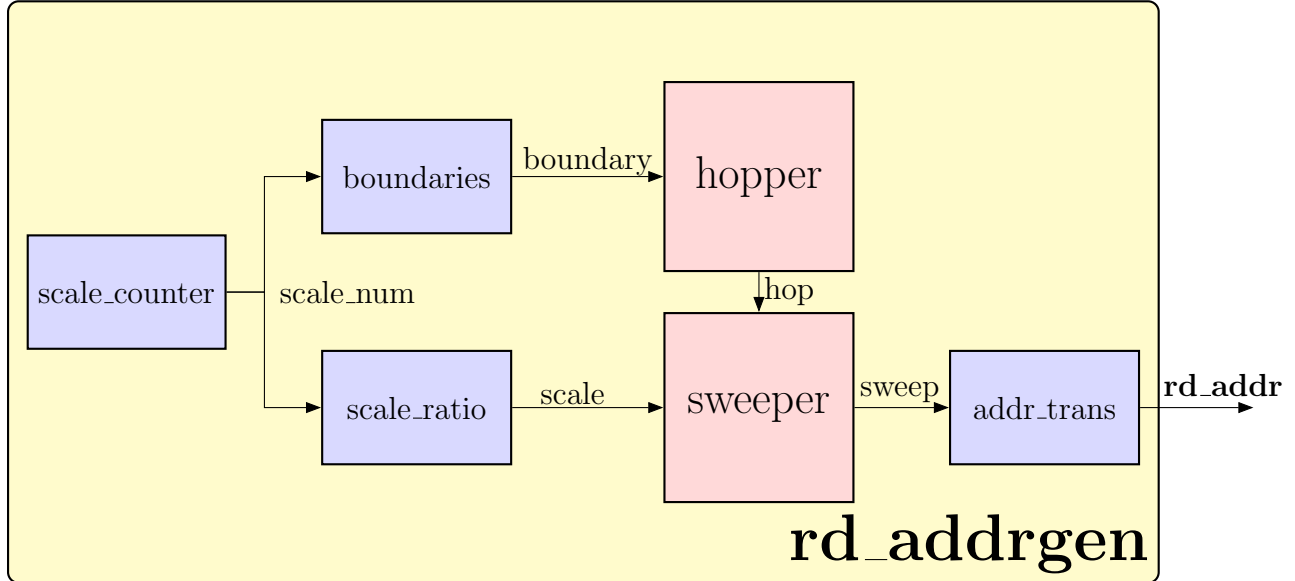
Veličina memorije je  $width * height * 8$  bit. Za dimenziju slike 240x320 veličina memorije je 614400 bita.

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_addrngen`

Modul `rd_addrngen` 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_addrngen` 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 podatak. 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 ugnježđena for petlja gde iteratori petlje predstavljaju koordinate **X** i **Y**.

Prvo se iterira po **X** koordinati pa zatim po **Y**.

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

---

```

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  }
```

---

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 ugnježđene for petlje. Ponovo se prvo iterira po X koordinati pa onda po Y.

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

---

```

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   }

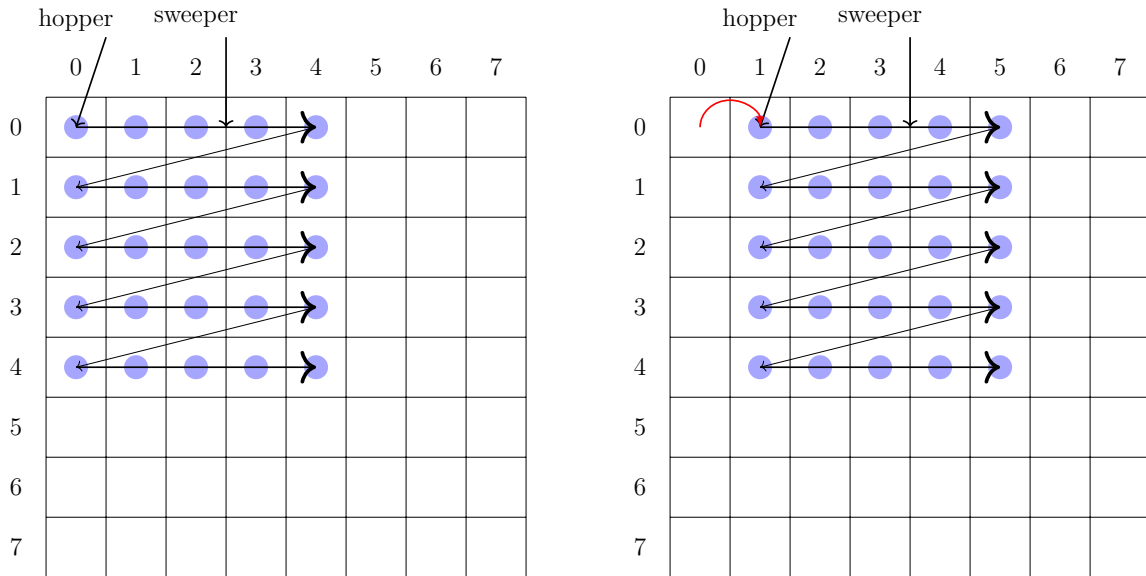
```

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 predstavljani su pikseli koje **sweeper** prebriše za jedan položaj **hopper**-a.

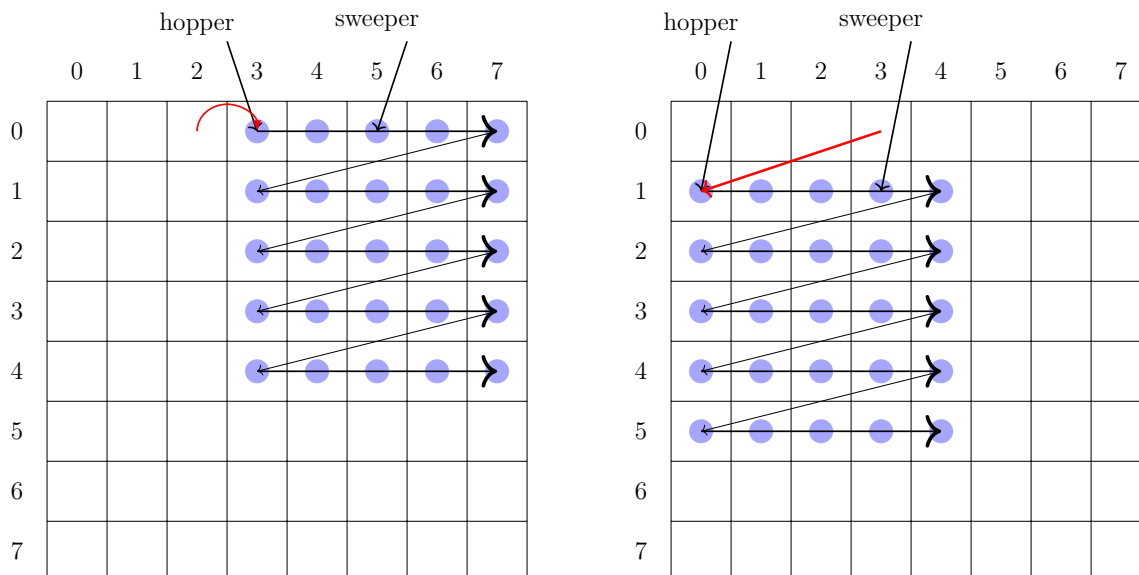
Gornji levi kružić predstavlja koordinatu trenutnog položaja **hopper**-a. Crevnom strelicom je obeleženo iteriranje **hopper**-a kroz sliku.



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.

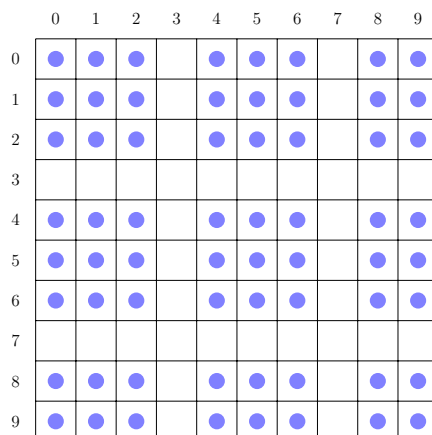
Slika 5.4: Prelazak **hopper**-a u novi red.

#### 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 uštedeno 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 potrebno je linearizovati adresu dobijenu iz prethodnih modula.

Odnosno adresu u formatu  $(y, x)$  treba pretvoriti u linearnu pomoću sledeće formule:

$$lin\_addr = (y * img\_width) + x \quad (5.1)$$

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

Možemo izabrati sekvencijalni ili paralelni algoritam.

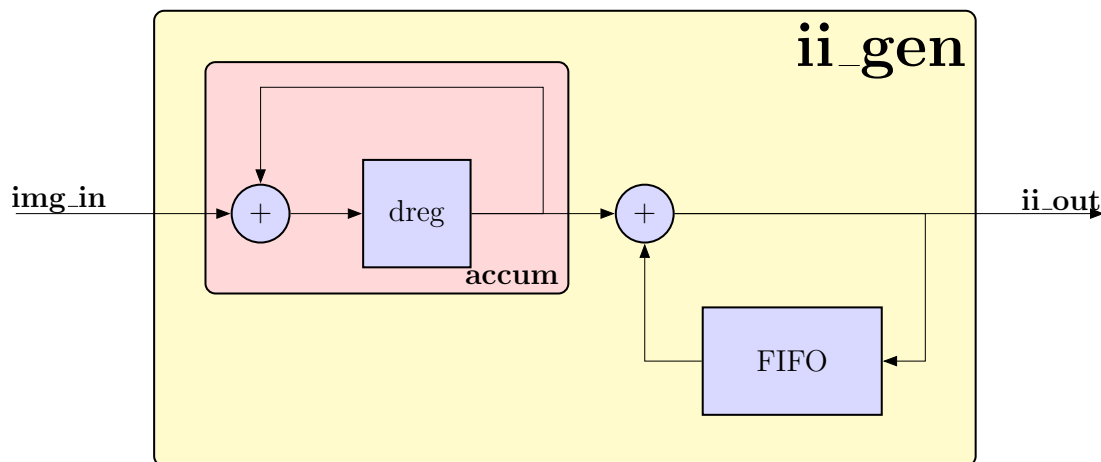
Ukoliko bi se odabrao paralelni algoritam koji može da računa više piksela u paraleli povećanje resursa bi se drastično odrazilo na `img_ram` i `frame_buffer` memorije. Ali bi dobili bolje performanse sistema.

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 je manje pristupa memoriji preko ulaznog i izlaznog interfejsa, potrebno je manje funkcionalnih jedinica i unutrašnje memorije. 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 ovog modula je povezan sa 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 prethodnog 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.

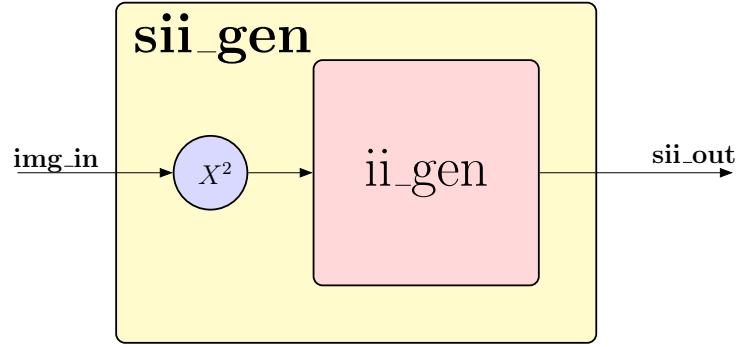
U ovom slučaju standardan FIFO bafer je modifikovan na sledeći način:

- 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.

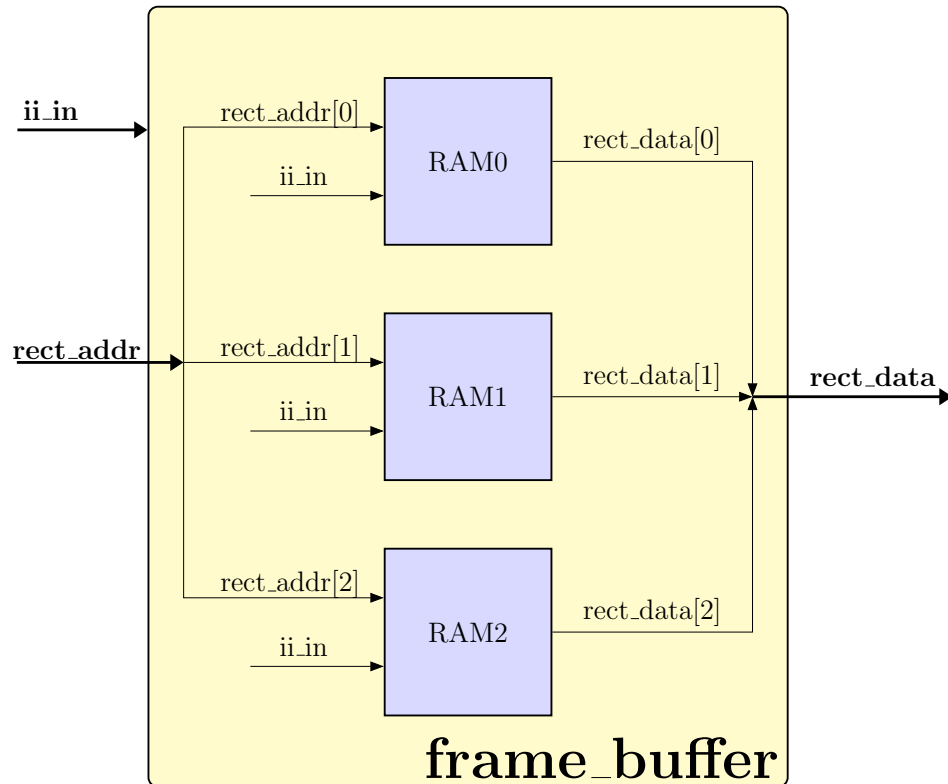
Generisanje kvadratne integralne slike je jednostavno uz korišćenje generatora integralne slike. Potrebno je kvadrirati ulazne piksele i dovesti ih na generator integralne slike kao na slici(5.7).



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 inkrementuje za jedan nakon svakog primljenog podatka kao i kod IMG RAM modula. Brojač nije prikazan na blok dijagramu iznad.

Potrebna veličina jedne RAM memorije je data sledećim vezama:

$$size[bit] = frame\_width * frame\_height * w\_ii \quad (5.2)$$

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

Gde su `frame_width` i `frame_height` širina i visina obeležja modela, `w_ii` je širina magistrale integralne slike, `w_img` je širina ulazne 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. U ovom slučaju bi nam trebala RAM memorija sa jednim portom za upis i tri porta za čitanje. 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 [10]. 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.

Primer 5.3: Primer računanja standardne devijacije u C-u

---

```

1 long calcStddev(long sii[FRAME_HEIGHT][FRAME_WIDTH],
2                 long ii[FRAME_HEIGHT][FRAME_WIDTH]){
3     long mean, stddev;
4
5     mean = ii[0][0] + ii[FRAME_HEIGHT-1][FRAME_WIDTH-1] -
6           ↪ ii[0][FRAME_WIDTH-1] - ii[FRAME_HEIGHT-1][0];
7
8     stddev = sii[0][0] + sii[FRAME_HEIGHT-1][FRAME_WIDTH-1] -
9           ↪ sii[0][FRAME_WIDTH-1] - sii[FRAME_HEIGHT-1][0];
10
11     stddev = (stddev * (FRAME_WIDTH-1)*(FRAME_HEIGHT-1));
12     stddev = stddev - (mean*mean);
13     stddev = getSqrt(stddev);
14     return stddev;
15 }
```

---

Za računanje standardne devijacije potrebni su pikseli ivice prozora integralne i kvadratne integralne slike, što se vidi u liniji 5 i 7 primera(5.3).

Sabiranjem gornjeg levog i donjeg desnog piksela zatim oduzimanje gornjeg desnog i donjeg levog piksela 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.

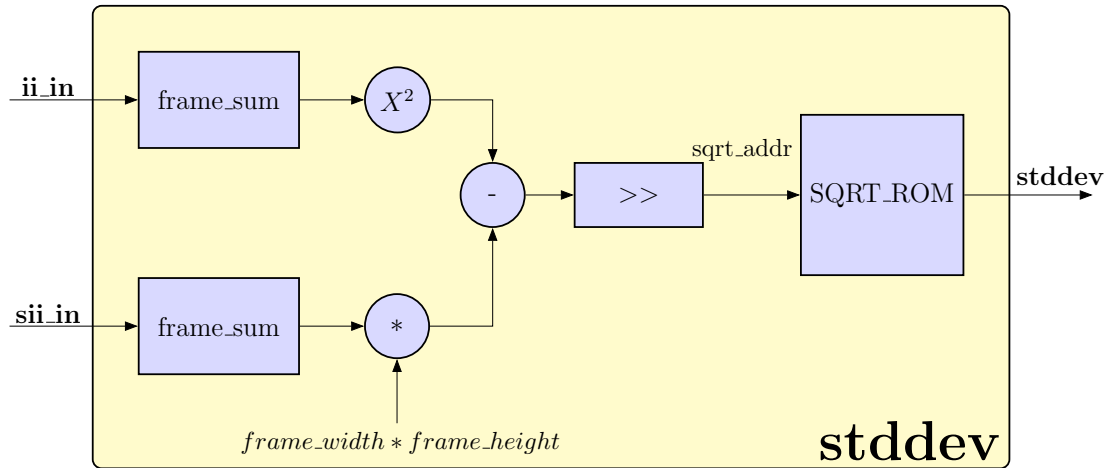
Konačno potrebno je odrediti kvadratni koren u liniji 11.

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 da se koristi lookup tabela.

Za unapred definisani opseg vrednosti operanda za korenovanje softverski su izračunate vrednosti kvadratnog korena i smeštene u ROM memoriju.

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 obavlja ju 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.

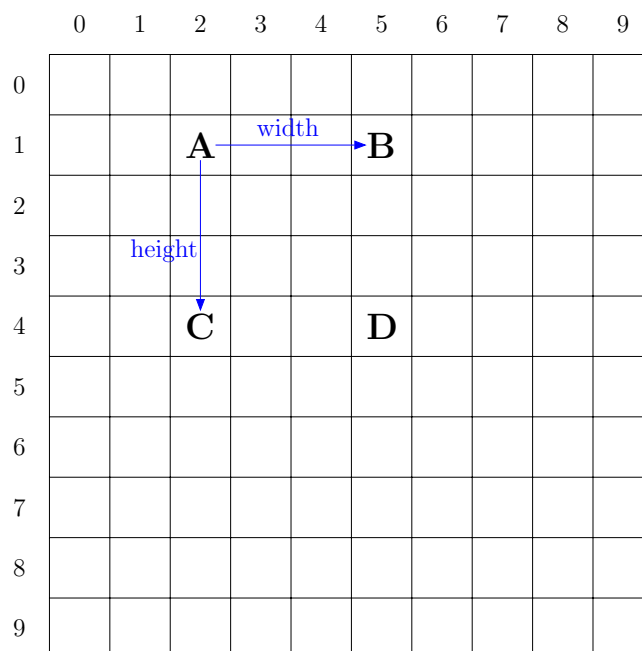


Svaki pravougaonik je definisan sa četiri koordinate (A, B, C, D) i sa težinom njegove površine.

Zbog čestog i ponovljenog pristupa ovim obeležjima potrebno ih je skladištiti u lokalnoj memoriji. U ovom slučaju se radi o ROM memoriji, koja sadrži obeležja odabranog OpenCV modela.

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.

Koordinate smeštene u ovoj memoriji se koriste za adresiranje podataka iz frame\_buffer memorije opisane ranije. 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:

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

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

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

Tabela 5.1: Struktura zapakovane memorije rect\_ROM0 za model(2.1)

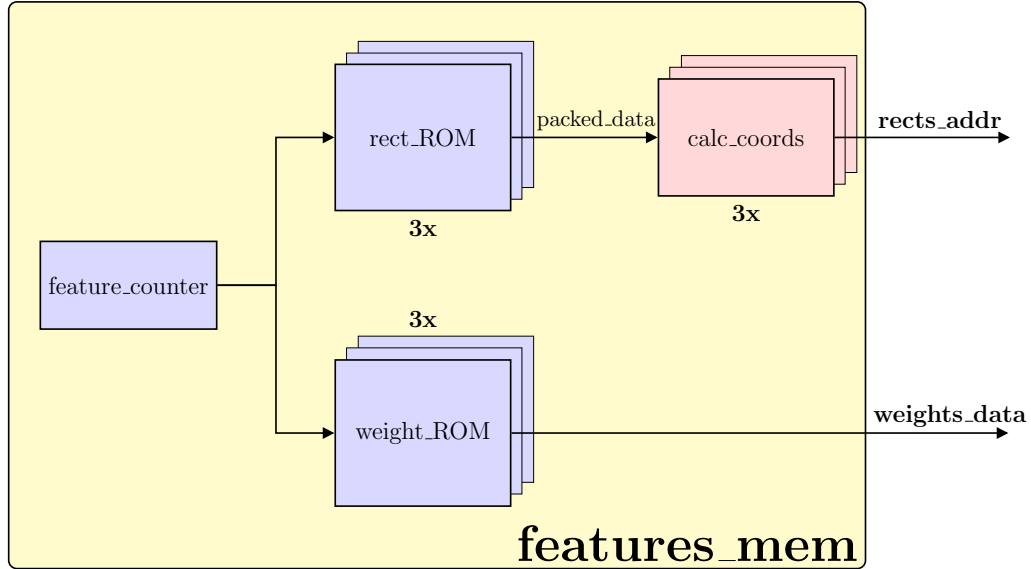
N	Packed value(20 bit)
0	0x1A989
1	0x1A987
2	0x39249
3	0x72926
.	.
.	.
.	.
feature num - 1	0x088d6

Tabela 5.2: Struktura raspakovane 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

Na tabeli(5.1) je prikazana struktura 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. Površine pravougaonika obeležja je potrebno pomnožiti sa ovim težinama.



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 jednačinama(5.4, 5.5, 5.6).

Konačno može se izračunati potrebna količina memorije u ovom modulu u slučaju modela opisanog u sekciji(2.1).

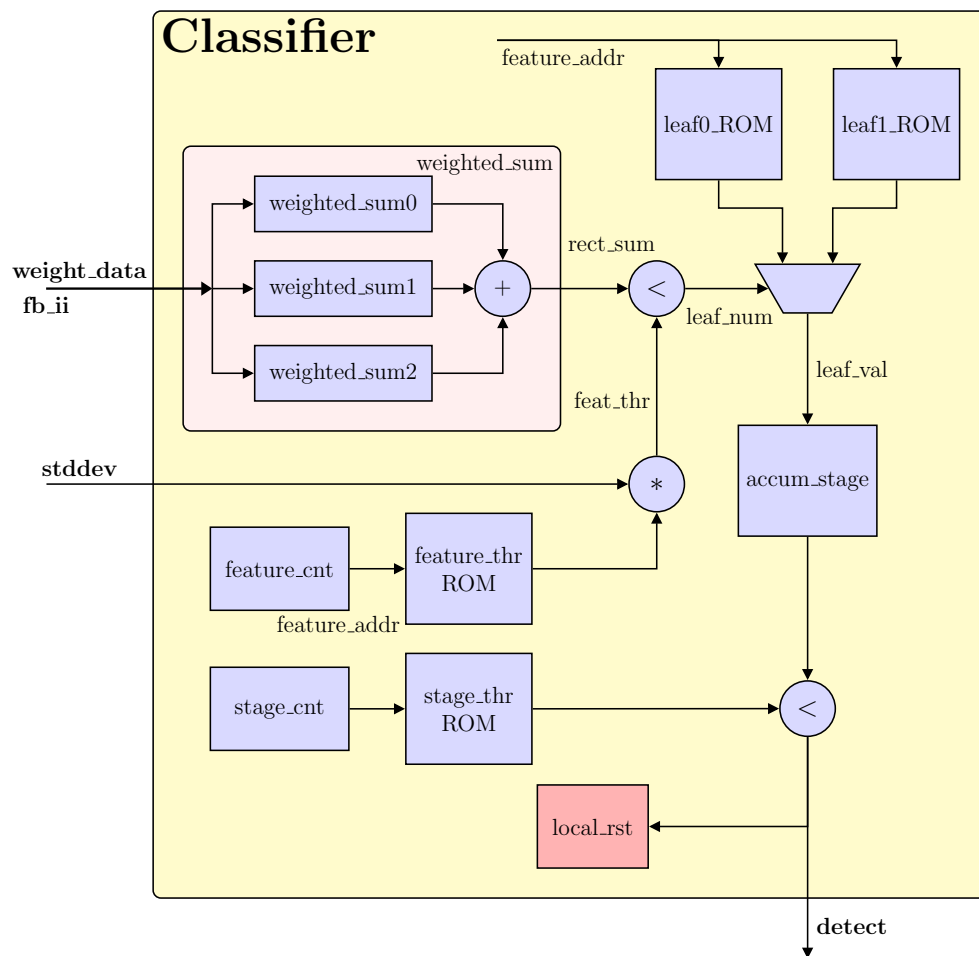
Tabela 5.3: Veličina memorije features\_mem za model 2.1

<b>feature_num</b>	2913
<b>rect_num</b>	3
<b>w_weight</b>	3 bits
<b>w_rect</b>	20 bits
<b>TOTAL</b>	524,340 bits

## 5.9 Modul classifier

**Classifier** modul vrši klasifikaciju prozora. Odnosno signalizira da li se na posmatranom prozoru nalazi traženi objekat.

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.

Primer 5.4: Weighted\_sum u C-u

---

```

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 približno prikazuje algoritam rada weighted\_sum komponente sa slike(5.12). Memorije weights i rects se nalaze u features\_mem komponenti u hardverskoj implementaciji i objašnjene su u sekciji(5.8). Memorija ii predstavlja frame\_buffer modul u hardverskoj arhitekturi.

Funkcija weighted\_sum\_i() računa površinu pravougaonika na slici. Pritom koristi integralnu sliku radi bržeg proračuna. 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.

Primer 5.5: Leaf\_val u C-u

---

```
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(5.6) prikazuje algoritam računanja leaf\_val vrednosti. Memorijska feature\_thresholds se nalazi na slici(5.12) pod nazivom feature\_thr ROM. Memorijske 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.

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 možda nalazi objekat i potrebno je izvršiti sledeću etapu.

Primer 5.6: Stage\_res u C-u

---

```
1 int stage_res(int leaf_val,
2               int feature_start,
3               int feature_end){
4     int64_t sum = 0;
5
6     for(int feature_num = feature_start;
7         feature_num < feature_end;
8         feature_num++){
9         sum += leaf_val;
10    }
11
12    if(sum < stageThresholds[stage_num]) return -1;
13    else return 1;
14 }
```

---

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. Nakon čega će se preći na analizu sledećeg prozora.

Ukoliko akumulirane vrednosti obeležja svih etapa imaju veću vrednost od praga etapa, može se zaključiti da se posmatranom prozoru zaista nalazi lice. U tom slučaju potrebno je proslediti adresu hopper-a na izlazni interfejs IP jezgra.

## 5.10 Interfejsi

Komunikacija svih unutrašnjih modula i komunikacija IP jezgra sa okolinom realizovana je pomoću **Data Transfer Interface (DTI)**[6] interfejsa.

Korišćenjem ovog interfejsa dobijena je robusnija komunikacija između modula zahvaljujući *handshaking*-u.

Dodatna prednost ovog interfejsa je kompatibilnost sa **AXI-Stream**[11], **Avalon-ST**[12] i sličnim streaming interfejsima. Zahvaljujući tome integracija IP jezgra koje koristi DTI spoljne interfejse sa AXI-Stream IP jezgrima je trivijalna.

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

## 6 PyGears metodologija

### 6.1 Uvod

PyGears[13] pozajmljuje koncepte iz funkcionalnog programiranja i primenjuje ih na dizajn 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[6] predstavlja alternativu Register Transfer Methodology (RTL)[14] metodologiji za opis hardvera. RTL metodologija pruža standardan način translacije sekvencijalnog algoritma u hardverski 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[15] Controlpath diriguje podacima u Datapath delu i sačinjen je od mašine stanja.

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.

Gears metodologija je posebno pogodna u sistemima koji su Dataflow orijentisani. U Dataflow sistemima može se razmišljati o tokovima podataka i njihovim transformacijama. PyGears omogućava pisanje interfejsa proizvoljnog tipa. Tako da se mogu napisati Gear-ovi koji svoju funkcionalnost prilagođavaju tipom povezanog ulaznog interfejsa. Na ovaj način napisani Gear-ovi su apstraktni i laki za razumevanje.



### 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<sup>7</sup>. 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[17, 6], Scala [18, 19], Haskell[20] generišu konačan HDL kod u Verilog-u, SystemVerilog-u ili VHDL-u pogodnom za simulatore i alate za sintezu.

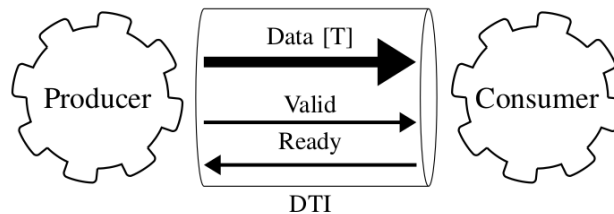
Dok se ovi jezici uglavnom baziraju na RTL metodologiji i uvode jezik višeg nivoa kao zamenu za standardan HDL. PyGears dodatno izdvaja i uvođenje nove metodologije zvane Gears.

### 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[6]

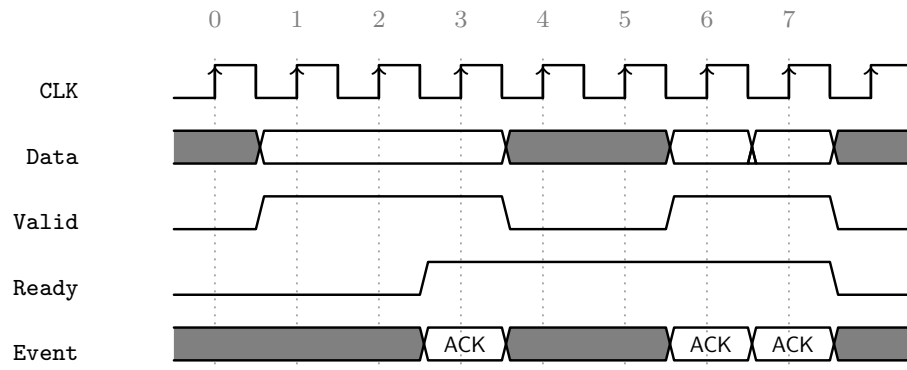
Jednostavan **DTI** interfejs se koristi za komunikaciju između gear-ova. Modul koji šalje podatke se naziva Producer, a modul koji prima podatke se naziva Consumer. DTI se sastoji od sledećih signala:

- **Data** linije za podatke, proizvolje širine i proizvoljnog tipa.

<sup>7</sup>Postoji inicijativa da se dokumentuju FPGA čipovi svih velikih proizvođača u projektu zvanom SymbiFlow[16].

- **Valid** jednobitan signal kojim upravlja Producer i signalizira da je podataka na Data liniji validan.
- **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 Consu-mera.
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 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.

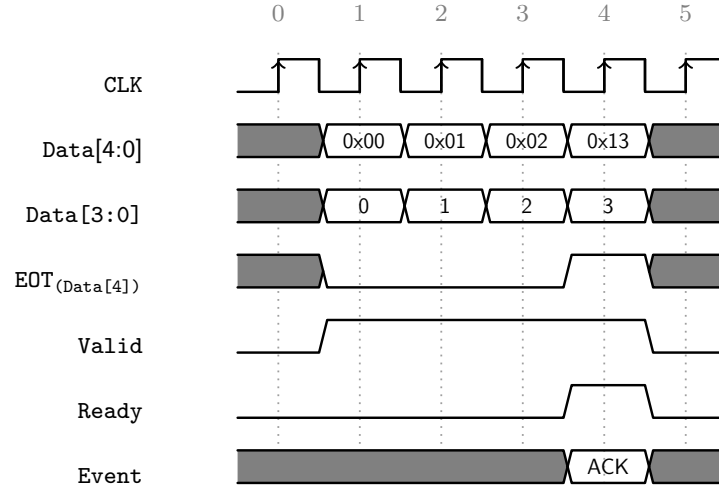
Format Fixed point tipa je **WI** širina celobrojnog dela i **W** ukupna širina magistrale, širina decimalnog dela je razlika ove dve širine.

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**.

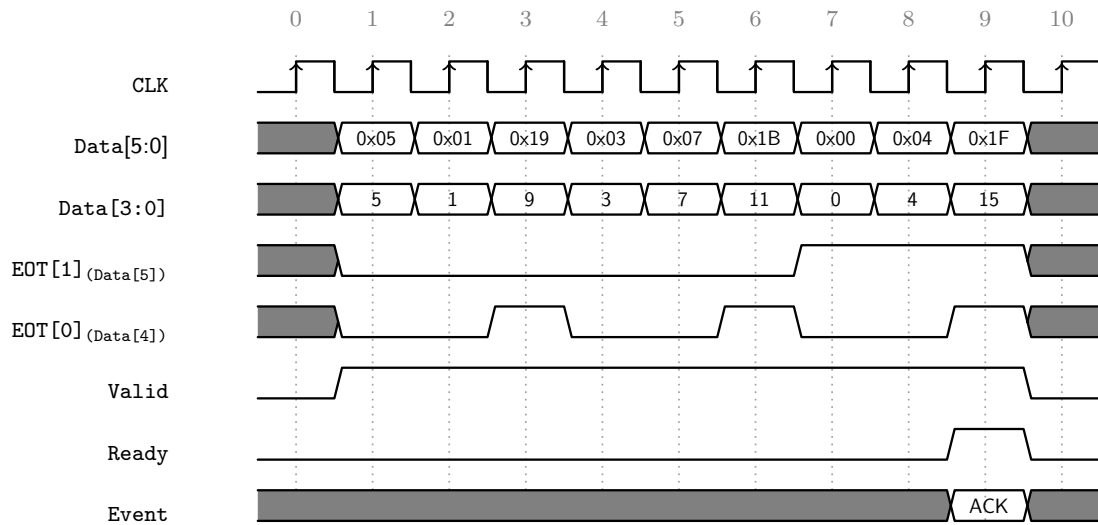


Slika 6.3: Primer transakcije tipa

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

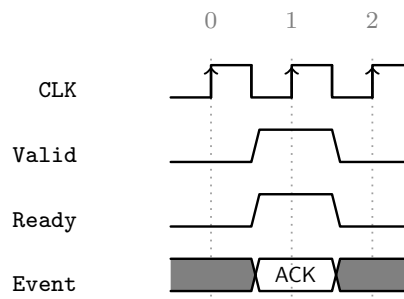
Na slici(6.4) u taktovima(3, 6, 9) niži bit EOT-a je na visokom nivou što predstavlja podatak u poslednjoj koloni matrice. U taktovima(7, 8, 9) viši bit EOT-a je na visokom nivou što označava podatke u poslednjoj liniji matrice. Konačno u taktu 9 oba EOT bita su na visokom nivou što označava poslednji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

Primer 6.1: Primer definisanja Gear komponente

---

```

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

```

---

Primer(7.1) 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.

Primer 6.2: Primer kompajliranog serialize gear-a

---

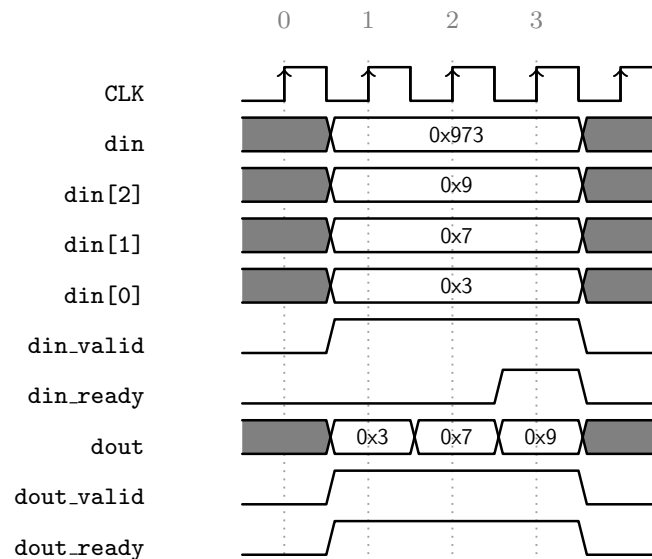
```

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]

```

---

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 interfejsom, tako da je integracija sa sistemima koji koriste druge interfejse poput Avalon-a trebala biti jednostavna uz adaptacije.

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) sistema 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.

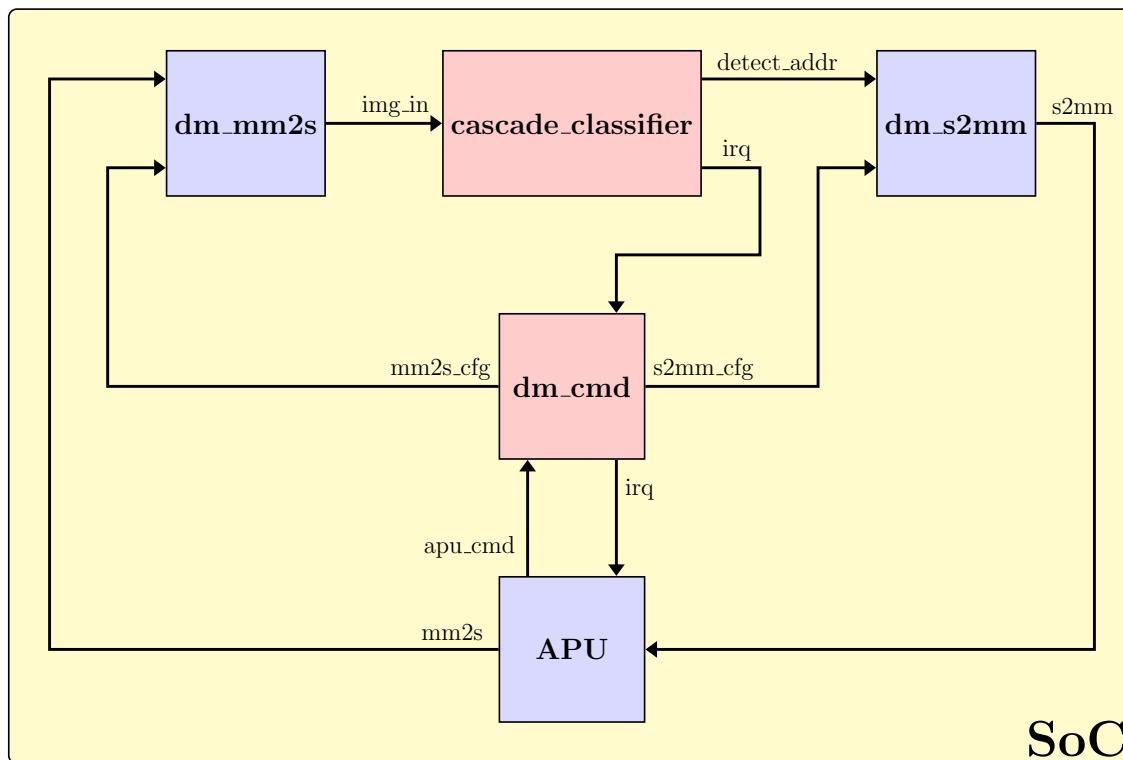
ARM procesor ima mogućnost povezivanja eksterne Double Data Rate (DDR) memorije preko internog DDR kontrolera. Preko ovog kontrolera pristup memoriji ima i PL sistem. Zynq SoC ima mogućnost pokretanja GNU/Linux operativnog sistema.



## 7.2 Predloženi blok dijagram sistema

U nastavku je prikazan okviran blok dijagram povezivanja projektovanog IP jezgra sa procesorskim sistemom.

Na slici su izostavljene interkonekcijske komponente, generatori reset i clk signala, DDR konekcije itd...



Slika 7.1: Okviran blok dijagram sistema.

Blokovi obojeni crvenom bojom su projektovani u okviru ovog rada. Blokovi obojeni plavom bojom su standardne komponente i mogu se naći u okviru softverskog alata od proizvođača korišćenog SoC-a.

Kako je već rečeno spoljni interfejsi **cascade\_classifier** IP jezgra su Streaming tipa. Streaming interfejsi nemaju informaciju o adresi, tako da sa njima nije moguće adresirati podatak iz DDR memorije preko DDR kontrolera.

Komponente **dm\_s2mm** i **dm\_mm2s** su Data Mover-i, koji pretvaraju Streaming interfejs u Memory Mapped (MM) interfejse. Kako Streaming interfejs prenosi samo podatak i nema informaciju o adresi, potrebno je proslediti adresu i broj podataka preko posebnog komandnog interfejsa, na ovoj slici to su **s2mm\_cfg** i **mm2s\_cfg**.

Glavno IP jezgro nema koristi od informacije sa koje adrese dolazi slika i na kojoj adresi se smeštaju rezultati, pa je zato za generisanje komande za Data Mover komponente zadužen

dm\_cmd.

Komponenta dm\_cmd od procesora dobija adresu bafera slike i rezultata iz eksterne DDR memorije i na osnovu toga generiše komande za Data Mover-e.

Adrese i broj podataka procesor šalje preko apu\_cmd interfejsa.

Nakon konfigurisanja Data Mover-a prvo će sa radom početi dm\_mm2s Data Mover, koji će poslati zahtev za sliku DDR kontroleru i pročitane podatke proslediti glavnom IP jezgru na obradu. Nakon popunjavanja IMG RAM memoriju unutar glavnog IP jezgra, jezgro počinje obradu slike.

Ukoliko dođe do detekcije objekta na nekoj koordinati, ta koordinata će biti poslata preko dm\_s2mm Data Mover-a u određenu lokaciju za smeštanje rezultata u okviru eksterne DDR memorije.

Nakon završetka obrade trenutne slike aktiviraće se irq signal koji označava interrupt za procesor. Glavno IP jezgro će držati aktivnim ovaj signal samo jedan takt, dok će se taj signal sačuvati u registru u okviru dm\_cmd jezgra i biće prosleđen na izlazni irq signal povezan sa procesorom.

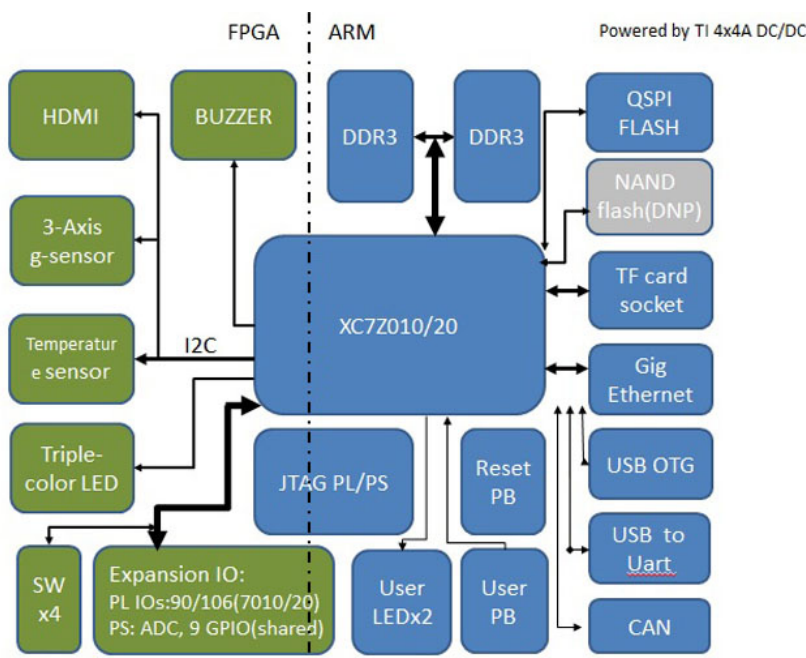
Nakon detekcije interapt signala od strane procesora, poslaće se zahtev za reset ovog registra preko apu\_cmd interfejsa. Sam interapt signal procesoru signalizira da može početi sa pripremanjem sledeće slike. Nakon smeštanja naredne slike u bafer, procesor ponovo šalje zahtev za obradu slike preko apu\_cmd interfejsa.

Procesor pored slanja komande IP jezgru treba da preuzme sliku sa kamere ili učitava iz fajla, zatim je konvertuje u grayscale reprezentaciju. Nakon prihvatanja rezultata detekcije procesor može koristiti dobijene koordinate za željeni zadatak.

Broj primena algoritama za detekciju objekata je ogroman i od korisnika zavisi dalja implementacija softvera. Rad IP jezgra u ovom radu je demonstriran iscrtavanjem pravougaonika na pozicijama detektovanih objekata i prikazivanjem slike sa iscrtanim pravougaonicima na HDMI monitor.

### 7.3 Implementirani sistem na Zynq SoC

U ovom radu sistem je implementiran na Zynq-7020 SoC. Korišćena je ZTurn[7] ploča firme MYiR. Ova ploča se sastoji od Zynq-7020 čipa, 1GB eksterne DDR3 memorije, High-Definition Multimedia Interface (HDMI) kontroler i konektor, tasteri i Light Emitting Diode (LED) za testiranje, g-senzor, temperaturni senzor, buzzer, flash memorija, SD card socket, Gig ethernet, JTAG, USB, CAN itd... Što se može videti na sledećoj slici.



Slika 7.2: Zturn ploča[7]

Od eksternih perifera u ovom projektu korišćeni su SD kartica, Ethernet, HDMI, USB i DDR memorija. Procesor pokreće Arch Linux ARM[21] operativni sistem, a komunikacija sa IP jezgrom je odrađena preko napisanog Linux Kernel Drivera i korisničke aplikacije.

Na sledećoj strani prikazan je blok dijagram Vivado integratora implementiranog sistema. Može se primetiti da pored komentarisanih komponenti za povezivanje procesora sa IP jezgrom, na blok dijagramu se nalazi i Video kontroler pod nazivom hdmi\_core. Ovo jezgro se koristi za slanje sadržaja framebuffer-a Linux Kernel-a na eksterni HDMI kontroler, u cilju grafičkog interfejsa korisničke aplikacije.

IP jezgro je povezano sa AXI stream periferijom preko `axi_to_dti` i `cast` blokova, ovi blokovi prilagođavaju DTI EOT signal sa AXI TLast signalom koji označavaju kraj transakcije. Ulazni EOT signal označava kraj upisa u IMG RAM memoriju, dok izlazni EOT signal signalizira Data Moveru da je završena transakcija i može upisati baferovane podatke u DDR.



## 7.4 Rezultati implementacije sistema

### 7.4.1 Sinteza i implementacija hardvera

FPGA čipovi se sastoje od velikog broja programabilnih primitivnih blokova i mreže za rutiranje. Ove blokove je potrebno konfigurisati i međusobno ih povezati kako bi se dobila željena funkcionalnost. Kako se tipični dizajnovi koji ciljaju FPGA čipove sastoje od desetina hiljada LUT-ova i ostalih komponenti, ovaj korak nije moguće odraditi ručno. Sintezu i implementaciju digitalnog hardvera je jedino moguće uraditi pomoću automatskog alata.

Proizvođači FPGA čipova uglavnom imaju svoj softverski alat za obavljanje ove radnje. Tako u slučaju Xilinx FPGA čipova potrebno je koristiti Vivado.

Prilikom sinteze hardvera alat hijerarhijski struktuirane HDL modele izravna i napravi model sa jednakom hijerarhijom sačuvan u formatu netliste. Tokom sinteze se rade i razne optimizacije poput deljenja funkcionalnih resursa, logička minimizacija, optimizacija mašina stanja itd... Konačno alat za sintezu može mapirati komponente generisane netliste u primitivne blokove ciljane FPGA arhitekture, tzv Technology Mapping.

Nakon koraka sinteze moguće je proceniti koliko će se hardverskih primitivnih blokova koristiti u konačnoj hardverskoj implementaciji.

Na osnovu toga se može zaključiti da li implementirani hardver zadovoljava ograničenja potrošnje resursa.

Nakon sinteze hardvera potrebno je odraditi Place and Route komponenti na željenom FPGA čipu. To je takođe moguće odraditi automatskim alatom. Nakon ovog koraka moguća je procena vremenskih karakteristika implementiranog hardvera. Pošto su poznata kašnjenja primitivnih blokova i moguća je estimacija kašnjenja mreže za rutiranje može se proceniti da li će dizajn zadovoljavati vremenska ograničenja.

Takođe moguće je proceniti i potrošnju konačne implementacije.

Konačno potrebno je generisati Bitstream fajl kojim će se konfigurisati FPGA čip.

Pored zvaničnih alata za sintezu i implementaciju digitalnog hardvera na FPGA čipovima, postoje i alternativna Open Source rešenja. Problem sinteze rešava alat pod nazivom yosys[22], ovaj alat je brži od zvaničnih alata za sintezu, uz to u nekim slučajevima dobijeni rezultat je bolji po potrošnji resursa i brzini, dok je manje efikasan u mapiranju hardvera na tehnologiju.

Za Place and Route može se koristiti nextpnr[23]. Pored toga postoji projekat SymbiFlow[16] koji nastoji da poveže sve ove alate i obezbedi jedinstveni alat za implementaciju digitalnog hardvera na FPGA čipove nezavisno od proizvođača. Veliki problem u ovom projektu predstavlja što FPGA proizvođači ne objavljuju javno arhitekturu čipova i format Bitstream-a, pa se do ovih informacija mora doći reverznim inženjeringom.

### 7.4.2 Analiza potrošnje hardverskih resursa

U narednoj tabeli biće prikazano zauzeće hardverskih resursa u slučaju PyGears implementacije. Implementirano je IP jezgro koje radi sa slikama dimenzije 240x320 i koristi OpenCV model za detekciju lica, opisan ranije. Biće prikazana potrošnja samo glavnog IP jezgra i ukupnog sistema.

Tabela 7.1: Hardverski resursi nakon sinteze PyGears implementacije

Name	LUT	FF	BRAM	DSP
Total	14,259	14,033	47.5	7
Cascade Classifier	6,216	2,252	35	7

Nakon implementacije dobijaju se oko 10% niže brojke za LUT i FF komponente. Kao što se može videti najkritičniji deo ovog IP jezgra je broj korišćenih BRAM komponenti što je bilo i očekivano na osnovu analize predložene arhitekture.

Može se videti da je projektovano jezgro veoma efikasno u pogledu ostalih hardverskih resursa i ostaje prostora za ubrzanje dodavanjem hardverskih resursa, odnosno paralelizma.

U nastavku je data tabela zauzeća hardverskih resursa u slučaju SystemVerilog implementacije iste arhitekture.

Tabela 7.2: Hardverski resursi nakon sinteze SystemVerilog implementacije

Name	LUT	FF	BRAM	DSP
Total	9,648	13,875	53.5	15
Cascade Classifier	1,605	2,094	41	15

Kao što je i očekivano SystemVerilog verzija troši manje hardverskih resursa. Dodatni hardverski resursi u slučaju PyGears implementacije se uglavno ogledaju u dodatnim LUT-ovima i FF-ovima. Ovi LUT-ovi i FF-ovi implementiraju sinhronizaciju i handshaking u okviru PyGears dizajna. Kao što se može videti postoji razlika između potrošnje DSP blokova i BRAM memorija, ove komponente uglavnom se koriste prilikom implementacije aritmetike i memorije potrebne za samu funkcionalnost dizajna, a ne za implementaciju sinhronizacionih komponenti.

Prilikom sinteze alat može odlučiti na osnovu širina magistrala da li će aritmetičku operaciju implementirati kao DSP blok, pomoću LUT-ova ili drugih komponenti, u ova dva slučaja zbog razlike u širinama internih signala alat za sintezu je istu komponentu početne arhitekture implementirao na različite načine. Tada se može desiti da su neka množenja i sabiranja u PyGears verziji implementirana pomoću LUT-ova, dok su neke RAM ili ROM memorije implementirane kao FF ili distribuirani RAM-ovi.

Pored toga u PyGears verziji se često dodaju redundantne sinhronizacione komponente. Primer toga je kada se iz istog izvora razdvaja signal u dve grane, pa se ti signali dalje

“broadcastuju” na više mesta, u ovoj situaciji određen broj dodatnih LUT-ova i FF-ova se može ukloniti prilikom sinteze. Nažalost zbog prilično jedinstvenog slučaja Vivado ne vrši ovu optimizaciju, usled čega konačna implementacija rezultuje u značajnom povećanju potrošenih hardverskih resursa.

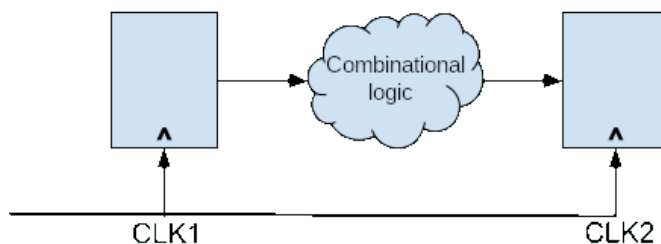
Ova optimizacija je za potrebe PyGears-a dodata u Yosys open source alat za sintezu. Ova tvrdnja najlakše se može videti na primeru CORDIC IP jezgra. Čista Verilog verzija CORDIC IP jezgra je sintetisana pomoću Vivado alata i dobijeno je oko 700 LUT-ova i 800 FF-ova. Nakon toga implementirano je isto IP jezgro u PyGears-u, gde je nakon sinteze u Vivadu dobijeno zauzeće od 2400 LUT-ova i 1100 FF-ova.

Dok je u slučaju prvobitne sinteze pomoću Yosys alata pa nakon toga Vivada dobijeno oko 800 LUT-ova i 900 FF-ova. Kao što se može videti optimizacioni korak u Yosys sintezi značajno može smanjiti broj hardverskih resursa i PyGears metodologija se značajno oslanja na kvalitetan alat za sintezu.

Iako je ovo slučaj, u ovom projektu nije korišćen Yosys za sintezu.

### 7.4.3 Analiza vremenskog izveštaja i Timing Closure

Frekvencija takta je bitna stavka, što većom frekvencijom taktujemo IP jezgro brzina obrade slike će biti brža. Naravno postoji ograničenje maksimalne frekvencije koja se može postići. Ovo ograničenje postoji zbog vremena propagacije signala kroz logičke blokove i mreže za rutiranje unutar FPGA čipa. Izlazni signali flip flopova prolaze kroz kombinacionu logiku i dolaze do ulaza sledećeg flip fropa, kašnjenje ove putanje mora biti kraće od periode takta uz dodatan uticaj Setup i Hold vremena flip flopova.



Slika 7.3: Kašnjenje kombinacione logike

Mreža kombinacione logike predstavljena oblačićem može imati značajno vreme propagacije. Jedan od načina da se vreme propagacije smanji je ubacivanjem registara unutar kombinacione mreže, odnosno presecanje kombinacione putanje.

To je ujedno i najčešća tehnika ubrzavanja dizajna u krajnjoj fazi i zadovoljavanja vremenskih ograničenja dizajna, takozvani Timing Closure.

Nakon podešavanja takta na 47MHz i pokretanja implementacije može se videti da su vremena zadovoljena za odabrani takt.

Empirijski se može zaključiti da frekvencija sistema srednje veličine implementiranog na Zynq-7020 čipu može dostići brzine oko 110MHz, na osnovu čega možemo zaključiti da se u ovom slučaju mogu postići bolji rezultati.

Na slici ispod su prikazane najkritičnije putanje u okviru dizajna.

Name	Slack ^1	Levels	H...	From	To	Total Delay	Logic Delay	Net Delay	Requirement
↳ Path 61	0.015	20	21	hd...K	hdmi_out_i/w...ARDADDR[11]	19.946	12.917	7.029	21.000
↳ Path 62	0.204	20	21	hd...K	hdmi_out_i/w...ARDADDR[10]	19.752	13.001	6.751	21.000
↳ Path 63	0.221	19	21	hd...K	hdmi_out_i/...RARDADDR[5]	19.746	12.658	7.088	21.000
↳ Path 64	0.251	19	21	hd...K	hdmi_out_i/...RARDADDR[8]	19.705	12.754	6.951	21.000
↳ Path 65	0.258	19	21	hd...K	hdmi_out_i/...RARDADDR[6]	19.698	12.762	6.936	21.000
↳ Path 66	0.286	19	21	hd...K	hdmi_out_i/...RARDADDR[7]	19.676	12.678	6.998	21.000
↳ Path 67	0.316	20	21	hd...K	hdmi_out_i/...RARDADDR[9]	19.651	12.897	6.754	21.000
↳ Path 68	0.571	19	21	hd...K	hdmi_out_i/...RARDADDR[4]	19.385	12.710	6.675	21.000
↳ Path 69	1.546	24	14	hd...K	hdmi_out_i/wr...eg_reg[13]/D	18.775	9.879	8.896	21.000
↳ Path 70	1.650	24	14	hd...K	hdmi_out_i/wr...eg_reg[12]/D	18.671	9.775	8.896	21.000
↳ Path 71	1.663	23	14	hd...K	hdmi_out_i/wr...reg_reg[9]/D	18.658	9.762	8.896	21.000
↳ Path 72	1.671	23	14	hd...K	hdmi_out_i/wr...eg_reg[11]/D	18.650	9.754	8.896	21.000
↳ Path 73	1.747	23	14	hd...K	hdmi_out_i/wr...eg_reg[10]/D	18.574	9.678	8.896	21.000
↳ Path 74	1.767	23	14	hd...K	hdmi_out_i/wr...reg_reg[8]/D	18.554	9.658	8.896	21.000
↳ Path 75	1.780	22	14	hd...K	hdmi_out_i/wr...reg_reg[5]/D	18.541	9.645	8.896	21.000

Slika 7.4: Vremenski izveštaj pre skraćivanja kombinacionih putanja

Takođe može se i primetiti da je Logic Delay koji predstavlja vreme propagacije kroz kombinacionu logiku veće od Net Delay-a koji predstavlja vreme propagacije kroz mrežu za rutiranje. Na Net Delay se ne može mnogo uticati i uveliko zavisi od kvaliteta korišćenog FPGA čipa.

Može se primetiti veliko odstupanje između prvih desetak najkritičnijih putanja, kao i da su putanje koje imaju Logic Delay od približno 13ns grupisane u okviru jednog interfejsa. To je znak da postoje putanje koje imaju značajno duže vreme propagacije od ostatka sistema, pa je njih potrebno ubrzati.

Pomoću Vivado alata moguće je identifikovati putanju propagacije kritičnih putanja i odlučiti gde bi se kombinaciona putanja mogla prekinuti i ubaciti registar. Ovo je dugotrajan proces i mora se raditi iz više iteracija, nakon svakog ubacivanja registara mora se verifikovati funkcionalna ispravnost sistema, zatim se ponovo pokreće implementacija.

U svakoj narednoj iteraciji sve je teže uočiti kritične putanje.



PyGears dodatno olakšava proces skraćivanja kombinacionih putanja jednostavnim ubacivanjem registara u dizajn što je prikazano na sledećem primeru.

Primer 7.1: PyGears implementacija stddev komponente

---

```
1 from pygears import gear
2 from pygears.typing import Queue, Uint
3 from .frame_sum import frame_sum
4 from pygears.lib.rom import rom
5 from pygears.lib import dreg as dreg_sp
6
7 @gear
8 def stddev(ii_s: Queue[Uint['w_ii']], 2),
9           sii_s: Queue[Uint['w_sii']], 2), *,
10          casc_hw):
11
12     ii_sum = ii_s | frame_sum | dreg_sp
13     ii_sum_squared = ii_sum[0] * ii_sum[0]
14
15     sii_sum = sii_s | frame_sum | dreg_sp
16     sii_mult1 = sii_sum[0] * (casc_hw.frame_size[0] - 1)
17     sii_mult2 = sii_mult1 * (casc_hw.frame_size[1] - 1)
18
19     sub_s = sii_mult2 - ii_sum_squared
20
21     sqrt_addr = sub_s >> casc_hw.sqrt_shift | Uint[8]
22
23     stddev_res = sqrt_addr | rom(
24         data=casc_hw.sqrt_mem, dtype=Uint[casc_hw.w_sqrt])
25
26     return stddev_res
```

---

U kodnom segmentu iznad u liniji 5 importuje se gear dreg koji predstavlja registar i lokalno se naziva dreg\_sp. Ovo može biti koristan detalj jer pomaže u boljoj distinkciji između registara koji su prisutni funkcionalno i onih koji su ubačeni radi skraćivanja kombinacionih putanja.

U linijama 12 i 15 može se videti skraćivanja kombinacione putanje posle frame\_sum gear-a ubacivanje dreg\_sp gear-a.

Dodatna prednost PyGears-a u ovoj situaciji je činjenica da se ubacivanjem registara unutar dizajna ne može izgubiti funkcionalna korektnost sistema. Iako se ne može izgubiti funkcionalna korektnost, nasumičnim ubacivanjem registara se može pogoršati vremenske performanse. Prilikom ubacivanja registara može doći do gubljenja balansa između dve paralelne grane, pa će jedan podatak uvek kasniti jedan takt do sinhronizacionog gear-a u

odnosu na drugu granu, usled čega će se na izlazu sinhronizacionog gear-a pojaviti neželjeni takt pauze. Pažljivom analizom paralelnih grana i sinhronizacionih tačaka, moguće je balansirati grane dodavanjem dodatnog registra u nebalansiranu granu.

Konačno nakon skraćivanja kombinacionih putanja dostignuta je frekvencija takta od 100MHz.

Pored skraćivanja kombinacionih putanja dodatno je uključena i Performance ExplorePostRoutePhysOpt strategija za implementaciju u okviru Vivado alat, ova strategija će učiniti da vreme implementacije traje duže, ali će se kao rezultat dobiti implementacija sa boljim vremenskim karakteristikama, ponekad sa cenom dodatnih hardverskih resursa.

Kao što se može videti dostignuto je više nego duplo ubrzanje sistema ovom tehnikom. Treba napomenuti da je u RTL metodologiji preporučljivo voditi računa o kombinacionim putanjama u ranijim fazama implementacije, dok je zbog lakog ubacivanja registara u kasnijim fazama dizajna u PyGears metodologiji pogodniji ovakav pristup.

Name	Slack ^1	Levels	Hig...	Fr...	To	Total Delay	Logic Delay	Net Delay	Requirement
↳ Path 41	0.046	13	31	h...	h...	9.346	3.722	5.624	10.0
↳ Path 42	0.046	13	31	h...	h...	9.346	3.722	5.624	10.0
↳ Path 43	0.046	13	31	h...	h...	9.346	3.722	5.624	10.0
↳ Path 44	0.046	13	31	h...	h...	9.346	3.722	5.624	10.0
↳ Path 45	0.085	7	2	h...	h...	8.974	6.166	2.808	10.0
↳ Path 46	0.096	12	31	h...	...	9.261	3.598	5.663	10.0
↳ Path 47	0.096	12	31	h...	...	9.261	3.598	5.663	10.0
↳ Path 48	0.096	12	31	h...	h...	9.261	3.598	5.663	10.0
↳ Path 49	0.115	13	31	h...	h...	9.645	3.722	5.923	10.0
↳ Path 50	0.120	12	31	h...	h...	9.239	3.598	5.641	10.0

Slika 7.5: Vremenski izveštaj posle skraćivanja kombinacionih putanja

Može se primetiti da je dostignuto ogromno skraćenje Logic Delay vremena, a može se primetiti i da je Net Delay značajno skraćen to je verovatno rezultat lakšeg zadatka za Place and Route alat nakon ubacivanja registara.

Na slici je prikazano 10 kombinacionih putanja u prošlom slučaju smo primetili veliku razliku ukupnog kašnjenja u istom broju putanja, u ovom slučaju ne bismo primetili veliku razliku u totalnom kašnjenju čak i kod prvih 100 kritičnih putanja, to je pokazatelj da se dolazi do granica mogućnosti ubrzavanja sistema ovim pristupom.

U tom trenutku dolazimo do Timing Closure-a, kada smo zadovoljni sa vremenskim karakteristikama sistema.

Nakon implementacije bez skraćivanja kombinacionih putanja SystemVerilog dizajna dostiže se frekvencija takta od 66MHz. Takođe bi i za ovu implementaciju bilo potrebno odraditi skraćivanje kombinacionih putanja, ali to u ovom radu nije obrađeno.

Za očekivati je da se može dostići malo veća krajnja frekvencija takta u odnosu na PyGears

implementaciju. Pored toga proces skraćivanja kombinacionih putanja može biti i značajno teži u slučaju SystemVerilog implementacije i usled nepažnje može doći do narušavanja funkcionalnosti dizajna.

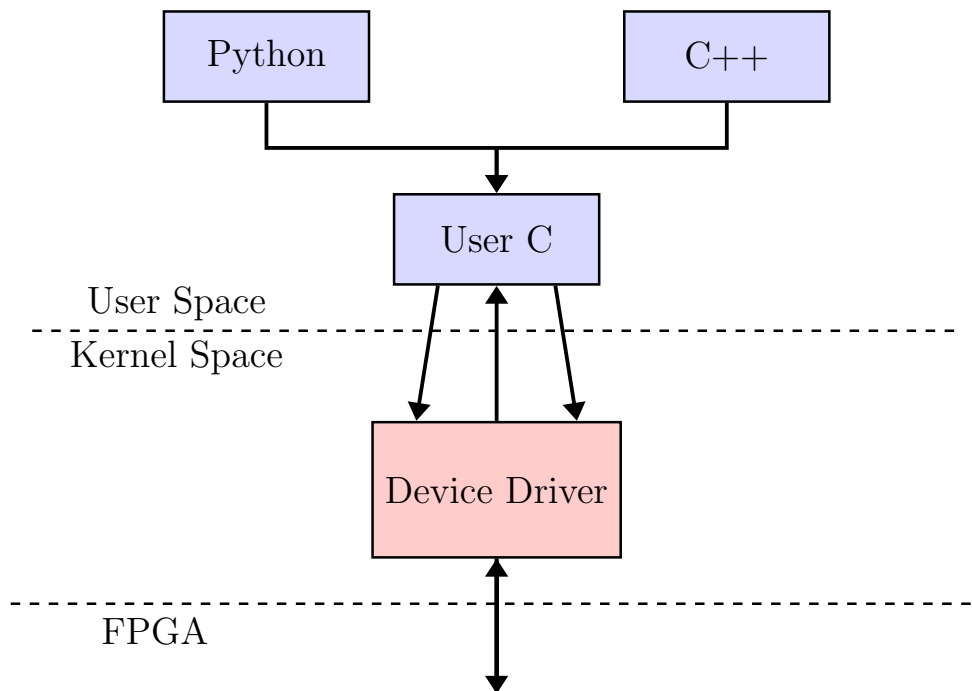
## 7.5 Softver procesora

Nakon što je isprojektovan i implementiran sistem, potrebno je napisati softver za procesor koji će konfigurisati IP jezgro i koristiti rezultate detekcije. Ovde postoji izbor da li će procesor izvršavati operativni sistem ili će izvršavati Bare Metal aplikaciju.

Prednosti Bare Metal aplikacije je veća pouzdanost, jednostavnost i bolje performanse zbog nepostojanja schedule-inga.

Prednosti operativnog sistema je brži i lakši razvoj aplikacije, nezavisnost aplikacije od platforme, laka implementacija korisničkog interfejsa, lak pristup mrežama i internetu itd...

U ovom radu je odlučeno da procesor radi na GNU/Linux operativnom sistemu. Softver je partitionisan na sledeći način:



Slika 7.6: Softverski stek.

Kernel Device driver šalje komande dm\_cmd komponenti preko AXI Lite interfejsa. Procesor komunicira sa AXI Lite komponentom kao sa memorijski mapiranom periferijom. Registri unutar dm\_cmd komponente su mapirani na određenu memorijsku lokaciju u okviru adresnog prostora procesora. Procesor može pristupiti ovim registrima prostim pisanjem na adresu mapiranih registara.

AXI Lite dm\_cmd komponenta se sastoji od sledećih registara.

Tabela 7.3: Registarska mapa dm\_cmd komponente

Address Offset	Register Name	Description
0x00	MM2S_ADDR	Address of source image in DDR.
0x04	MM2S_BTT	Bytes to transfer for source image.
0x08	S2MM_ADDR	Destination address for results.
0x0C	S2MM_BTT	Max buffer size for results.
0x10	IRQ_ENABLE	Enable register for interrupt signal.
0x14	START_CMD	Start register.

Kernel drajver konfiguriše IP jezgro upisivanjem potrebnih podataka na prethodno pokazane registre. Pored toga interrupt signal koji označava kraj obrade slike se obrađuje unutar Kernel drajvera.

Kernel drajver ima implementirane sledeće operacije:

- **Write** operacija startuje IP jezgro upisivanjem potrebnih adresa u dm\_cmd komponentu i postavlja bit za start na jedinicu.
- **Read** operacija vraća sadržaj bafera za rezultate korisničkoj aplikaciji.
- **Mmap** operacija memorijski mapira bafer slike iz korisničke aplikacije na fizički kontinualni memorijski bafer unutar kernel prostora. Fizičku adresu kontinualnog memorijskog bafera je potrebno poslati dm\_cmd komponenti unutar Write operacije.

Prilikom završetka obrade slike poslaće se zahtev za interrupt procesoru, taj zahtev će se obraditi u okviru Kernel drajvera. Nakon pojave interrupta poslaće se komanda za resetovanje interrupt registra u okviru dm\_cmd komponente, zatim će se otključati read\_mutex kako bi se omogućila operacija Read.

Korisnička aplikacija je podeljena u dva nivoa. Niži nivo aplikacije predstavlja C funkciju koja je zadužena za komunikaciju sa kernel drajverom. Ova funkcija obavlja operacije čitanja rezultata i memorijskog mapiranja prosledene slike na kernel memorijski bafer. Funkciju je moguće kompajlirati kao Shared Object (SO) biblioteku tako da je moguće pristup iz Python-a preko ctypes biblioteke, kao i pristup iz C++-a.

Viši nivoi korisničke aplikacije mogu biti napisani u programskim jezicima višeg nivoa. U ovom radu je napisana aplikaciju u Python jeziku. Prednost korišćenja jezika kao što je Python je jednostavno korišćenje biblioteka poput OpenCV-a. Pomoću OpenCV biblioteke može se na lak način pročitati slika iz fajla ili preuzeti frejm sa kamere, zatim pretvoriti u grayscale reprezentaciju.

Takođe kao vizualizaciju rada IP jezgra u okviru OpenCV biblioteke može se na lak način nacrtati pravougaonici na pozicijama detektovanih objekata.

## 8 Performanse i optimizacije

### 8.1 Performanse

Performanse projektovanog IP jezgra biće upoređene sa softverskim implementacijama. Poređiće se C++ specifikacija napisana u ovom radu, OpenCV implementacija i hardverska implementacija.

Mereno je samo vreme izvršavanja detekcije, odnosno isti zadatak koji obavlja IP jezgro obavlja i softver. OpenCV implementacija biće poređena sa automatskim skaliranjem slike i sa fiksnim faktorom skaliranja korišćenim u hardverskoj implementaciji.

Poređene platforme su Ryzen 5 1600 procesor pod Arch Linux-om, Thinkpad T430s i5-3210m pod Arch Linux-om i Zynq 7020 pod Arch Linux ARM-om.

U sledećoj tabeli je prikazano prosečno vreme detekcije na Caltech dataset-u slika veličine 240x320.

Tabela 8.1: Poređenje performansi

Platform	OpenCV Scaling Auto	OpenCV Scaling Fixed	C++ Spec	IP Core
Zynq	571 ms	205 ms	3593 ms	926 ms
Thinkpad	28 ms	10.3 ms	194 ms	N/A
Ryzen	22.8 ms	9.7 ms	192 ms	N/A

Kao što je bilo i očekivano OpenCV implementacija pod Ryzen procesorom ima najkraće vreme izvršavanja. Može se videti da iako Ryzen procesor ima 6 bržih CPU jezgara za razliku od 2 kod Thinkpad laptopa ne postoje drastične razlike u vremenu izvršavanja softverskih implementacija.

Zynq je očekivano najsporiji od tri platforme. Projektovano IP jezgro je 4.5 puta sporije od OpenCV implementacije na istoj platformi.

Iako su performanse IP jezgra slabije od OpenCV implementacije treba uzeti u obzir i odnos frekvencija procesora i IP jezgra. Zynq 7020 se sastoji od dva Cortex-A9 procesora taktovana sa 866MHz, što znači 8.5 puta veća frekvencija takta od IP jezgra. Pored toga projektovano IP jezgro zauzima samo oko 15% hardverskih resursa Zynq čipa, što znači da postoji prostora za ubrzanjem paralelizmom.

Prednost FPGA implementacije leži i u tome što je procesor značajno manje opterećen prilikom detekcije, pa je procesor slobodan da radi ostale zadatke u paraleli. Pošto IP jezgra zauzimaju malo hardverskih resursa, postoji mogućnost instancioniranja više IP jezgra u sistemu i paralelne detekcije više slika istovremeno, pritom bez gubitaka performansi usled paralelizma.

## 8.2 Optimizacije

Performanse OpenCV softverske implementacije je moguće postići uvođenjem nekih izmena u hardverskoj arhitekturi, neke od izmena su:

### 8.2.1 Refaktorisanje generisanja integralne slike

Trenutni način generisanja integralne slike je jednostavan, ali veoma neefikasan.

Kao što je prikazano na slici(5.3), prilikom iteracije hopper-a po X koordinati, prethodno izračunati pikseli integralne slike se odbacuju i integralna slika se računa za ceo sledeći prozor. Moguće je iskoristiti vrednosti integralne slike za kolone 1-4, zatim je potrebno izračunati samo 5. kolonu integralne slike u sledećoj iteraciji.

Na slici(8.1) na prvom interfejsu može se videti vreme računanja integralne slike u odnosu na vreme rada klasifikatora. U slučaju ove optimizacije to vreme bi bilo značajno smanjeno.

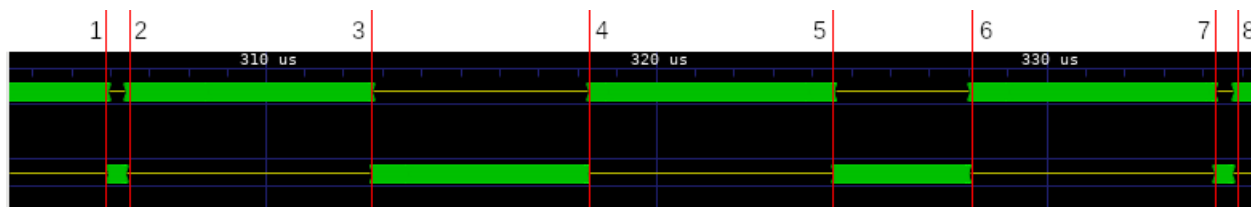
Ovo će dovesti do značajnog rasta vrednosti integralne slike, pa je zbog toga potrebna i veća memorija u okviru frame\_buffer modula, kao i šire magistrale aritmetičkih operacija u classifier modulu. Pored toga ii\_gen i rd\_addrngen moduli bi bili značajno komplikovaniji.

Iako bi se količina memorije i broj aritmetičkih funkcionalnih jedinica povećao, ovu optimizaciju bi trebalo prvo razmatrati u slučaju daljeg ubrzanja IP jezgra. Ukoliko bi se optimizacija implementirala moglo bi se očekivati značajno ubrzanje IP jezgra.

### 8.2.2 Generisanje integralne slike tokom rada klasifikatora

Nakon što ii\_gen generiše integralnu sliku, ona se skladišti u frame\_buffer modul. Tokom rada klasifikatora ii\_gen čeka na rezultat klasifikatora.

Na sledećoj slici se mogu videti interfejs za upis i čitanje frame\_buffer modula. Gornji interfejs predstavlja izlaz iz ii\_gen modula i povezan je interfejsom za upis frame\_buffer memorije. Donji interfejs je izlaz frame\_buffer modula.



Slika 8.1: Performanse frame\_buffer modula

Na osnovu aktivnosti donjeg interfejsa može se videti vreme rada klasifikatora, kao što se može videti to vreme je promenljivo. Vreme aktivnosti gornjeg interfejsa je konstantno, jer je vreme računanja integralne slike konstantno.

Nakon završetka računanja integralne slike u trenutku kursora 1, klasifikator počinje sa radom. Rad klasifikatora traje do kursora 2. U ovom slučaju vreme rada klasifikatora je kratko jer je prozor odbačen posle prve etape.

Nakon kursora broj 2 počinje generisanje integralne slike i traje do kursora 3. Integralna slika se generiše takođe i između kursora 4-5 i 6-7, kao što se može videti ovo vreme je konstantno.

Između kursora 3-4 može se videti značajno duža aktivnost na donjem interfejsu, posledica toga je što je klasifikator stigao do 4. etape nakon čega je odbačen prozor. Između kursora 5-6 klasifikator odbacuje se prozor nakon 3. etape.

Može se videti da se u periodu između kursora 3-4 može izračunati ceo prozor integralne slike tokom rada klasifikatora. Kako bi se ovo odradilo potrebno je duplirati `frame_buffer` memoriju.

### 8.2.3 Paralelno računanje prve etape

Kao što se vidi na grafiku(1.6) prva etapa će se izvršiti u svakom analiziranom prozoru, procenat izvršavanja svake naredne etape eksponencijalno opada. Posle 5. etape procenat izvršavanja je manje od 1%. Prva etapa ima 9 obeležja, a zatim svaka naredna eksponencijalno više. Pošto će se veliki procenat vremena prilikom rada IP jezgra provesti na računanje prve etape, računanje ove etape bi se moglo računati posebnim prilagođenim klasifikatorom.

### 8.2.4 Paralelni klasifikatori

Kako bi se omogućio rad paralelnih klasifikatora, potrebno je izdeliti ulaznu IMG RAM memoriju na regione. Tako da će svaki paralelni klasifikator obrađivati svoj region.

U ovom slučaju treba umnožiti `ii_gen`, `sii_gen`, `frame_buffer`, `stddev` module. Adresu za čitanje koju generiše `rd_addrngen` je moguće izvesti za svaki region uvođenjem `offset`-a adrese, tako da `rd_addrngen` nije potrebno umnožiti u ovom slučaju. Moguće je koristiti jednu `features_mem` memoriju za sve paralelne klasifikatore, ali u tom slučaju svi paralelni klasifikatori će raditi brzinom najsporijeg klasifikatora. Moguće je ponovno koristiti i memorije iz `classifier` modula, što uvesti isti efekat kao i `features_mem` memorije. Aritmetičke operatore iz `classifier` modula je potrebno umnožiti.

### 8.2.5 Računanje više obeležja u paraleli

Kao što se može videti na blok dijagramu klasifikatora(5.12) trenutno se jedno obeležje računa u trenutku. Potrebno je četiri takta da se površina jednog pravougaonika izračuna. Moguće je računati 2 obeležja u paraleli. U tom slučaju bi `frame_buffer` memorija bila duplirana zbog većeg broja portova za čitanje. Dok bi `features_mem` memorija bila značajno komplikovanija, kao i `classifier` modul.



### 8.2.6 Računanje skaliranih slika u paraleli

Kao što se može videti na slici(1.7) originalna slika se mora obrađivati više puta nakon skaliranja. Rezultati obrade slika na različitim skalama nemaju međusobnu zavisnost, zbog toga je moguće obrađivati ih u paraleli.

Primenom ove optimizacije ne može se dobiti linearno uvećanje brzine. Pošto je najviše vremena potrebno da se obradi originalna slika, dok je za svaku narednu skaliranu sliku potrebno sve manje vremena za obradu. Dobitak performansi ove optimizacije verovatno nije opravdan za dodatne hardverske resurse.

## 9 Zaključak

U ovom radu je uspešno projektovana arhitektura hardverskog akceleratora Viola-Jones algoritma. Performanse hardverskog rešenja nisu dostigle performanse OpenCV softverske implementacije.

Predloženim optimizacijama moguće je dostići približne performanse OpenCV implementacije. Bolje performanse bi se postigle i korišćenjem većeg i kvalitetnijeg FPGA čipa, na kojem bi se sistem implementirao sa višom frekvencijom takta. U slučaju Application Specific Integrated Circuit (ASIC) implementacije moguće je dostići mnogo više frekvencije rada, pa dobiti značajno brže rešenje od softverskog.

Tokom implementacije hardverske arhitekture korišćen je inovativni pristup opisa hardvera korišćenjem PyGears metodologije. Takođe ista arhitektura je implementirana i u standardnoj RTL SystemVerilog implementaciji. Zaključeno je da funkcionalni pristup koji nameće PyGears metodologija može značajno ubrzati razvoj u nekim situacijama, dok je pojedine komponente jednostavnije implementirati RTL metodologijom.

Dodatni hardverski resursi u slučaju PyGears metodologije mogu biti zanemarljivi za manje dizajnove i FPGA čipove. U slučaju ASIC implementacije često je potrebno dobiti što minimalniju implementaciju i tada dodatni hardverski resursi uveliko utiču na cenu i performanse, u tom slučaju RTL metodologija ima veliku prednost.

U radu je uspešno implementiran sistem na Zynq SoC platformi, prilikom čega je bilo potrebno napisati Linux Device Driver. Isto tako i konfigurisati i kompajlirati U-Boot i Linux Kernel za potrebe projektovanog embeded sistema.

Napisana korisnička aplikacija u dva hijerarhijska nivoa, omogućava jednostavnu komunikaciju sa projektovanim Kernel Driver-om iz različitih programskih jezika. U ovom slučaju najviši nivo hijerarhije korisničke aplikacije je napisan u Python-u.

## Literatura

- [1] A. Jain, “Computer vision – face detection,” 2016. [Online]. Available: <https://vinsol.com/blog/2016/06/28/computer-vision-face-detection/>
- [2] O. Jensen, “Implementing the viola-jones face detection algorithm,” 2008.
- [3] P. A. Viola and M. J. Jones, “Rapid object detection using a boosted cascade of simple features,” in *CVPR*, 2001.
- [4] Z. Ye, “5kk73 gpu assignment 2012,” 2012. [Online]. Available: <https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection>
- [5] M. Weber, “Frontal face dataset,” 1999. [Online]. Available: [www.vision.caltech.edu/Image\\_Datasets/faces/faces.tar](http://www.vision.caltech.edu/Image_Datasets/faces/faces.tar)
- [6] 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>
- [7] Myir, “Zturn board.” [Online]. Available: <http://www.myirtech.com/list.asp?id=502>
- [8] K. Cen, “Study of viola-jones real time face detector,” 2016.
- [9] OpenCV, “Opencv docs.” [Online]. Available: [https://docs.opencv.org/3.4.3/d7/d8b/tutorial\\_py\\_face\\_detection.html](https://docs.opencv.org/3.4.3/d7/d8b/tutorial_py_face_detection.html)
- [10] Xilinx, “Xst user guide.” [Online]. Available: [https://www.xilinx.com/support/documentation/sw\\_manuals/xilinx10/books/docs/xst/xst.pdf](https://www.xilinx.com/support/documentation/sw_manuals/xilinx10/books/docs/xst/xst.pdf)
- [11] —, “Axi reference guide.” [Online]. Available: [https://www.xilinx.com/support/documentation/ip\\_documentation/ug761\\_axi\\_reference\\_guide.pdf](https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf)
- [12] Intel, “Avalon® interface specifications.” [Online]. Available: [https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl\\_avalon\\_spec.pdf](https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/manual/mnl_avalon_spec.pdf)
- [13] B. Vukobratović, “Pygears.” [Online]. Available: [www.pygears.org](http://www.pygears.org)
- [14] P. P. Chu, *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. John Wiley & Sons, 2006.
- [15] 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>
- [16] “Symbiflow.” [Online]. Available: <https://symbiflow.github.io>
- [17] J. Decaluwe, “Myhdl: a python-based hardware description language.” *Linux journal*, no. 127, pp. 84–87, 2004.

- [18] 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.
- [19] “Spinalhdl,” <https://github.com/SpinalHDL/SpinalHDL>, accessed: 2018-08-12.
- [20] 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.
- [21] “Arch linux arm.” [Online]. Available: <https://archlinuxarm.org/>
- [22] C. Wolf, “Yosys.” [Online]. Available: <http://www.clifford.at/yosys/>
- [23] “Nextpnr.” [Online]. Available: <https://github.com/YosysHQ/nextpnr>