

Projektovanje, implementacija i verifikacija embedded sistema za prepoznavanje životinja korišćenjem namenske duboke neuronske mreže na Zybo platformi

David Vidović
EE81/2019

Ivan Milin
EE179/2019

Ivan Čejčić
EE17/2019

29. avgust 2023.

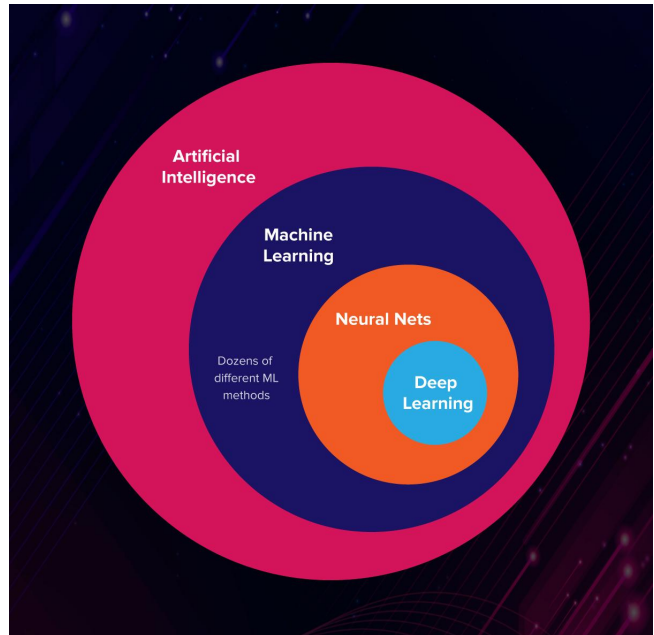
Sadržaj

1	Uvod	3
2	Specifikacija	3
3	Procena vremenske zahtevnosti	5
4	Bitska analiza	6
5	Virtuelna platforma	7
6	RTL	10
6.1	Datapath	10
6.1.1	Ulazna memorije za skladistenje slike i tezina	10
6.1.2	Memorija za bias-e i izlazne podatke	10
6.1.3	Linijski bafer	11
6.1.4	Keš blok za sliku	11
6.1.5	Keš blok za tezine	12
6.1.6	MAC modul	12
6.1.7	Modul za sabiranje	13
6.2	Controlpath	13
6.3	TOP module	17
6.4	Zauzetost resursa i procenjena frekvencija rada	17
7	Funkcionalna verifikacija hardvera	18
7.1	Verifikacioni plan	18
7.2	Struktura verifikacionog okruženja	18
7.2.1	Top modul, test, environment i configuration	20
7.2.2	Agent_axi_lite	20
7.2.3	Agent_axi_stream_master	20
7.2.4	Agent_axi_stream_slave	20
7.2.5	Sequences	20
7.2.6	Scoreboard	21
7.2.7	Prikupljanje pokrivenosti i regresivni testovi	21
8	Razvoj Linux drajvera	22
8.1	Init i exit funkcije	22
8.2	Probe i remove funkcije	23
8.3	Read, write i mmap funkcije	24
8.4	Prekidne rutine	25
9	Zaključak	26

1 Uvod

Algoritmi veštačke inteligencije postaju sve zastupljeniji u današnjim aplikacijama kako bi zamenili čoveka u svakodnevnim poslovima. Oni omogućavaju mašinama da vide svet na način na koji ga i čovek vidi, ali takođe i da koristi znanja radi prepoznavanja objekata na slikama, analizu slika i njihovu klasifikaciju, obradu prirodnog govora i još mnogo toga. Konvolucijska neuronska mreža (CNN) je algoritam koji spada u klasu takozvanih 'Deep learning' algoritama, što je podskup algoritama mašinskog učenja. Odnos različitih algoritama u polju veštačke inteligencije mogu se videti na slici 1.

Ovakvi algoritmi na svom ulazu dobijaju sliku i na osnovu arhitekture mreže i baze podataka nad kojim je trenirana, mreže mogu da prepoznaju šta je na slici.



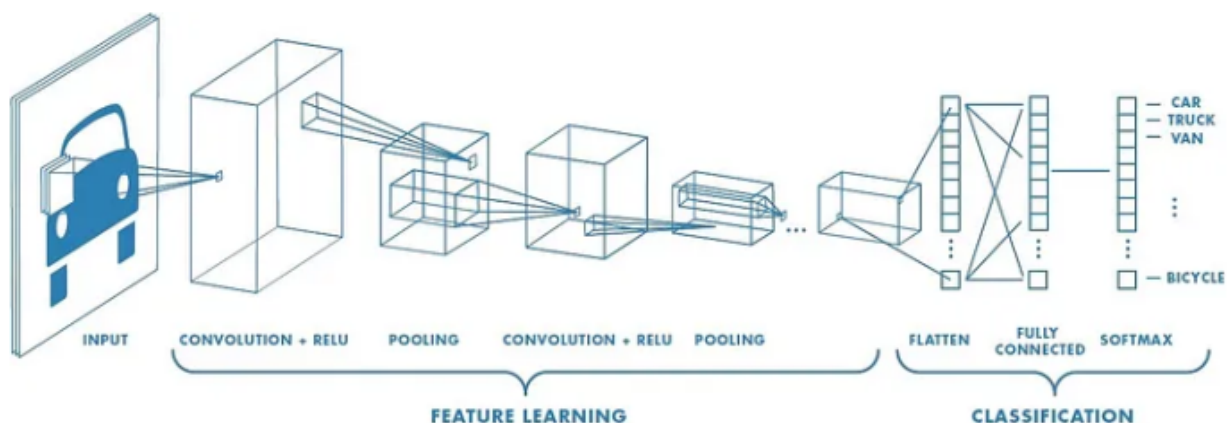
Slika 1: Odnos različitih algoritama u oblasti veštačke inteligencije

2 Specifikacija

Za potrebe projekta korišćena je CIFAR-10 baza podataka. Ona se sastoji od 60000 slika koje imaju dimenzije 32x32 piksela u boji, a broj klasa u okviru baze je 10. Svaka klasa sadrži 6000 slika, a klase koje se nalaze u okviru baze podataka su: avion, automobil, ptica, mačka, jelen, pas, žaba, konj, brod i kamion. Baza podataka je podeljena na trening skup koji se sastoji od 50000 slika, na kojem će se mreža trenirati, i na test skup od 10000 slika, na kojem će biti testirana tačnost mreže. Cilj je projektovanje mreže koja će što uspešnije klasifikovati date objekte. Svaka konvoluciona mreža se sastoji od više konvolucijskih, pooling i fully connected slojeva. Osnovna arhitektura mreže je prikazana na slici 2.

Konvolucijski slojevi vrše matematičku operaciju konvoluciju nad matricama i na taj način izvlače važne karakteristike slike koje će služiti za klasifikaciju. U okviru jednog konvolucijskog sloja može se nalaziti veći broj filtera, gde svaki filter izvlači jednu karakteristiku slike. Na izlazu konvolucijskog sloja se generisana mapa obeležja, koja se propušta kroz aktivacionu funkciju. Često kao aktivaciona funkcija se koristi ReLU(rectified linear unit).

$$f(x) = \max(0, x) \quad (1)$$



Slika 2: Osnovna arhitektura CNN-a

Pooling sloj ima zadatak da smanji veličinu slike koja prolazi kroz mrežu, ali se očuvavaju najvažnije karakteristike koje su izvučene pomoću konvolucijskih slojeva. Na ovaj način se ujedno i smanjuje broj parametara za opis mreže. Najčešći tip pooling sloja je max pooling, koji deli mapu obeležja na pravougaone regione, koji se ne preklapaju, i uzima maksimalne vrednosti u svakom od regiona kao izlaz.

Dense sloj (fully connected sloj) je sloj koji povezuje sve neurone prethodnog sloja sa svim neuronima sledećeg sloja. Za razliku od konvolucijskog i pooling sloja, dense sloj ima ulogu klasifikatora. Ovaj sloj koristi mapu obeležja koja je napravljena od strane prethodnih slojeva kako bi izvršio klasifikaciju.

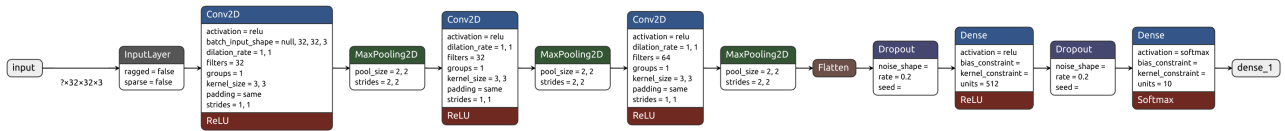
Nakon što su objašnjeni glavni delovi konvolucijske neuralne mreže, na slici 3 je prikazana arhitektura koja je korišćena u ovom projektu. Arhitektura mreže, koja je izabrana za klasifikaciju CIFAR-10 baze podataka se sastoji od 3 konvolucijski slojeva gde je posle svakog dodat po jedan max-pooling sloj. Prvi i drugi konvolucijski slojevi sastoje se od 32 filtera (kernel-a) koji je svaki veličine 3x3, dok treći sloj ima 64 filtera iste veličine. Aktivaciona funkcija svakog konvolucijskog layera je ReLU, koja je data formulom 1.

Svaki max-pooling sloj se sastoji od prozora veličine 2x2 koji smanjuje mapu obeležja sa faktorom 2 (ulaz: 32x32 32 kanala ; izlaz: 16x16 32 kanala).

U arhitekturi mreže nalaze se 1 skriveni dense sloj od 512 neurona, čija je aktivaciona funkcija ReLU. Poslednji izlazni dense sloj sadrži 10 neurona (što potiče od 10 klasa koje trebaju da se prepoznaju), i on ima softmax aktivacionu funkciju koja je data formulom 2, gde z_i predstavlja vrednost na izlazu svakog neurona.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{10} e^{z_j}} \quad za \ i = 1, 2, \dots, 10 \quad (2)$$

Ovakva specifikacija, nakon treniranja mreže, daje tačnost od 76.66% na test skupu podataka.



Slika 3: Arhitektura mreže za CIFAR-10 bazu podataka

3 Procena vremenske zahtevnosti

Analiziranjem izvršavanja programa koji implementira CNN može se doći do zaključka koji deo implementacije oduzima najviše resursa i koji deo predstavlja usko grlo pri izvršavanju na samom računaru. Ovi delovi su dobri kandidati za ubrzavanje njihovog izvršavanja na FPGA ploči.

Nakon pokretanja analize vremenske zahtevnosti pri uzorku od 100 slika, rezultati se mogu videti u tabeli 1.

Convolution layer - forward propagation	89.83%
Dense layer - forward propagation	6.46%
Ostalo	3.71%

Tabela 1: Analiza vremena izvršavanja

Može se primetiti da je najviše vremena utrošeno na izvršavanje forward propagation funkcije unutar konvolucijskog sloja, što u suštini predstavlja matematičku operaciju konvlucije nad matricama.

Pošto ovo troši skoro 90% od ukupnog vremena izvršavanja, ovaj deo će biti implementiran i ubrzan u hardveru, dok će ostatak mreže ostati implementiran u softveru.

4 Bitska analiza

Prilikom softverske implementacije mreže korišćen je float tip podataka (vrednosti sa pokretnim zarezom). Implementacija floating point operacija u hardveru je veoma kompleksna, pa je zbog toga izabran fix point format vrednosti u okviru hardverskog modula. Potrebno je pronaći sa koliko bita i u kom formatu je potrebno predstaviti podatke u hardveru kako bi se dobila što približnija tačnost mreže, u odnosu na korišćenje floating point reprezentacije.

Rezultati bitske analize prikazani su u tabeli 2

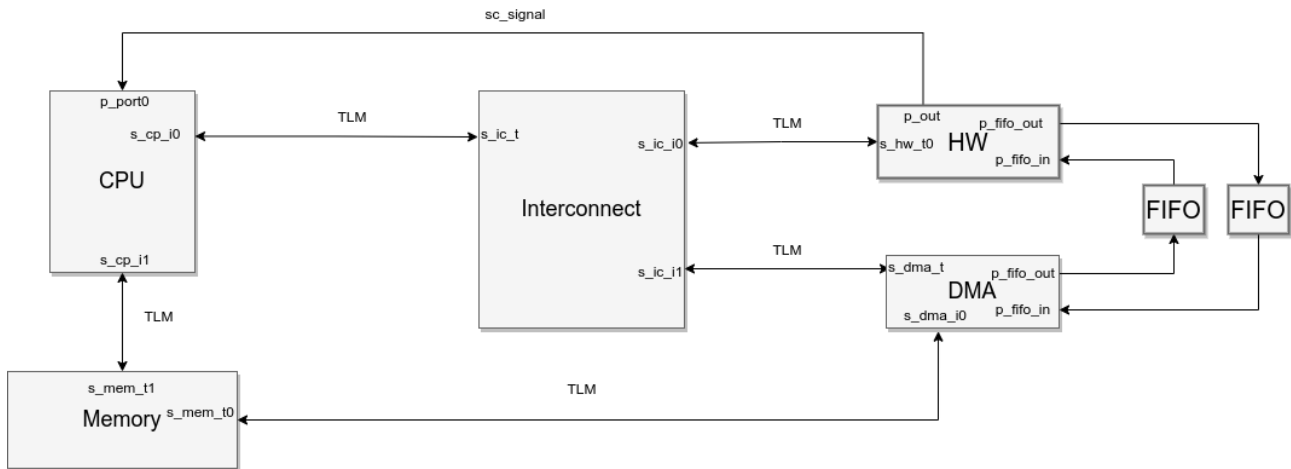
Broj bita	Format	Tačnost
14	4.10	72.55%
	5.9	65.44%
15	4.11	75.33%
	5.10	72.56%
16	4.12	76.1%
	5.11	75.34%
17	4.13	76.4%
	5.12	76.14%
18	4.14	76.61%
	5.13	76.42%
19	4.15	76.67%
	5.14	76.63%

Tabela 2: Bitska analiza

Iz tabele 2 se može primetiti da tačnost mreže prvi put prelazi tačnost od 76% ukoliko se vrednosti u okviru konvolucijskog sloja predstave sa 16 bita u formatu 4.12 (4 bita za celobrojnu vrednost, 12 bita za razlomljeni deo). Dodatno ova reprezentacija je pogodna iz ugla hardvera pošto koristi tačno 2 bajta.

Iz navedenih razloga ova reprezentacija vrednosti će biti i korišćena u hardverskoj implementaciji.

5 Virtuelna platforma



Slika 4: Blok šema virtualne platforme

Na slici 4 prikazana je virtuelna platforma koja je implementirana u systemC jeziku. Komponenta CPU predstavlja softver koji kontrolira rad celog sistema. CPU kontrolira rad hardvera (HW) preko 14 komandi :

1. 00000000000001 - učitaj bias parametre konvolucijskih layera iz memorije preko DMA
2. 00000000000010 - učitaj weight parametre za 1. konvolucijski layer iz memorije preko DMA
3. 000000000000100 - učitaj ulaznu sliku za 1. konvolucijski layer iz memorije preko DMA
4. 000000000001000 - startuj 1. konvolucijski layer
5. 00000000010000 - učitaj polovinu weight parametara za 2. konvolucijski layer iz memorije preko DMA
6. 00000000100000 - učitaj ulaznu sliku za 2. konvolucijski layer iz memorije preko DMA
7. 00000001000000 - startuj 2. konvolucijski layer (izvršava se iz dva puta sa po pola weight parametara)
8. 00000010000000 - učitaj četvrtinu weight parametara za 3. konvolucijski layer iz memorije preko DMA
9. 00000100000000 - učitaj ulaznu sliku za 3. konvolucijski layer iz memorije preko DMA
10. 00001000000000 - startuj 3. konvolucijski layer (izvršava se iz dva puta sa po pola weight parametara)
11. 00010000000000 - restartuj sve
12. 00100000000000 - čitaj izlaznu memoriju nakon 1. konvolucijskog layera
13. 01000000000000 - čitaj izlaznu memoriju nakon 1. konvolucijskog layera
14. 10000000000000 - čitaj izlaznu memoriju nakon 1. konvolucijskog layera

Rad celokupnog sistema biće opisan u narednih nekoliko koraka:

- 1.korak - CPU zadaje komandu 00001 HW komponenti, HW je spreman da učitava bias parametre sva tri konvolucijska layera
- 2.korak - CPU startuje DMA, koji šalje bias podatak po podatak HW bloku na svakih 10ns. Kada HW primi sve podatke, šalje signal CPU komponenti koji sve vreme čeka na njega
3. korak - CPU zadaje komandu 00010 HW komponenti, HW je spreman da učitava weight parametre za 1. konvolucijski layer
4. korak - CPU startuje DMA, koji šalje weight parametre podatak po podatak HW bloku na svakih 10ns. Kada HW primi sve podatke, šalje signal CPU komponenti koji sve vreme čeka na njega
- 5.korak - CPU zadaje komandu 00011 HW komponenti, HW je spreman da učitava paddovanu ulaznu sliku za 1. konvolucijski layer
6. korak - CPU startuje DMA, koji šalje parametre ulazne slike podatak po podatak HW bloku na svakih 10ns. Kada HW primi sve podatke, šalje signal CPU komponenti koji sve vreme čeka na njega
7. korak - CPU zadaje komandu 01000 HW komponenti, kojom zahteva da HW nakon konvolucije pošalje izlaznu sliku u memoriju. Slika se šalje podatak po podatak na svakih 10ns. Kada HW pošalje poslednji podatak iz izlazne slike obaveštava CPU signalom da je transakcija završena
8. korak - CPU čita podatke iz memorije
9. korak - CPU izvršava prvi maxpool layer
10. korak - upisuje podatke nazad u memoriju
11. - 20. - ponavljaju se koraci od 3 do 10, za 2. konvolucijski layer koji se radi iz dva navrata: u HW se učitava celokupna ulazna slika, te se prvo pošalje prva polovina weight parametara i konvolucija se uradi za prvu polovinu izlazne slike (16 filtera) nakon čega se proces ponavlja za drugu polovinu weight parametara i izlazne slike (poslednjih 16 filtera)
21. - 36. ponavljaju se koraci od 3 do 10 za 3. konvolucijski layer, slično kao 2. layer i ovaj je particionisan u više faza izvršavanja. 3. konvolucijski layer se izvršava iz 4 puta (šalje se po 1/4 weight parametara i nakon 4 pokretanja konvolucije dobija se kompletna izlazna slika)
24. korak - CPU izvršava dense layer-e

Protok vremena je simuliran pri prenošenju podataka iz memoriju u HW blok preko DMA i pri izvršavanju konvolucijski layera. Unutar konvolucijskih layera postoji 9 MAC modula koji izvršavaju 9 konkurentnih množenja. Podaci koji se smeštaju u BRAM memoriju u IP bloku su formatirani od strane CPU komponente pre slanja tako da na prigodan način pune linijske buffere. Svaki konvolucijski layer se sastoji od pomenutih linijskih buffera koji nakon inicijalne faze punjenja, na svojim izlazima u svakom narednom taktu imaju sadržaj jednog 3x3 image slicea. Na ulaze MAC modula su dovedeni odgovarajući weight parametri, čime obezbeđujemo 9 paralelnih množenja i u jednom taktu izvršavamo konvoluciju jednog image slicea veličine 3x3 piksela. Zavisno od broja ulaznih kanala, na svaka 3 (za 1. layer) ili na svaka 32 (za 2. i 3. layer) takta se dobija jedan izlazni piksel. On se zatim sabira sa bias parametrom i izvršava se ReLU aktivaciona funkcija.

Prilikom simulacije dolazimo do zaključka da sa trenutnim modelom ukupno vreme potrebno za obradu jedne slike iznosi 6,9ns što daje procenjeni framerate od 145 slika u sekundi. Procenjeno vreme se odnosi samo na simulaciono vreme koje troši HW komponenta prilikom konvolucije i transfera podataka ka i iz memorije. Naša pretpostavka je da će i operacije koje se izvršavaju od strane CPU komponente da oduzimaju vreme, te očekujemo konačni framerate od 100 slika u sekundi.

Od ukupnog vremena za obradu slike, 13,2% vremena se potroši na komunikaciju podacima između memorije i HW komponente, dok preostalih 86,8% simulacionog vremena se troši na izvršavanje konvolucije, čime smo potvrdili pretpostavku da komunikacija ne predstavlja bottleneck sistema. Najviše simulacionog vremena oduzima 2. konvolucion layer, čak 59,5% ukupnog konvolucionog vremena.

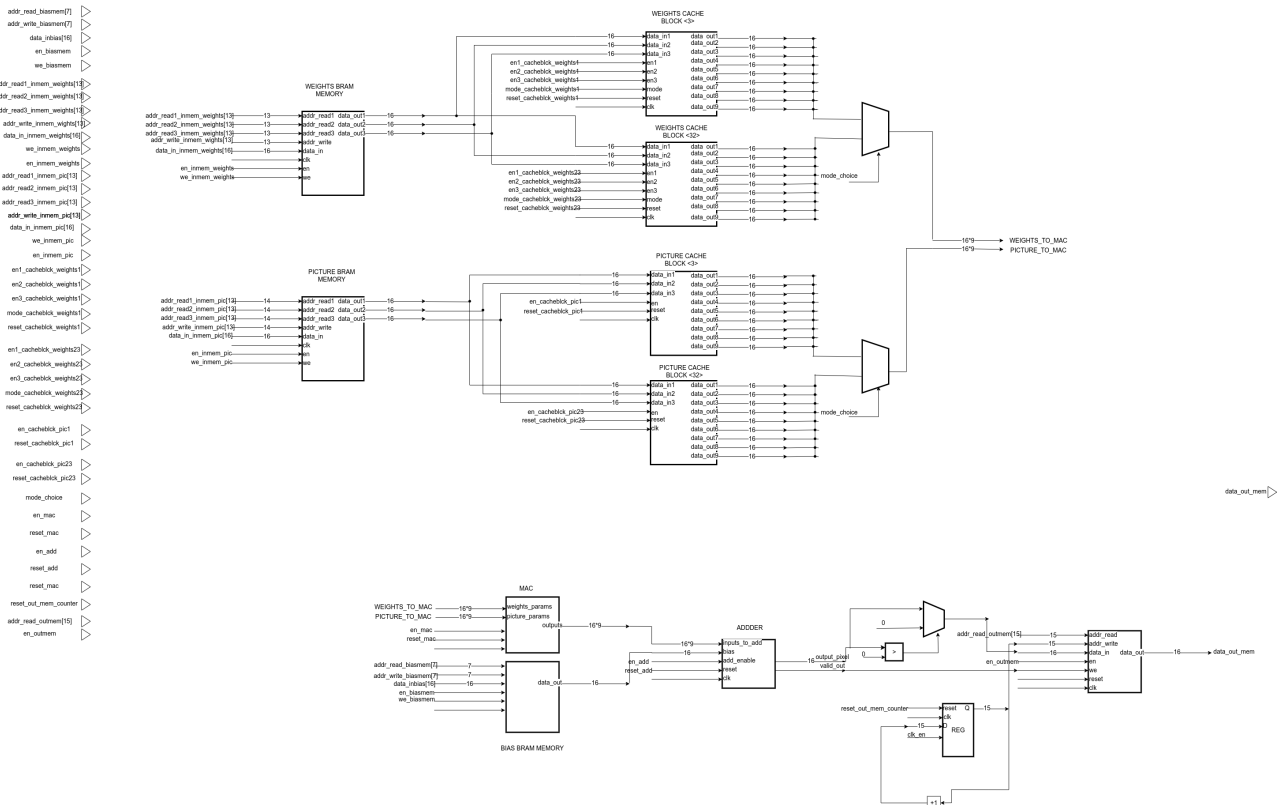
Glavna i najzahtevnija operacija koja se koristi u sistemu je MAC(Multiply-accumulate) operacija. Nakon što se ova operacija implementira u Vivado razvojnom okruženju i mapira na DSP komponentu, procenjena maksimalna frekvencija je 414MHz. Medjutim ubacivanjem dodatne logike, rutiranje postaje problem i nije realno očekivati da ceo sistem radi sa taktom od 414MHz, već će maksimalna frekvencija opasti i više od duplo. Procena je da će ceo sistem moći da radi sa taktom od 150MHz.

6 RTL

Implementacija konvolucije je izvedena u VHDL jeziku u Vivado alatu primenom RTL metodologije. Postoje dva glavna dela dizajna, datapath i cotrolpah. Za svaki od njih će biti opisani interfejs, očekivana funkcionalnost kao i od kojih elemenata se sastoji. Na slici 5 prikazan je

6.1 Datapath

Na slici 5 prikazan je kompletna struktura modula za obradu podataka pa će svaki modul unutar njega biti posebno objašnjen.



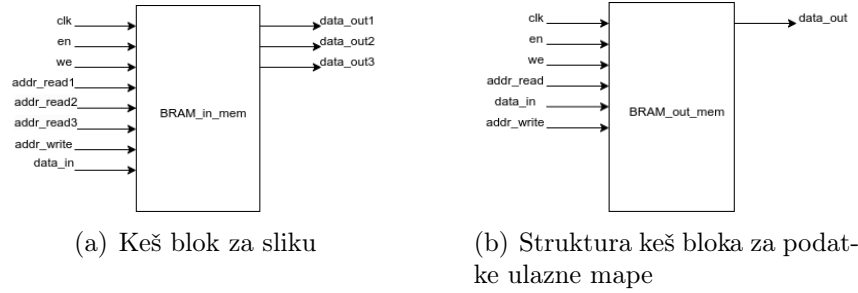
Slika 5: Arhitektura modula za obradu podataka

6.1.1 Ulazna memorije za skladištenje slike i tezina

Na slici 6(a) prikazan je intefejs memorija koje se koriste za skladištenje i citanje podataka slike i tezina. Memorija se sastoji od 3 porta za citanje i 1 porta za pisanje. Ova memorija zauzima 3 puta više BRAM ćelija od predvidjenog zbog 3 porta za čiranje.

6.1.2 Memorija za bias-e i izlazne podatke

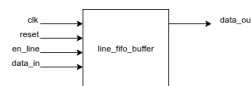
Na slici 6(b) prikazan je intefejs memorija koje se koriste za skladištenje bias-a kao i rezultata konvolucije. Memorija se sastoji od 1 porta za citanje i 1 porta za pisanje.



Slika 6: Interfejsi BRAM memorija

6.1.3 Linijski bafer

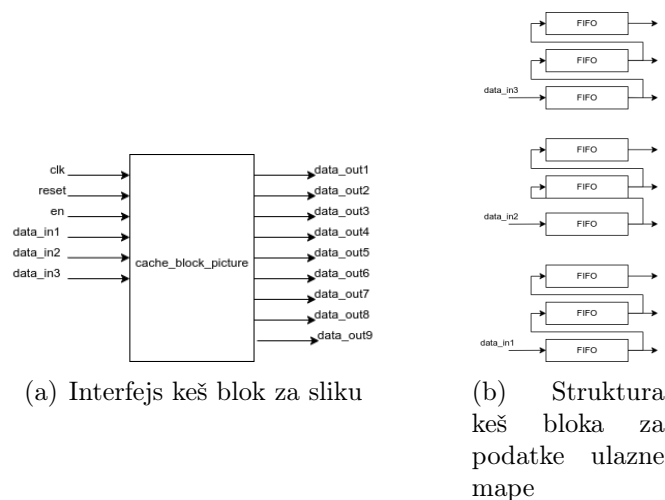
Linijski bafer je sastavni deo keš blokova koji će biti kasnije objašnjeni. Linijski bafer je ništa drugo nego FIFO bafer koji se sastoji od proizvoljnog broja elemenata(parametrizovan). Nakon što se en_line signal postavi na '1', podataka će biti upisan, i posle određenog broja taktova(u zavisnosti od dužine bafera) taj podatak će biti vidljiv na izlazu. Reset postavlja sve elemnta bafera na '0'. 7



Slika 7: Linijski bafer

6.1.4 Keš blok za sliku

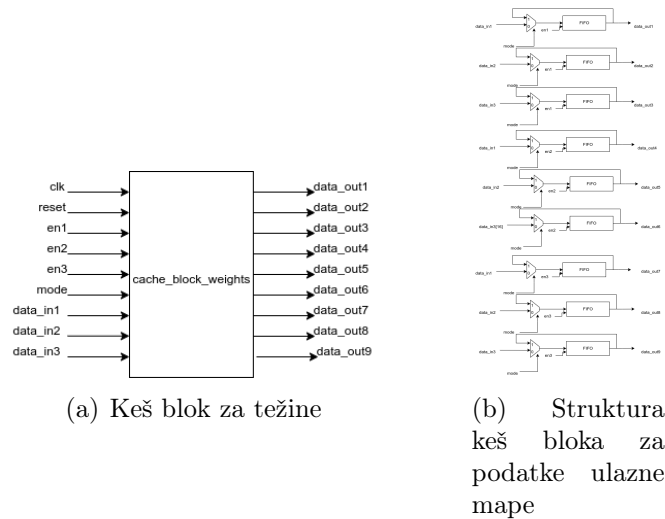
Ovaj keš blok se sastoji od 3 ulazna podatka i 9 izlaznih podataka. Memorija za sliku i težine sastoji se od 3 read porta, pa se ovim blokokm postiže paralelizam, na način da je moguće izvesti 9 umesto 3 množenja u istom taktu. Ukoliko je en signal aktivan, podaci se upisuju u keš blok. Ovaj blok se sastoji od 9 linijskih fifo bafera. Način funkcionisanja [2].



Slika 8: Keš blok za sliku

6.1.5 Keš blok za težine

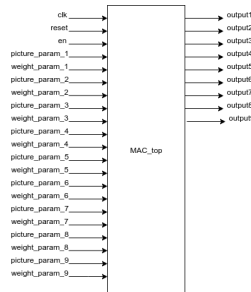
Kao i kod keš bloka za sliku, postoje 3 ulaza i 9 izlaza. Ovaj keš blok se sastoji od 9 linijskih bafera i postoje 3 enable signala, en1, en2, en3. Linijski baferi su grupisani u grupe od po 3 elemenata, i svaki od enable signala je dodeljen po jednoj grupi. Postoji i 2 moda rada ovog keš bloka. Ukoliko je mode = '0', tada podaci koji se upisuju u keš blok su podaci sa ulaznih portova data_in1, data_in2, data_in3, a ukoliko je mode = '1', podaci koji se upisuju u keš blok su podaci sa izlaza (svaki linijski bafer se ponaša kao kružni bafer). Izgled arhitekture prikazan je na slici 9(b).



Slika 9: Keš blok za težine kernela

6.1.6 MAC modul

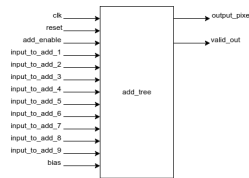
Interfejs MAC modula je prikazan na slici 10. MAC modul sadrži 9 MAC jedinica, od kojih je svaki mapiran na jedan DSP. Podaci iz keš blokova se dovode na ovaj modul i vrši se množenje i akumuliranje rezultata. U zavisnosti koji konvolucijski layer se izvršava, množenje i akumuliranje se završava za 3 ili 32 takta (u zavisnosti od broja kanala ulazne slike).



Slika 10: MAC modul

6.1.7 Modul za sabiranje

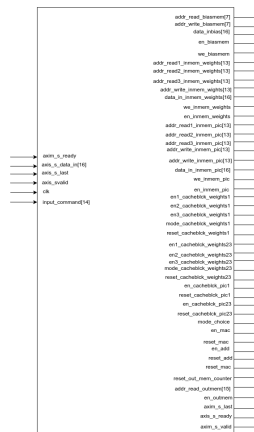
Interfejs modula sa sabiranje prikazan je na slici 11. Kada je množenje i akumuliranje gotovo, treba 9 rezultata međusobno sabrati, i dodati na njih bias koji odgovara filteru koji se koristi. Kada je `add_enable` na '1' tada se uzimaju u obzir podaci koji su na ulazu i oni se sabiraju. Ovaj modul zauzima 9 DSP na Zybo ploči, pošto je sabiranje izvedeno u obliku stabla. `Add_enable` signal se propagira do izlaza, i u taktu kada su podaci sa ulaza sabrani i pojave se na izlazu, `valid_out` se postavlja na '1'.



Slika 11: Modul za sabiranje

6.2 Controlpath

Na slici 12 prikazan je interfejs controlpath-a koji predstavlja logiku i generiše kako izlazne signale tako i signale kojim se upravlja datapath. Ulazni signali čiji naziv počinje sa `axi` kao i poslednja 3 signala predstavljaju signale koji definišu `axi stream master` i `axi stream slave` interfejsa za naš IP blok. Oni će u okviru IP integratora biti povezani na stream interfejsa DMA kontrolera. Ostatak izlaznih signala pomaže pri upravljanju datapath-a.



Slika 12: Interfejs controlpath-a

Upravljački modul ima ukupno 26 stanja. Kako se 3 konvolucijska sloja izvršavaju u hardveru, za svaki od slojeva postoje različita stanja kroz koja se prolazi sa ciljem dobijanja izlazne mape obeležja. Stanja koja postoje u okviru contropath modula :

- GLOBAL_RESET
- IDLE
- LOAD_PARAMETERS
- NEW_FILTER_0, INITIAL_LOADING_BUFFER_0, DO_CONV_0, ADD_TREE_0,
FILL_FIRST_ROW_0, FILL_SECOND_ROW_0, FILL_OTHERS_ROW_0

- **NEW_FILTER_1**, **INITIAL_LOADING_BUFFER_1**, **DO_CONV_1**, **ADD_TREE_1**, **FILL_FIRST_ROW_1**, **FILL_SECOND_ROW_1**, **FILL_OTHERS_ROW_1**
- **NEW_FILTER_2**, **INITIAL_LOADING_BUFFER_2**, **DO_CONV_2**, **ADD_TREE_2**, **FILL_FIRST_ROW_2**, **FILL_SECOND_ROW_2**, **FILL_OTHERS_ROW_2**
- **READ_OUTPUT_DATA**
- **END_COMMAND**

U stanje **GLOBAL_RESET** se ulazi samo kada se prosledi komanda za reset U stanju **IDLE** čeka se da se postavi određena komanda na ulazu, i u zavisnosti od komande prelazi u jedno od sledećih stanje:

- **GLOBAL_RESET** - iz **IDLE** stanja u ovo stanje se prelazi ako je ulazna komanda 11 [5]
- **LOAD_PARAMETARS** - u ovom stanju se implementira AXI STREAM Slave protokol, i u zavisnosti od komande podaci se upisuju u jednu od tri BRAM memorije. Iz **IDLE** stanja u ovo stanje se prelazi ukoliko se pošalje neka od sledećih komandi : 1, 2, 3, 5, 6, 8, 9 [5]
- **NEW_FILTER_0** - kada se pošalje komanda za startovanje prvog kovnolucionog sloja, komanda 4 [5], automat prelazi u ovo stanje.
- **NEW_FILTER_1** - kada se pošalje komanda za startovanje drugo kovnolucionog sloja, komanda 7 [5], automat prelazi u ovo stanje.
- **NEW_FILTER_2** - kada se pošalje komanda za startovanje trećeg kovnolucionog sloja, , komanda 10 [5], automat prelazi u ovo stanje.
- **READ_OUTPUT_DATA** - kada se cela izlazna mapa upiše u izlaznu memoriju, u ovom stanju se implementira AXI STREAM Master protokol i podaci se šalju u glavnu memoriju. Iz **IDLE** stanja u ovo stanje se prelazi ukoliko se pošalje neka od sledećih komandi : 12, 13, 14 [5]

Postoji posebno stanje pod nazivom **END_COMMAND** u kojem se aktivira prekid koji signalizira da je komanda koja se izvršavala završena. U ovo stanje se prelazi nakon svake komande, osim komande koja restartuje ceo IP blok. Iz ovog stanja se uvek prelazi u **IDLE** stanje u kome se čeka nova komanda.

Automat ostaje u stanju **LOAD_PARAMETARS** sve dok se signal *axis_s_last* ne postavi na visok nivo što signalizira da je poslednji podatak postavljen na magistrali podataka. Kako više komandi dovode automat u ovo stanje, u zavisnosti koja je trenutna komanda, različiti signali se aktiviraju čime se kontroliše upis u različite BRAM memoriju u modulu za obradu podataka. Ukoliko je poslata komanda 1, aktiviraju se signali koji dozvoljavaju upis u memoriju koje čuvaju bias podatke. U slučaju komandi 2, 5 i 8, postavljaju se signali za memorijski modul koji čuva parametre za kernel podatke. Za komande 3, 6 i 9 omogućuje se upis podataka u memorijski modul za ulaznu mapu obeležja. Brojač *addr_write_inmem* koji se koristi u ovom stanju za generisanje adresa lokacija na koju se treba upisati podaci koji pristižu. Nakon svakog validnog primljenog podatka, ovaj brojač se uvećava za 1, dok je u ostalim stanjima 0. Nakon što se upiše poslednji podatak, automat prelazi u stanje koje

signalizira da je komanda završena.

Kada je konvolucija završena, i kada se pošalje komanda za slanje podataka iz IP bloka u memoriju, iz **IDLE** stanja prelazi se u stanje **READ_OUTPUT_DATA**. Pri prelasku u ovo stanje, u zavisnosti od komande koja se šalje, registru *number_of_output_data* se dodeljuju različite vrednosti. Ako je poslata komanda 12, registru se dodeljuje vrednost 32768, za komandu 13 vrednost registra je 8192, a za komandu 14 registar dobija vrednost 4096. Ovo predstavlja broj podataka koje je potrebno poslati iz izlazne memorije nakon svakog od konvolucijskog sloja. U stanju **READ_OUTPUT_DATA** se implementira AXI STREAM Master protokol. Postavljaju se signali *axim_s_last*, *axim_s_valid*, kao i adresa sa koje se čita podatak koji je potrebno poslati. Registar *data_sent_to_out* skladišti trenutnu adresu koja sa koje se podatak čita iz BRAM memorije u kojoj je smeštena izlazna mapa obeležja, i nakon svakog pročitano podataka, on se uvećava za 1. Kada su svi podaci poslani, automat prelazi u stanje koje obeležava kraj komande.

Za sve konvolucijske slojeve koriste se sledeći registri :

- *addr_read1_inmem_pic*, *addr_read2_inmem_pic*, *addr_read3_inmem_pic* - čuvaju adrese sa kojih se čitaju podaci iz memorije za ulaznu mapu obeležja
- *addr_read1_inmem_weights*, *addr_read2_inmem_weights*, *addr_read3_inmem_weights* - čuvaju adrese sa kojih se čitaju podaci iz memorije za kernele
- *fill_buff_counter* - služi pri punjenju keš blokova za ulaznu mapu obeležja
- *channel_counter* - čuva informaciju o broju kanala nad kojima se izvršila konvolucija
- *rows_counter* - čuva se informaciju o indeksu reda nad kojim se primenjuje kernel
- *columns_counter* - čuva se informaciju o indeksu kolone nad kojim se primenjuje kernel

Nakon slanje komande za izvršavanje prvog konvolucijskog sloja, prelazi se u stanje **NEW_FILTER_0**. Ovo stanje označava da se primenjuje novi filter. U ovom stanju se restartuju sadrži keš blokova, MAC modula i pripremaju se podaci na izlazima memorije za ulaznu sliku, kako bi u narednom taktu keš blokovi mogli da se pune sa podacima. Iz ovog stanja se uvek prelazi u stanje **INITIAL_LOADING_BUFFER_0**, osim kada su svi kerneli iskorišćeni za izvršavanje konvolucije.

Iz **NEW_FILTER_0** prelazi se u stanje **INITIAL_LOADING_BUFFER_0** u koje se inicijalno pune keš blokovi za sliku i kernele. Postavljaju se signali dozvole za memorije kako bi se omogućilo čitanje iz njih, kao i signali dozvole koji omogućavaju punjenje keš blokova. Postavlja se mod za keš blok koji se odnosi na kernele, kojim se signalizira da se keš blok puni podacima iz BRAM memorije. Generišu se i adrese sa kojih se podaci iz memorija žele učitati u keš blokove. Pošto keš blokovi sadrže FIFO bafere dubine 3, potrebno je ukupno 9 taktova kako bi se inicijalno napunili. Stoga registru *fill_buff_counter* se uvećava vrednost svakog takta, dok ne dostigne vrednost 9. Tada se prelazi u stanje **DO_CONV_0**.

Podaci unutar keš blokova su pripremljeni i nastupa trenutak za izvođenje operacija množenja i akumulacije (MAC operacije). U stanju označenom kao **DO_CONV_0**, MAC moduli počinju sa radom. Paralelno sa tim, keš blok se dopunjuje novim podacima, tako da nakon

završetka obrade svih kanala za trenutni segment ulazne slike, podaci na izlazu keš bloka su odmah spremni za primenu konvolucija na naredni deo ulazne slike. Adrese sa kojih se čitaju podaci iz memorije za ulaznu mapu obeležja zavise od trenutnog reda u kojem se konvolucija izvršava. Takođe, mod rada keš bloka za kernele se menja, pri čemu sada podaci cirkulišu unutar FIFO bafera unutar samih keš blokova. Nakon što se izvrše MAC operacija svih kanala za ovaj sloj, generiše se adresa za čitanje bias podataka i prelaze u stanje u kojem je potrebno sabrati sve akumulirane podatke.

U stanju **ADD_TREE_0**, signal nazvan *enable_add* se aktivira kako bi se omogućilo sabiranje unutar modula za obrade podataka. Istovremeno, registar koji prati informacije o trenutnoj koloni se povećava. Ukoliko obrada nije stigla do kraja reda unutar ulazne mape obeležja, sledeće stanje postaje **DO_CONV_0**. Ova dva stanja se smenjuju sve dok se ne završi obrada čitavog reda. Kada se dođe do kraja reda, iz ovog stanja postoje 4 opcije za naredno stanje:

- **NEW_FILTER_0** - ukoliko su se svi redovi izvršili, prelazi se na novi filter
- **FILL_FIRST_ROW_0** - ukoliko se završilo izvršavanje nad prvim redom i prelazi se u drugi red
- **FILL_SECOND_ROW_0** - ukoliko se završilo izvršavanje nad drugim redom i prelazi se u treći red
- **FILL_OTHERS_ROW_0** - u ostalim slučajevima pri prelasku iz jednog reda u drugi

U ovom stanju je potrebno postaviti početne adrese sa kojih se želi učítavati novi redovi u keš blok, i to zavisi od trenutnog reda izvršavanja. Keš blokovi se inicijalno popunjavaju jer se prešlo na početak novog reda, pa trenutni podaci u bloku nisu korisni. Zbog specifičnog rasporeda podataka unutar memorija, koji zazuzima manje resursa pri generisanju naredne adrese, postoje tri različita načina generisanja adresa za čitanje podataka iz ulazne mape obeležja.

U svakom od ovih stanja, početno punjenje keš blokova zahteva 9 taktova, što se podudara sa taktovima u stanju **INITIAL_LOADING_BUFFER_0**. Glavna razlika je što u ovim stanjima keš blokovi za kernele neće biti punjeni. Kada se ovaj proces završi, sistem se vraća u stanje **DO_CONV_0**. Ovaj ciklus se ponavlja sve dok se ne obavi obrada svih redova unutar ulazne mape obeležja. Nakon toga se sistem vraća u stanje **NEW_FILTER_0**, nakon čega se ponovo inicijalno pune keš blokovi, ali ovog puta se učítava novi kernel. Kada se primene svi kerneli na ulaznu mapu obeležja, automat prelazi u stanje koje označava kraj komande i prekidom se signalizira kraj konvolucije.

Za izvođenje konvolucije u drugom i trećem konvolucijskom sloju, logika je identična. Za svaki od ovih slojeva, primenjuju se ista stanja, pri čemu se svako stanje označava indeksom koji se odnosi na određeni sloj : **NEW_FILTER**, **INITIAL_LOADING_BUFFER**, **DO_CONV**, **ADD_TREE**, **FILL_FIRST_ROW**, **FILL_SECOND_ROW**, **FILL_OTHERS_ROW**. Zbog različitosti u dimenzijama ulaznih mapa obeležja i broja filtera za svaki konvolucijski sloj, svako stanje je prilagođeno datom sloju. Neke od razlika koje postoje:

- broj kernela koji se primenjuje na ulaznu mapu obeležja, kao i njegov broj kanala
- dimenzije ulazne mape obeležja

- generisanje adresa kojim se čitaju podaci iz BRAM memorija sa ciljem inicijalnog punjenja ili dopunjavnja keš blokova

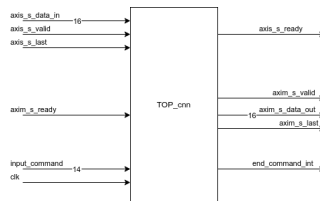
6.3 TOP module

Na slici 13 prikazan je interfejs top modula. Postoje 3 interfejsa pomoću kojih se pristupa ovom modulu. Pomoću axi-lite interfejsa se postavlja komanda koju je potrebno izvršiti (input_command). Postoji izlazni signal end_command_int koji signalizira procesoru da je komanda koja se izvršavala, završena. Za potrebe verifikacije potrebno je postaviti komandu 0 odmah nakon što se završi prethodna komanda.

Postoji AXI STREAM SLAVE interfejs (signali axis_s_data_in, axis_s_valid, axis_s_last, axis_s_ready) pomoću kojeg se prihvataju podaci koji stižu iz memorije preko DMA.

AXI STREAM MASTER interfejs (signali axim_s_data_in, axim_s_valid, axim_s_last, axim_s_ready) služi za slanje podataka iz IP u memoriju preko DMA.

Redosled kojim treba da se upravlja IP blokom je dat u sekciji 5.



Slika 13: Interfejs TOP modula

6.4 Zauzetost resursa i procenjena frekvencija rada

Na slici 14 prikazana je zauzetost svih resursa na ZYBO razvojnoj platformi u okviru PL dela. Broj resursa je očekivan.

Ukupna utrošenost BRAM-ova je 54.5. BRAM memorije za ulaznu mapu obeležja i kernele imaju po 1 port za upis i 3 porta za čitanje za punjenje keš blokova. Ulazna mapa obeležja se sastoji od 16,384 16-bitne lokacije, mapirane na 24 BRAM memorije. Memorija za kernel parametre ima 8,192 16-bitne lokacije, mapirane na 12 BRAM memorija.

Memorije za bias podatke i izlaznu mapu obeležja imaju po 1 port za čitanje i pisanje. Bias memorija sadrži 128 16-bitnih lokacija, mapiranih na pola BRAM memorije. Izlazna memorija sadrži 32,768 16-bitnih lokacija, mapiranih na 16 BRAM memorija. Ukupna iskorišćenost BRAM memorija iznosi 52.5, pri čemu se koristi 60 dostupnih jedinica. Dodatna dve BRAM memorije se instanciraju za DMA komponentu.

U okviru MAC modula, instancirano je 9 DSP komponenti koje izvršavaju MAC operacije. Takođe i 9 sabirača se nalazi u okviru modula za sabiranje koji je implementiran u obliku stabla i sabira 10 16-bitnih podataka. Iz toga zaključujemo da je iskorišćenost DSP komponenti očekivana i iznosi 18.

Procenjena frekvencija je bila 150 MHz. Nakon pokretanja implementacije u Vivado alatu, realna frekvencija sistema je 125 MHz.

7 Funkcionalna verifikacija hardvera

Funkcionalna verifikacija ima za cilj da utvrdi da li projektovani sistem radi kako je definisano po funkcionalnoj specifikaciji.

Da bi se dokazala ispravnost projektovanog DUT-a (*eng. Design Under Test*) neophodno je da na njegove ulaze dovedemo signale kojima ćemo ga testirati, posmatramo kakav odziv daje DUT za priložene signale.

UVM (*eng. Universal Verification Methodology*) je metodologija koja će biti korišćena za testiranje projektovanog sistema, realizovana pomoću sistem verilog (*system verilog*) jezika.

7.1 Verifikacioni plan

Pre početka verifikovanja dizajna neophodno je razviti temeljan plan kojim će se tim voditi tokom testiranja DUT-a, to je živ dokument koji se često menja i kreira se na osnovu funkcionalne specifikacije u cilju ispitivanja realizovanog dizajna.

Verifikacioni plan može se podeliti u tri grupe:

1. U ovu grupu spadaju funkcionalnosti dizajna koje se moraju verifikovati:
 - 1.1. Osnovni protokol(*axi_lite*, *axi_stream*) na ulaznom i izlaznom portu.
 - 1.2. Osnovne operacije svake komande koja je navedena
2. Kompleksniji scenariji koji se moraju verifikovati:
 - 2.1. Da li dizajn ignoriše ulazne podatke sve dok oni nisu validni.
 - 2.2. Da li se podaci *bias*, *weights* i *image* dobro smeštaju u memoriju.
 - 2.3. Da li je vrednost registra *filter_counter_reg* = 32 nakon prve konvolucije
 - 2.4. Da li je vrednost registra *filter_counter_reg* = 32 nakon druge konvolucije
 - 2.5. Da li je vrednost registra *filter_counter_reg* = 64 nakon treće konvolucije
 - 2.6. Vrednosti koje su upisane u registar *SLV_REG0* nakon sve tri konvolucije
 - 2.7. Da li se posle svake komande, registar *SLV_REG0* postavlja na vrednost nula
 - 2.8. Koliko je odstupanje proračuna između stvarne i očekivane vrednosti.
- 3 Klasične provere koje se nalaze u većini verifikacionih planova:
 - 3.1. Da li dizajn ispravno reaguje na komandu *RESET IP*.
 - 3.2. Da li dizajn blokira slanje podataka na izlaz ukoliko *axim_s_ready* nije 1.
 - 3.3. Da li se dobro učitavaju vrednosti iz fajlova u *base_seq_slave* i u *scoreboard-u*.
 - 3.4. Da li se na izlazu nakon svake konvolucije generiše očekivana veličina podatka.

7.2 Struktura verifikacionog okruženja

Kao što je prikazano na slici 15 verifikaciono okruženje sastoji se od : *top modul*, *test*, *configuration*, *environment*, *sequences*, *scoreboard*, *agent_axi_lite*, *agent_axi_stream_master* i *agent_axi_stream_slave*.

U nastavku svaka komponenta biće detaljnije opisana.

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	54	5	0	60
RAMB36/FIFO*	54	0	60	90.00
RAMB36E1 only	54	0	60	90.00
RAMB18	1	0	120	0.83
RAMB18E1 only	1	0	120	0.83

(a) Zauzetost BRAM-ova

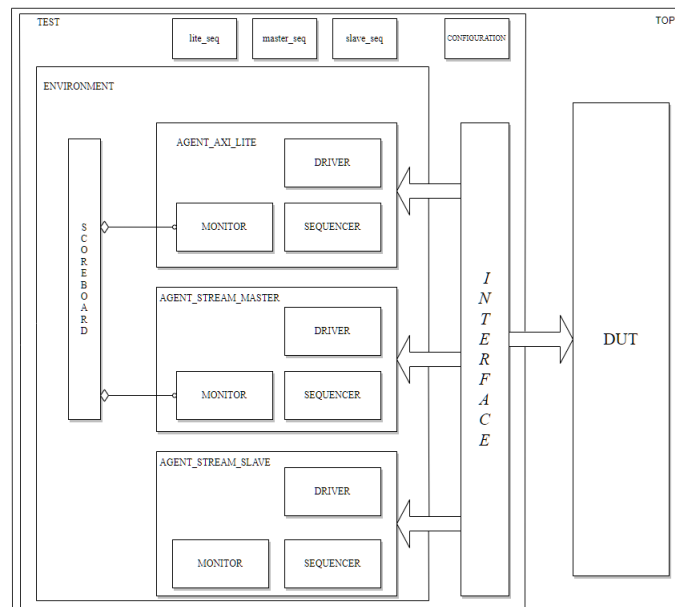
Site Type	Used	Fixed	Available	Util%
DSPs	18	0	80	22.50
DSP48E1 only	18	0	80	22.50

(b) Zauzetost DSP-jeva

Site Type	Used	Fixed	Available	Util%
Slice	1831	0	4400	41.61
SLICEL	1186	0		
SLICEM	643	0		
LUT as Logic	3681	0	17600	20.91
using O5 output only	1			
using O6 output only	2328			
using O5 and O6	1352			
LUT as Memory	514	0	6000	8.57
LUT as Distributed RAM	18	0		
using O5 output only	0			
using O6 output only	2			
using O5 and O6	16			
LUT as Shift Register	496	0		
using O5 output only	0			
using O6 output only	420			
using O5 and O6	76			
Slice Registers	5470	0	35200	15.54
Register driven from within the Slice	3019			
Register driven from outside the Slice	2451			
LUT in front of the register is unused	1388			
LUT in front of the register is used	463			
Unique Control Sets	226		4400	5.14

(c) Zauzetost LUT-ova

Slika 14: Zauzetost pojedinih resursa



Slika 15: Struktura verifikacionog okruženja

7.2.1 Top modul, test, environment i configuration

Top modul predstavlja najviši hijerarhijski nivo unutar kog se instancira interfejs i DUT čiji portovi su povezani sa interfejsom, pokreće se test i generiše taktni signal clk.

Interfejs predstavlja vezu između UVM testbenča i DUT-a.

U **testu** biramo koje sekvence ćemo puštati na sekvencere.

Unutar *test_simple1* postavljamo komande i generišemo sekvence koje će biti poslate ka DUT-u kako bi se testirao prvi konvolucionni layer, isto važi i za *test_simple2* i *test_simple3*, jedina razlika je što se generišu druge sekvence i šalju druge komande ukoliko se testira drugi, odnosno treći konvolucionni layer.

Pomoću **environment** grupišemo sve agente u jednu celinu, ideja grupisanja je mogućnost ponovnog korišćenja.

Konfiguraciona klasa koristi se za inicijalno konfigurisanje kako će verifikaciono okruženje da se ponaša, u našem slučaju u njoj postavljamo da li je agent **aktivan** ili **pasivan**.

Agent je aktivan kada se sastoji od monitora, drajvera i sekvencera, ukoliko se sastoji samo od monitora onda je pasivan.

7.2.2 Agent_axi_lite

Agent_axi_lite je aktivan i sastoji se od komponente monitor, drajver i sekvencer:

1. **Monitor_axi_lite** nagleda ulazno/izlazne portove *axi_lite* interfejsa i prosleđuje scoreboardu koja komanda je poslata
2. **Sequencer_axi_lite** sinhronizuje komunikaciju između *monitor_axi_lite* i *sequence_axi_lite*.
3. **Driver_axi_lite** prima podatke od sekvencera u vidu *sequence_item_lite* i postavlja ih na ulaze DUTa.

7.2.3 Agent_axi_stream_master

Ovaj monitor je takođe aktivan i sastoji se od istih komponenti kao prethodni:

1. **Monitor_axi_stream_master** nagleda ulazno/izlazne portove *axi_stream_master* interfejsa i skladišti rezultate sa vif.axim_s_data koje šalje scoreboard-u
2. **Sequencer_axi_stream_master** sinhronizuje komunikaciju između *monitor_axi_stream_master* i *sequence_axi_stream_master*.
3. **Driver_axi_stream_master** prima podatke od sekvencera u vidu *sequence_item_master* i postavlja ih na ulaze DUTa.

7.2.4 Agent_axi_stream_slave

Kao i prethodna dva monitora, takođe je aktivan i sastoji se od istih komponenti:

1. **Monitor_axi_stream_slave** nagleda ulazno/izlazne portove *axi_stream_slave* interfejsa.
2. **Sequencer_axi_stream_slave** sinhronizuje komunikaciju između *monitor_axi_stream_slave* i *sequence_axi_stream_slave*.
3. **Driver_axi_stream_slave** prima podatke od sekvencera u vidu *sequence_item_slave* i postavlja ih na ulaze DUTa

7.2.5 Sequences

Simple_seq_lite prima komandu *command_global* iz testa koja će dalje biti poslata preko sekvencera drajveru.

Simple_seq_master postavlja *item.axim_s_ready* na vrednost 1.

Simple_seq_slave nasleđuje klasu *base_seq_slave*, u njoj se učitavaju vrednosti iz fajlova u redove "bias", "image" i "weights", zatim u *simple_seq_slave* dodeljujemo te vrednosti iz nizova *item.axis_s_data_in*.

7.2.6 Scoreboard

Uloga scoreboard komponente je da poredi rezultat na izlazu DUT-a sa očekivanom vrednošću.

Unutar *scoreboard-a* može biti realizovan **referentni model** u kojem će biti reimplemen-tirana funkcionalnost DUT-a ili da se pre početka simulacije van UVM okruženja generišu unapred poznate vrednost koje će se porediti sa stvarnim vrednostima, taj koncept zove se **zlatni vektori**

U ovom slučaju *scoreboard* je realizovan pomoću zlatnih vektora, prethodno izgenerisane vrednosti učitavaju se iz fajlova u red *expected_image*.

U funkciji *write_axi*, scoreboard prima komandu koju mu šalje *monitor_axi_lite*.

U funkciji *write_stream*, scoreboard prima vrednosti koje mu šalje *monitor_axi_stream_master* i poredi stvarnu i očekivanu vrednost.

7.2.7 Prikupljanje pokrivenosti i regresivni testovi

Unutar sva tri monitora implementirane su grupe za pokrivanje(*eng. covergroups*), prikazane su na slici 17.



Njihova uloga je proverava da li su se dogodile sve zadate kombinacije vrednosti signala na interfejsu DUT-a.

Nije vršena provera pokrivenosti(*eng. coverage*) podataka na ulazu *axis_s_data_in*, vrednosti su unapred već poznate jer ih sami generišemo, one su proverene vizuelno u *waveform-u*.

Takođe nije vršena ni provera pokrivenosti podataka(*eng. coverage*) na izlazu *axim_s_data* jer će se ta vrednost porediti u scoreboard-u.

Pošto nije moguće jednom simulacijom(jednim testom) proveriti sve kombinacije zadate pomoću pokrivenosti, neophodno je pokrenuti više testova u nizu kako bi se sve kombinacije pojavile na interfejsu DUT-a.

Ti testovi se takođe zovu regresivni testovi(*eng. regression*), eksperimentalno je ustanovljeno da je dovoljno pokrenuti tri testa (*test_simple1*, *test_simple2*, *test_simple3*) u regresiji kako bi se ostvarila stopostotna pokrivenost, koja je prikazana na slici 16

Name 	Score 
agent_axi_lite_pkg::monitor_axi_lite::read_axi_lite	100
agent_axi_stream_master_pkg::monitor_axi_stream_master::read_axi_stream_master	100
agent_axi_stream_slave_pkg::monitor_axi_stream_slave::read_axi_stream_slave	100

Slika 16: Coverage

```

covergroup read_axi_stream_master;
  option_per_instance = 1;
  axim_s_valid: coverpoint vif.axim_s_valid{
    bins AXIM_S_VALID = {1'b1};
  }
  axim_s_last: coverpoint vif.axim_s_last{
    bins AXIM_S_LAST = {1'b1};
  }
  axim_s_ready: coverpoint vif.axim_s_ready{
    bins AXIM_S_READY = {1'b1};
  }
endgroup

covergroup read_axi_stream_slave;
  option_per_instance = 1;
  axis_s_valid: coverpoint vif.axis_s_valid{
    bins AXIS_S_VALID = {1'b1};
  }
  axis_s_last: coverpoint vif.axis_s_last{
    bins AXIS_S_LAST = {1'b1};
  }
  axis_s_ready: coverpoint vif.axis_s_ready{
    bins AXIS_S_READY = {1'b1};
  }
endgroup

covergroup read_axi_lite;
  option_per_instance = 1;
  cover_input_command: coverpoint vif.s00_axi_wdata{
    bins LOAD_BIAS_CMD = {32'h00000001};
    bins LOAD_WEIGHTS_0_CMD = {32'h00000002};
    bins LOAD_PICTURE_0_CMD = {32'h00000004};

    bins DO_CONV_0_CMD = {32'h00000008};
    bins LOAD_WEIGHTS_1_CMD = {32'h00000010};
    bins LOAD_PICTURE_1_CMD = {32'h00000020};

    bins DO_CONV_1_CMD = {32'h00000040};
    bins LOAD_WEIGHTS_2_CMD = {32'h00000080};
    bins LOAD_PICTURE_2_CMD = {32'h00000100};

    bins DO_CONV_2_CMD = {32'h00000200};

    bins RESET_CMD = {32'h00000400};

    bins SEND_OUTPUT_FROM_CONV_0_CMD = {32'h00000800};
    bins SEND_OUTPUT_FROM_CONV_1_CMD = {32'h00001000};
    bins SEND_OUTPUT_FROM_CONV_2_CMD = {32'h00002000};
  }
endgroup

```

Slika 17: Covergroups

8 Razvoj Linux drajvera

Kako bi novi uređaj bio integrisan i prepoznat od strane operativnog sistema potrebno je razviti odgovarajući drajver koji će opisivati i definisati način izvođenja bazičnih operacija sa novim hardverom i koji će omogućiti sponu između korisničkog prostora i samog uređaja. Svaki put kada se novi uređaj preko perifernog interfejsa priključi na sistem, CPU se obaveštava i registruje postojanje novog hardvera. Nekada je to prepoznatljiv uređaj (npr. miš ili tastatura), dok u slučaju namenski projektovanih uređaja poput akcelaratora opisanog u ovom dokumentu, CPU i operativni sistem ne znaju kako da ispravno komuniciraju sa tim uređajem. Prvobitno ideja Linux operativnog sistema bila je da se drajver za svaki novi uređaj treba *ručno* dodati u *source* kod kernala te ga je potrebno ponovo kompajlirati kako bi se dobila najnovija verzija OS-a. Svakodnovenom pojavom novih uređaja (posebno napretkom FPGA tehnologije), održavanje ovakvog sistema je postalo nemoguće te je uveden koncept *device tree*-a koji se i danas koristi.

8.1 Init i exit funkcije

Ovaj uređaj spada u grupu platformskih uređaja jer AXI magistrala na ARM arhitekturi ne prepoznaje ovaj uređaj kao "univerzalni", kao što je to slučaj npr sa USB ili HDMI magistralama. S toga je potrebno *bootloaderu* i samom sistemu dati početne informacije o uređaju kako bi se rezervisali potrebni resursi pri podizanju sistema ili prvobitnom pozivu *insmod* komande.

Uređaj će biti predstavljen sistemu u vidu dva fajla locirana u `\dev\` direktorijumu. Ovaj uređaj zauzima jedan *major* i dva *minor* broja u *device-tree* listi uređaja, pa je potrebno kreirati dva uređaja i alocirati dva *minor* broja, što se radi pozivom funkcije `al-loc_chrdev_region` u *init* funkciji. Potom je kreirana klasa koja će se koristiti za kreiranje dva fajla, pozivom funkcije `class_create`.

Dva uređaja se kreiraju pozivanjem `device_create` funkcije, u ovom slučaju jedan predstavlja razvijeni IP akcelerator za konvoluciju, a drugi DMA. Ovim uređajima će se kontrolisati pomoću *AXI Lite* interfejsa preko *interconnect* komponente. Kompletan *cdev* struktura je na kraju registrovana u sistemu pozivom `cdev_alloc` funkcije, linkovanjem polja *file operations* struktura, i `cdev_add` funkcijom. Pošto uređaj koristi DMA za prenos velike količine podataka, potrebno je alocirati (rezervisati) potreban broj bajtova na uzastopnim adresama fizičke memorije kako bi DMA u direktnom modu znala odakle da čita ili gde da upisuje podatke. Konačno, uređaj se registruje pozivom `platform_driver_register`, koja u stablu uređaja pronalazi uređaje koji imaju isti string u polju *compatible* kao i struktura platform-

skog uređaja opisanog u drajveru.

```
static struct of_device_id cnn_of_match[] = {
    { .compatible = "cnn_ip", },
    { .compatible = "dma", },
    { /* end of list */ },
};
```

Prilikom izrade projekta u alatu PetaLinux, alat na osnovu `.xsa` fajla generiše generičke nazive za uređaje iz Vivado alata, koji su smešteni u `.dts` fajl pre *buildovanja* stabla uređaja. Ove generičke nazive je potrebno izmeniti u navedene stringove koristeći `.dtsi` fajl koji je neka vrsta *overlaya* za već postojeći *device-tree*. Nakon generisanja stabla uređaja za kernel u alatu PetaLinux, kreira se `.dtb` fajl koji nije u formatu čitljivom za čoveka. Ukoliko se ovaj fajl dekompileira koristeći `dtc` alat, moguće je proveriti da su se vrednosti polja *compatible* struktura koje predstavljaju uređaje DMA i IP jezgro promenile.

Exit funkcija se poziva pri isključenju uređaja ili pozivom *rmmod* komande drajvera, koja sve navedene operacije izvršava inverzno (briše iz sistema).

8.2 Probe i remove funkcije

Probe funkcija se poziva pri prepoznavanju uređaja koji sadrži isto *compatible* polje kao što je navedeno u modulu. U ovom slučaju ona će biti pozvana dva puta - jednom za DMA i jednom za IP jezgro. Realizovana je tako da se prvo proverava za koji od dva uređaja se poziva (čita se *compatible* polje platformskog uređaja koji ju poziva), te pomoću *switch* petlje se obavlja potrebne operacije za svaki od uređaja. Za oba uređaja se alokira prostor fizičke memorije i vrši se mapiranje fizičke memorije na virtualne memorijske lokacije. Pored toga, oba uređaja imaju prekide (eng. *interrupt*) signale kojima obaveštavaju CPU da je pokrenuta operacija u potpunosti izvršena. Pojednostosti ovih prekidnih signala su već registrovane u stablu uređaja sistema i potekle su iz `.xsa` fajla. Da bi se *interrupt* linije registrovale i kod CPU-a potrebno je rezervisati brojeve prekida, a zatim i zauzeti prekidnu liniju sa alociranim prekidnim brojem.

```
struct cnn_info
{
    unsigned long mem_start;
    unsigned long mem_end;
    void __iomem *base_addr;
    int irq_num;
};
```

Struktura koja predstavlja *char device* fajlove iz tog razloga ima dodatno polje *irq_num*. Ovo polje se koristi pri alociranju prekidnog broja.

AXI DMA uređaj se sastoji od dva kanala: MM2S (eng. *AXI4 memory-mapped to stream*) i S2MM (eng. *AXI stream to memory-mapped*). Oba kanala poseduju svoje interrupt linije te je za ovaj uređaj potrebno zauzeti dva *IRQ* broja. Kod IP jezgra to je dovoljno uraditi samo jedan put pošto on ima samo jedan interrupt sa nazivom *interrupt_done*. U nastavku je prikazan primer procedure za CNN IP. Pozivom metode *request_irq* prethodno alocirani broj prekida (njen prvi parametar) povezujemo sa prekidnom rutinom (eng. *Interrupt Service Routine, ISR*) koja će se izvršiti svaki put kada se aktivira prekid (drugi parametar). Kao treći parametar specificiraju se *flagovi*, u ovom slučaju to je samo jedan koji govori da je prekid

ostljiv na rastuću ivicu signala. Poslednja dva parametra prosleđuju osnovne informacije o uređaju koji alocira prekid. Kao poslednji korak poziva se metoda *enable_irq* koja osigurava da je obrada prekidne rutine dozvoljena.

```
/* Get irq number */
cnn_p->irq_num = platform_get_irq(pdev, 0);
if(!cnn_p->irq_num)
{
    printk(KERN_ERR "[cnn_probe] Could not get IRQ resource\n");
    rc = -ENODEV;
    goto error2;
}

if(request_irq(cnn_p->irq_num, cnn_isr, IRQF_TRIGGER_RISING, DEVICE_NAME,
    cnn_p))
{
    printk(KERN_ERR "[cnn_probe] Could not register IRQ %d\n", cnn_p->irq_num);
    return -EIO;
    goto error3;
}
else {
    printk(KERN_INFO "[cnn_probe] Registered IRQ %d\n", cnn_p->irq_num);
}

enable_irq(cnn_p->irq_num);
```

8.3 Read, write i mmap funkcije

Specifične funkcije za svaki drajver uređaja jesu funkcije za upis ili čitanje podataka iz uređaja. Kako je još u ESL fazi odlučeno da će se za prenos podataka koristiti DMA, iz korisničke aplikacije i *user-space*-a podaci se prosleđuju koristeći *mmap* i *memcpy* metode. Da bi one bile moguće potrebno je definisati *mmap* metodu u drajveru uređaja i pozivom *dma_mmap_coherent* metode mapirati virtuelne memorijske lokacije na fizičke adrese u memoriji, kako bi DMA uređaj čitao ili upisivao podatke na unapred određena i alocirana mesta u memoriji.

Read funkcija kao takva s toga nema ulogu jer naš uređaj ima samo dva vida povratne informacije - informacija da je završio obradu podataka koju javlja preko interrupta, i informacija koja sadrži veliki broj podataka u vidu obrađenih piksela slike, koja će se upisom ispravne komande u kontrolni registar pročitati iz uređaja preko DMA u fizičku memoriju i odatle memorijski mapirati u korisnički prostor. S toga je read funkcija definisana, ali pokušajem *cat* metode na neki od dva uređaja dobija se samo poruka upozorenja da čitanje nije validno.

Write funkcija ima smisla za oba uređaja, međutim da bi izbegli dvostruko pozivanje funkcija koje po prirodi uređaja treba zajedno da se pozivaju i olakšali način rada drajvera, nije omogućen upis u DMA uređaj. Moguće je upisivati samo u CNN uređaj validnu komandu u njegov kontrolni registar. Nakon provere koja komanda se poziva, ukoliko ona uključuje posredstvo DMA komponente, u istom koraku će se izvršiti i upis u DMA kontrolni registar i započinje transkacija. Na ovaj način krajnji korisnik ne treba da vodi računa o DMA komponenti i ispravnom upravljanju tim uređajem, već samo treba da zada validnu komandu

IP jezgu.

Komanda koju prosleđuje korisnik se čita pomoću standardnih `copy_from_user` i `sscanf` metoda, i nakon provere validnosti komande (nije dozvoljeno upisati bilo kakvu komandu koja nije opisana u sekciji 5), upis u kontrolni registar se vrši na sledeći način:

```
iowrite32((u32)input_command, cnn_p->base_addr);
```

Za pokretanje DMA transakcije koriste se pomoćne metode `dma_simple_write` za upis (MM2S kanal) i `dma_simple_read` za čitanje (S2MM kanal) koje konfigurišu kontrolne registre AXI DMA kanala.

Iz korisničke aplikacije, pre slanja bilo koje komande koja podrazumeva slanje podataka u ili iz uređaja posredstvom DMA komponente, potrebno je izvršiti memorijsko mapiranje koje mapira memorijske lokacije virtualne memorije iz korisničkog prostora na fizičke memorijske lokacije kernel prostora sa kojima radi DMA uređaj.

```
int fd = open("/dev/dma", O_RDWR | O_NDELAY);
p = (int*)mmap(0, 128*2, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
memcpy(p, input_bias, 128*2);
munmap(p, 128*2);
close(fd);
```

Na prethodnom listingu prikazan je način memorijskog mapiranja niza od 128 16-bitnih elemenata koji predstavljaju *biase* mreže i koje je po protokolu potrebno poslati pre klasifikacije, jednom na samom početku rada aplikacije. Kako bi dobili *handle* fajla DMA potrebno je otvoriti fajl uređaja sa sa kojim je asocirana DMA memorija i adrese virtuelnih i fizičkih memorijskih nizova u drajveru. Pozivom `mmap` metode vrši se memorijsko mapiranje tako što pokazivač `p` dobija vrednosti početne fizičke adrese niza asociranog sa DMA uređajem. Funkcija `memcpy` vrši kopiranje sadržaja niza `input_bias` koji se nalazi u korisničkom prostoru na fizičke memorijske lokacije počevši od adrese na koji pokazuje pokazivač `p`, pa sve do kraja niza trajanja navedenog kao treći parametar (u ovom slučaju 128 elemenata koji su 16-bitni, ukupno 256 bajtova). Nakon kopiranja, potrebno je *odmapirati* memoriju pozivom `munmap` metode i zatvoriti fajl. Nakon ovog kodnog segmenta uspešno su kopirani podaci iz korisničkog prostora u kernel prostor, a samim tim i na fizičke lokacije memorije. Pozivom već pomenute `dma_simple_write` metode u drajveru DMA će čitati i slati ispravne podatke prosleđene iz korisničkog prostora.

8.4 Prekidne rutine

Već je pomenuto da su za uređaj registrovane tri prekidne linije, 2 za DMA (po jedna za svaki kanal) i jedna za IP akcelerator. Ove prekidne rutine se izvršavaju svaki put kada se operacija povezana sa uređajem izvrši uspešno u potpunosti.

Prateći *Xilinxovu* dokumentaciju vezanu za DMA blok, pri pozivu prekidne rutine za neko od kanala vrši se resetovanje odgovarajućeg bita u statusnom registru uređaja. Po specifikaciji to nije potrebno raditi za IP akcelerator jer će on svoj *interrupt* signal držati jedan takt na visokom nivou i potom ga sam resetovati na niski logički nivo. Prekidne rutine su iskorištene tako da se pri svakom pozivu *write* funkcije drajvera dve globalne promenjive postavljaju na vrednost 1, i resetuju se na vrednost 0 unutar prekidnih rutina. Drajver je zamišljen tako da čeka da se vrednosti ovih promenjivih spuste na 0 i tek tada kontrolu predaje nazad korisničkoj aplikaciji, garantujući da je pozvana operacija u potpunosti izvršena. U narednom kodnom

segmentu prikazan je primer prekidne rutine IP akceleratora, gde je linija sa operacijom ipisa zakomentarisana kako bi se uštedelo vreme izvršavanja prekidne rutine.

```
static irqreturn_t cnn_isr(int irq, void* dev_id)
{
    ip_process_over = 0;
    // printk(KERN_INFO "[cnn_isr] IP finished operation. CNN Interrupt handled\n");
    return IRQ_HANDLED;
}
```

9 Zaključak

Izradom ovog projekta ispunjeni su svi projektni zadaci i kao proizvod dobio se uspešan i kompletan FPGA akcelerator za namensku konvolucionu neuronsku mrežu.

U fazi projektovanja uređaja na sistemskom nivou uspešno je konstruisana mreža koristeći već postojeće biblioteke u programskom jeziku *python* i koristeći tu mrežu u fazi treniranja dobijeni su parametri mreže u vidu *weightova* i *biasa* za filtre. Projektovanje uređaja je započelo prevođenjem *python* koda koji je koristio postojeće biblioteke, u *C++* program koji nije koristio biblioteke za propagaciju slike kroz mrežu. Taj model je zatim preveden u *System C* model koji je razlikovao različite komponente u sistemu koje su međusobno u toku simulacije komunicirale putem *eventova*. Spuštanjem nivoa hijerarhije modela razvijen je *TLM* model u kojem je jasno utvrđena veza između pojedinih podsistema u uređaju i način njihovog komuniciranja. Konačno, razvijen je *TLM LT* model na osnovu kojeg je razvijen algoritam koji je implementiran na *RTL* nivou.

Kao završni korak razvijen je drajver modul za operativni sistem Linux koji služi kao spona između korisničke aplikacije i uređaja. Drajver je projektovan i realizovan tako da krajnjem korisniku što više olakša korišćenje uređaja, pa tako korisnik ne treba da brine na upravljanje DMA komponentom. Rad drajvera se oslanja na prekidnim rutinama koje komponente iz uređaja koriste da obaveste CPU da su operacije koje se su izvršavale uspešno završene, što dodatno ističe akceleraciju i nikakvo vreme se ne troši uzaludno u toku rada drajvera ili korisničke aplikacije na čekanju rezultata.

Konačno, razvijena je korisnička aplikacija koja služi kao demonstracija uspešnosti realizovanog sistema. Pri poređenju sa početnom verzijom mreže koja se u potpunosti izvršavala na procesoru ne koristeći ugrađene biblioteke, obrada i klasifikacija jedne slike se putem ovog akceleratora obrađuje oko 5 puta brže. Podaci u tabeli 3 su prikupljeni u toku izvršavanja dve verzije aplikacije: početna verzija jeste model mreže koji se izvršavao isključivo koristeći procesorske resurse, dok krajnja verzija predstavlja podatke skupljene tokom izvršavanja korisničke aplikacije koja koristi akcelerator za tri konvoluciona sloja. Podaci su prikupljeni u istim uslovima, koristeći *Zybo Z7-10* razvojno okruženje sa 667MHz *dual-core Cortex-A9* procesorom.

Uz unapređenje performansi nije došlo do degradacije tačnosti mreže, koja je zadržala gotovo istu tačnost te na testnom skupu od 10 hiljada slika ima tačnost 76.56% čime je dokazano da je bitska analiza bila tačna i uspešna.

	Početna verzija	Krajnja verzija
Conv0	541ms	17ms
Conv1	959ms	7ms
Conv2	454ms	4ms
Flatten sloj	460us	505us
Dense0 sloj	148ms	154ms
Dense1 sloj	1.5ms	1.5ms
Ukupno slika	2163ms	319ms

Tabela 3: Poređenje vremenskih performansi klasifikacije slike početnog modela i krajnjeg uređaja

Literatura

- [1] <https://streamable.com/oz8ai6>
- [2] <https://www.elektronika.ftn.uns.ac.rs/funkcionalna-verifikacija-hardvera/wp-content/uploads/sites/89/2018/03/vezba10-1.pdf>