UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA

Marko Nikić

Funkcionalna verifikacija MLP IP jezgra za prepoznavanje cifara

DIPLOMSKI RAD
- Osnovne akademske studije –

Novi Sad, 2020

	UNIVERZITET U NOVOM SADU ● **FAKULTET TEHNIČKIH NAUKA** 21000 NOVI SAD, Trg Dositeja Obradovića 6	Datum:
	ZADATAK ZA IZRADU ZAVRŠNOG (BACHELOR) RADA	List/Listova: 1/1

(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:	**Marko Nikić**	Broj indeksa:	**EE86-2015**
Oblast	**Funkcionalna verifikacija hardvera**		
Mentor:	**dr Rastislav Struharik**		

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

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

NASLOV DIPLOMSKOG (BACHELOR) RADA:

Funkcionalna verifikacija MLP IP jezgra za prepoznavanje cifara

TEKST ZADATKA:

1. Teorijski uvod u funkcionalnu verifikaciju hardvera
2. Izrada verifikacionog plana za verifikovanje MLP IP jezgra
3. izrada verifikacionog okruženja
4. Diskusija verifikacionih rezultata i pokrivenosti

Rukovodilac studijskog programa:	Mentor rada:
dr Milan Sečujski	dr Rastislav Struharik

Primerak za: ☐ - Studenta; ☐ - Mentora

UNIVERZITET U NOVOM SADU ● **FAKULTET TEHNIČKIH NAUKA**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

Redni broj, **RBR**:		
Identifikacioni broj, **IBR**:		
Tip dokumentacije, **TD**:	monografska publikacija	
Tip zapisa, **TZ**:	tekstualni štampani dokument	
Vrsta rada, **VR**:	diplomski rad	
Autor, **AU**:	Marko Nikić	
Mentor, **MN**:	dr Rastislav Struharik, FTN Novi Sad	
Naslov rada, **NR**:	Funkcionalna verifikacija MLP IP jezgra za prepoznavanje cifara	
Jezik publikacije, **JP**:	srpski	
Jezik izvoda, **JI**:	srpski	
Zemlja publikovanja, **ZP**:	Srbija	
Uže geografsko područje, **UGP**:	Vojvodina	
Godina, **GO**:	2020	
Izdavač, **IZ**:	autorski reprint	
Mesto i adresa, **MA**:	Novi Sad, Fakultet tehničkih nauka, Trg Dositeja Obradovića 6	
Fizički opis rada, **FO**:		
Naučna oblast, **NO**:	Elektronika	
Naučna disciplina, **ND**:	Embeded sistemi	
Predmetna odrednica / ključne reči, **PO**:	FPGA, MLP, Funkcionalna verifikacija hardvera	
UDK		
Čuva se, **ČU**:	Biblioteka Fakulteta tehničkih nauka, Trg Dositeja Obradovića 6, Novi Sad	
Važna napomena, **VN**:		
Izvod, **IZ**:	U ovom diplomskom radu izvršena je funkcionalna verifikacija hardverskog IP jezgra. IP jezgro predstavlja deo hardveskog-softverskog rešenja akceleratora višeslojnog perceptrona za prepoznavanje cifara. Akcelerator je implementiran na FPGA čipu.	
Datum prihvatanja teme, **DP**:		
Datum odbrane, **DO**:		
Članovi komisije, **KO**:		
Predsednik:		
Član:		Potpis mentora:
Mentor:	dr Rastislav Struharik, redovni profesor, FTN Novi Sad	

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:			
Identification number, **INO**:			
Document type, **DT**:		monographic publication	
Type of record, **TR**:		textual material	
Contents code, **CC**:		BSc thesis	
Author, **AU**:		Marko Nikić	
Mentor, **MN**:		dr Rastislav Struharik, FTN Novi Sad	
Title, **TI**:		Functional verification of a multilayer perceptron IP core for digit recognition	
Language of text, **LT**:		Serbian	
Language of abstract, **LA**:		Serbian	
Country of publication, **CP**:		Serbia	
Locality of publication, **LP**:		Vojvodina	
Publication year, **PY**:		2020	
Publisher, **PB**:		author's reprint	
Publication place, **PP**:		Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6	
Physical description, **PD**:			
Scientific field, **SF**:		Electrical engineering	
Scientific discipline, **ND**:		Embedded systems	
Subject / Keywords, **S/KW**:			
UDC		FPGA, MLP, Functional verification of hardware	
Holding data, **HD**:		Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad	
Note, **N**:			
Abstract, **AB**:		This graduation thesis describes functional verification of an IP core.The IP core represents the hardware-software solution for a multilayer perceptron. The accelerator is implemented on an FPGA chip.	
Accepted by sci. board on, **ASB**:			
Defended on, **DE**:			
Defense board, **DB**:			
	President:		
	Member:		Mentor's signature
	Mentor:	Rastislav Struharik, PhD, full prof., FTN Novi Sad	

Izjava o akademskoj čestitosti

Student: **Marko Nikić**
Broj indeksa: **EE86-2015**
Student **osnovnih akademskih studija**
Autor rada pod nazivom: **Funkcionalna verifikacija MLP IP jezgra za prepoznavanje cifara**

Potpisivanjem izjavljujem:

- da je rad isključivo rezultat mog sopstvenog istraživačkog rada;
- da sam rad i mišljenja drugih autora koje sam koristio u ovom radu naznačio ili citirao i navedeni u spisku literature/referenci koji su sastavni deo ovog rada;
- da sam dobio sve dozvole za korišćenje autorskog dela koji se u potpunosti/celosti unose u predati rad i da sam to jasno naveo;
- da sam svestan da je plagijat korišćenje tuđih radova u bilo kom obliku (kao citata, parafraza, slika, tabela, dijagrama, dizajna, planova, fotografija, filma, muzike, formula, veb sajtova, kompjuterskih programa i sl.) bez navođenja autora ili predstavljanje tuđih autorskih dela kao mojih, kažnjivo po zakonu (Zakon o autorskom i srodnim pravima, Službeni glasnik Republike Srbije, br. 104/2009, 99/2011, 119/2012), kao i drugih zakona i odgovarajućih akata Univerziteta u Novom Sadu;
- da sam svestan da plagijat uključuje i predstavljanje, upotrebu i distribuiranje rada predavača ili drugih studenata kao sopstvenih;
- da sam svestan posledica koje kod dokazanog plagijata mogu prouzrokovati na predati rad i moj status;
- da je elektronska verzija rada identična štampanom primerku i pristajem na njegovo objavljivanje pod uslovima propisanim aktima Univerziteta.

Novi Sad, 22.10.2020.

Potpis studenta

_____

Sadržaj

Spisak slika u tekstu

Spisak kodova u tekstu

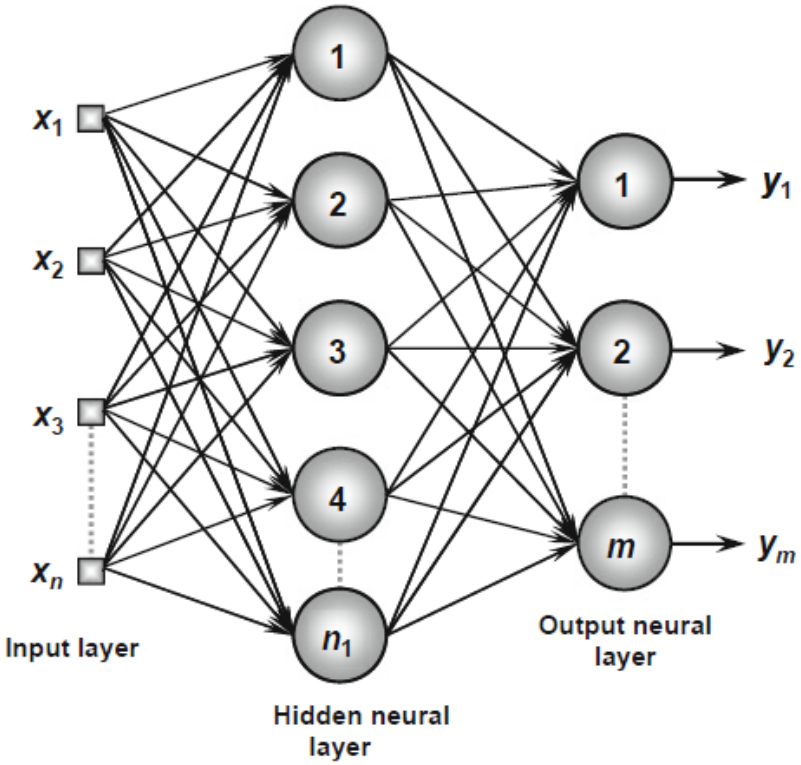
1 Uvod

Tema ovog rada je funkcionalna verifikacija IP jezgra koje implementira algoritam višeslojnog perceptrona (engl. *Multilayer Perceptron*, skraćeno MLP) za prepoznavanje cifara. Rad je deo šireg poduhvata koji obuhvata i sistemsko projektovanje, hardversku implementaciju i osposobljavanje mikroračunarskog sistema koji će podržati ovaj hardver. Rad se sastoji iz sledećih celina:

- Prvog, uvodnog poglavlja
- Drugog poglavlja, u kom su predstavljene teorijske osnove korišćene metodologije.
- Trećeg poglavlja, u kom je izložena funkcionalna specifikacija jezgra koje se verifikuje.
- Četvrtog poglavlja, koje čini verifikacioni plan.
- Petog poglavlja, u kom je dat detaljan opis verifikacionog okruženja.
- Šestog poglavlja, u kom je opisan način prikupljanja pokrivenosti.
- Sedmog poglavlja, gde je opisan tok verifikacije i iskomentarisani rezultati prikupljanja verifikacione pokrivenosti.
- Osmog, zaključnog poglavlja sa diskusijom o postignutim ciljevima.

Višeslojni perceptron je neuronska mreža koja se koristi u rešavanju čitavog spektra problema: u aproksimaciji funkcija, prepoznavanju i klasifikaciji šablona, u optimizaciji i kontroli procesa. Slika 1 pokazuje jednu ovakvu neuronsku mrežu, sastavljenu od ulaznog sloja sa n neurona, skrivenog sloja sa n_1 neurona i izlaznog sloja od m neurona koji predstavljaju izlazne vrednosti problema koji se analizira (u ovom slučaju rezultat klasifikacije). Jedna od najznačajnijih karakteristika veštačkih neuronskih mreža je njihova sposobnost da uče iz šablona. Nakon što je mreža naučila vezu između svojih ulaza i izlaza, ona može da uopštava rešenja, odnosno da proizvodi izlaze koji su bliski očekivanim, željenim izlazima za bilo koje ulazne vrednosti. Samo učenje podrazumeva proces treniranja, redosled koraka čijim izvršavanjem mreža podešava svoje parametre (težine i pragove) na osnovu prezentovanih podataka. [2]

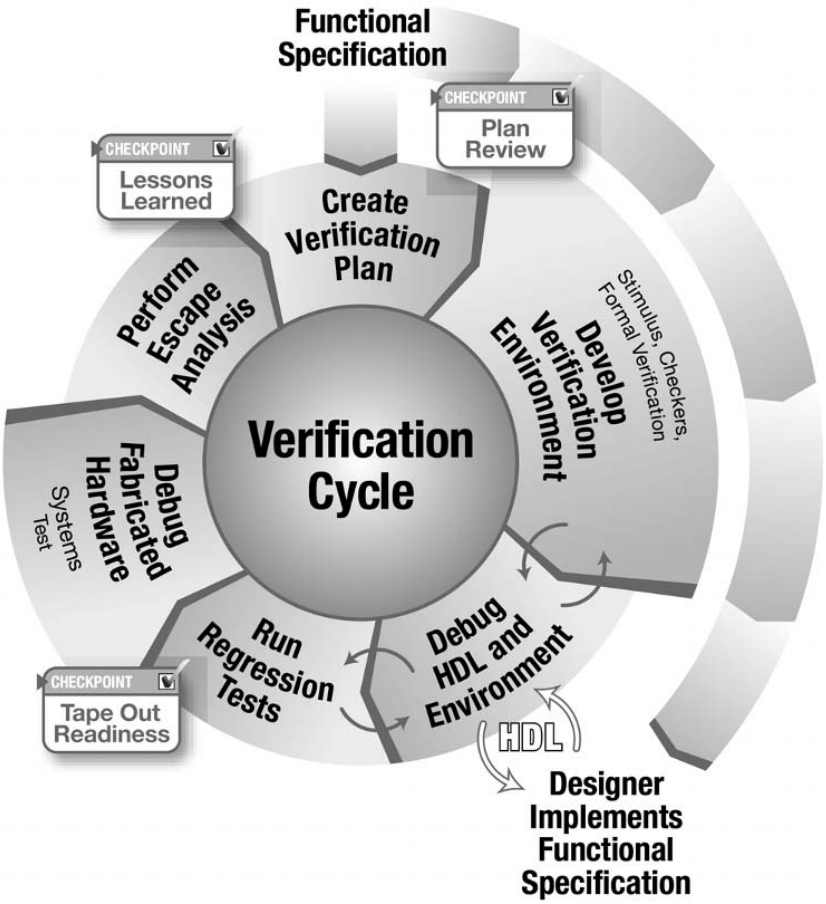
Širok spektar problema na koje se mogu primeniti, visok stepen uspešnosti u poređenju sa drugim algoritmima i jednostavna arhitektura višeslojnih perceptrona su bili odlučujući faktori za izbor baš ovog algoritma za rešavanje problema klasifikacije cifara. Hardverska akceleracija se pokazala kao zahvalno rešenje zbog relativno velikog broja računskih operacija množenja i sabiranja koje mreža vrši tokom klasifikacije, mogućnosti paralelizacije i jednostavnog mapiranja u hardver.



Slika 1: Ilustracija višeslojnog perceptrona

2 Funkcionalna verifikacija hardverskog dizajna i njena uloga

Cilj hardverskog dizajna je da napravi uređaj koji obavlja određenu funkciju na osnovu specifikacije dizajna, dok uloga funkcionalne verifikacije nije samo pronalaženje bagova već i garantovanje da dizajn obavlja svoju funkciji, odnosno funkcionalna verifikacija mora da potvrdi da je dizajn tačna reprezentacija specifikacije. Proces verifikacije je paralelan procesu pravljenja dizajna. Dizajner čita hardversku specifikaciju određenog bloka, interpretira taj opis i pravi odgovarajuću logiku na RTL nivou. Verifikacioni inženjer takođe čita hardversku specifikaciju i na osnovu nje pravi testove kojima utvrđuje da li dizajn obavlja željenu funkcionalnost. Razlog za ovakvu praksu je što svaka specifikacija može da bude dvosmislena zbog nedostatka detalja ili konfliktnih opisa, te ako više od jedne osobe analizira istu specifikaciju povećava se šansa da specifikacija dizajna bude u potpunosti implementirana. Proces verifikacije se može predstaviti verifikacionim ciklusom koji se sastoji iz sledećih koraka, predstavljenih i na slici 2 [7]:



Slika 2: Verifikacioni ciklus
[7]

- Funkcionalna specifikacija
- Pravljenje verifikacionog plana
- Razvijanje verifikacionog okruženja
- Debug HDL-a i okruženja
- Pokretanje regresije
- Debagovanje fabrikovanog hardvera
- *Escape* analiza

Verifikacioni ciklus se odvija u smeru kazaljke na satu, počevši od funkcionalne specifikacije, preko verifikacionog plana, sve do *escape* analize. Funkcionalna speci-

fikacija opisuje željeni proizvod i nju zadaje sistemski arhitekta. Ona obuhvata specifikaciju interfejsa preko koga dizajn komunicira, funkcije koje mora da izvršava i uslove koji utiču na dizajn. Ključni deo verifikacionog ciklusa je verifikacioni plan jer on daje detaljan opis verifikacionih napora. On daje odgovor na pitanja "šta verifikujem?" i "kako ću to da verifikujem?". Verifikacioni plan se sastoji od specifičnih testova i metoda, potrebnih alata, kriterijuma završetka verifikacije, resursa, funkcija koje treba da se verifikuju, funkcija koje ne moraju da se verifikuju na datom nivou hijerarhije. Nakon završetka verifikacionog plana započinje razvoj verifikacionog okruženja. Ono se sastoji iz softverskog koda, koji je uglavnom specifičan za dizajn, i alata, koji se koriste na više projekata. [7]

Naredni korak u verifikacionom ciklusu je debug dizajna i verifikacionog okruženja. U ovom koraku verifikacioni inženjer debaguje hardver pokretanjem testova definisanih u verifikacionom planu. Kako testovi odmiču, pronalaze se i analiziraju nove anomalije. Analiza otkriva uzrok anomalije koji može biti ili unutar okruženja ili unutar HDL dizajna. Ukoliko je anomalija u dizajnu, verifikacioni inženjer o tome obaveštava dizajn tim, čija je obaveza da grešku ispravi. Ukoliko je anomalija u verifikacionom okruženju, onda je verifikacioni inženjer zadužen za ispravljanje greške. Nakon završenog debaga sledi regresija, odnosno kontinualno puštanje testova definisanih u verifikacionom planu. Fabrikacija hardvera je korak koji sledi kada su zadovoljeni svi kriterijumi fabrikacije. Nakon fabrikacije sledi proces debaga, odnosno proizvod se postavlja u planirani sistem čime se testira da li će se pojaviti neka anomalija. U ovom delu se vidi najveći značaj verifikacije, a to je da se izbegnu problemi na fabrikovanom hardveru, jer oni mogu da budu veoma skupi. Ako se prilikom testiranja fabrikovanog hardvera pronađu problemi, verifikacioni tim mora da izvrši *escape* analizu kojom se utvrđuje zašto je verifikaciono okruženje previdelo taj problem. Verifikacioni tim reprodukuje problem u verifikacionom okruženju, kako bi (ako je to moguće) zaključili zašto bag nije uhvaćen u verifikacionom ciklusu. Ovo je jako bitno jer verifikacioni tim uči na svojim greškama kako u narednom verifikacionom ciklusu ne bi ponovio iste. [7]

Za verifikaciju MLP harverskog dizajna je odabrana univerzalna verifikaciona metodologija (engl. *Universal Verification Methodology*, UVM). UVM je standardizovana metodologija za funkcionalnu verifikaciju sa pomoćnom bibliotekom u SystemVerilog jeziku. Jedna od glavnih karakteristika ove metodologije je UVC (engl. *Universal Verification Component*), odnosno univerzalne verifikacione komponente koje imaju istu strukturu (sadrže monitore, drajvere, sekvencere,...). UVM obezbeđuje *framework* za verifikaciju zasnovanu na pokrivenosti koja ne zahteva kreiranje velikog broja testova, osigurava temeljnu verifikaciju na osnovu zadatih parametara i olakšava proces pronalaženja problema. [3]

3 Funkcionalna specifikacija MLP jezgra

Višeslojni perceptron (engl. *MLP - Multilayer Perceptron*) je mreža sa direktnim dejstvom (engl. *feed-forward*), bez povratne sprege, sačinjena od jednostavnih jedinica za obradu (neurona) i bar jednog skrivenog sloja. [1] Neuroni u MLP-u su potpuno povezani (engl. *fully connected*), što znači da je izlaz svakog neurona iz jednog sloja povezan na ulaz svakog neurona iz narednog sloja. Neuron ima više ulaza koje množi sa odgovarajućim težinama, kako bi sabrao sve dobijene proizvode i pripadajući *bias*. Taj zbir se propagira kroz nelinearnu funkciju aktivacije kako bi se konačno dobio izlaz neurona.

Ukoliko se uvedu oznake a_k^{l-1} za izlaz k-tog neurona iz $(l-1)$-og sloja, a_j^l sa izlaz jednog j-tog neurona iz sloja l, w_{jk}^l za težinu njihove veze i b_j^l za *bias* j-tog neurona iz sloja l, onda važi sledeća jednačina:

$$a_j^l = f(\sum_k w_{jk}^l \cdot a_k^{l-1} + b_j^l),$$

gde se sumiranje vrši po svim k neuronima iz $(l-1)$-og sloja. [5]

Korišćena funkcija aktivacije je *Leaky ReLu* data sledećom jednačinom:

$$f(x) = \begin{cases} 0,001 \cdot x, & x < 0 \\ x, & x \geq 0. \end{cases} \tag{1}$$

Na ulaz višeslojnog perceptrona se dovodi vektor koji predstavlja objekat koji se klasifikuje. Ovaj vektor se dalje množi i sabira sa unapred određenim koeficijentima i na izlazu mreže se dobija rezultat klasifikacije. Svaki od izlaznih neurona predstavlja jednu od kategorija klasifikacije: ako k-ti neuron u poslednjem sloju ima najveću izlaznu vrednost, ulazni vektor pripada k-toj kategoriji. U našem slučaju MLP ima 3 sloja: ulazni, skriveni i izlazni. Ulazni vektori su slike iz MNIST baze veličine 28x28 piksela, zbog čega ima 784 ulaznih neurona (odnosno postoje 784 komponente ulaznog vektora), a ulazi u neurone prvog sloja su vrednosti piksela od 0 do 1. U skrivenom sloju ima 30 neurona, a kako je u pitanju klasifikacija cifara od 0 do 9, u poslednjem, izlaznom sloju ima 10 neurona.

MLP IP core komunicira sa svojim okruženjem preko dva standardna AXI4 interfejsa: AXI *Lite* i AXI *Stream*. AXI *Lite* se koristi za čitanje statusnih i upisivanje u komandne registre, dok se AXI *Stream* koristi za slanje podataka potrebnih za izračunavanje (slika, težine i *bias*-i). Detalji o specifikaciji ovih interfejsa mogu se pronaći [6].

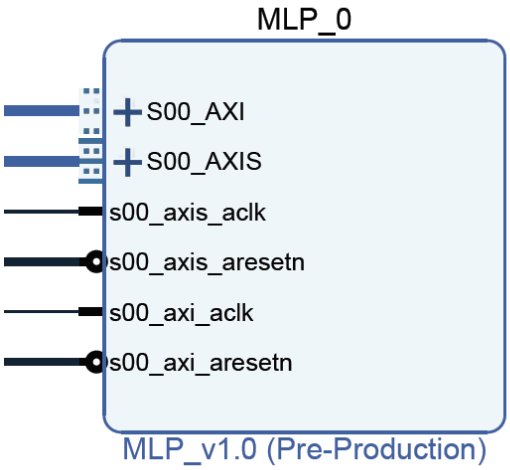
Unutrašnji registri i njihove adrese su prikazani u tabeli 1.

Interfejs MLP jezgra je prikazan na slici 3. Sa slike uočavamo:

- S00_AXI - predstavlja standardni AXI *Lite* interfejs

Tabela 1: Registri MLP IP jezgra

naziv registra	adresa pristupa AXI *Lite* interfejsom
start	0
ready	4
toggle	8
cl_num	12



Slika 3: MLP IP jezgro sa naznačenim interfejsima

- S00_AXIS - predstavlja standardni AXI *Stream* interfejs
- s00_axi_aclk - ulazni AXI *Lite* taktni signal
- s00_axis_aclk - ulazni AXI *Stream* taktni signal
- s00_axi_aresetn - ulazni AXI *Lite* reset signal
- s00_axis_aresetn - ulazni AXI *Stream* reset signal

Spremnost MLP jezgra za novu klasifikaciju se dobija čitanjem stanja unutrašnjeg *ready* registra. Ukoliko je njegova vrednost 0, jezgro je zauzeto klasifikovanjem, a ako je njegova vrednost 1, jezgro je slobodno i može se započeti nova klasifikacija. Klasifikacija se započinje postavljanjem unutrašnjeg *start* registra na 1. Nakon toga je neophodno preko AXI *Stream* interfejsa poslati prvo 28x28 piksela slike koja se klasifikuje, a potom težine i *bias* za svaki neuron u MLP-u. Slanje ovih podataka je potrebno izvršiti u tačno definisanom redosledu, kako bi jezgro znalo

da ih protumači na pravilan način: prvo se šalje svih 784 piksela slike, a potom težine i *bias* za svaki neuron pojedinačno.

Jezgro dobijene ulaze za prvi sloj skladišti u lokalnu BRAM memoriju. Kada krenu da pristižu težine, jezgro ih množi sa ulazima skladištenim u svojoj memoriji, a rezultat množenje sabira sa postojećom vrednošću u akumulatorskom registru. To čini za svaku pristiglu težinu, na kraju dodaje *bias* i tu vrednost propagira kroz funkciju aktivacije. Vrednost izlaza neurona se skladišti na prvoj narednoj slobodnoj lokaciji u memoriji, a akumulatorski registar se resetuje. Računanje vrednosti izlaza neurona iz narednog sloja vršiće se čitanjem vrednosti izlaza prethodnog sloja iz memorije i njihovim množenjem težinama dobijenim preko *AXI Stream* interfejsa.

Nakon što se izračuna izlaz poslednjeg neurona u poslednjem sloju, traži se maksimum vrednosti izlaza neurona iz poslednjeg sloja. Oni su upisani u poslednjih 10 nepraznih lokacija u BRAM memoriji. Po pronalasku maksimuma, *ready* registar dobija vrednost 1, a relativni indeks (od 0 do 9) memorijske lokacije maksimuma se skladišti u *cl_num* registar, odakle se čita klasifikovani broj.

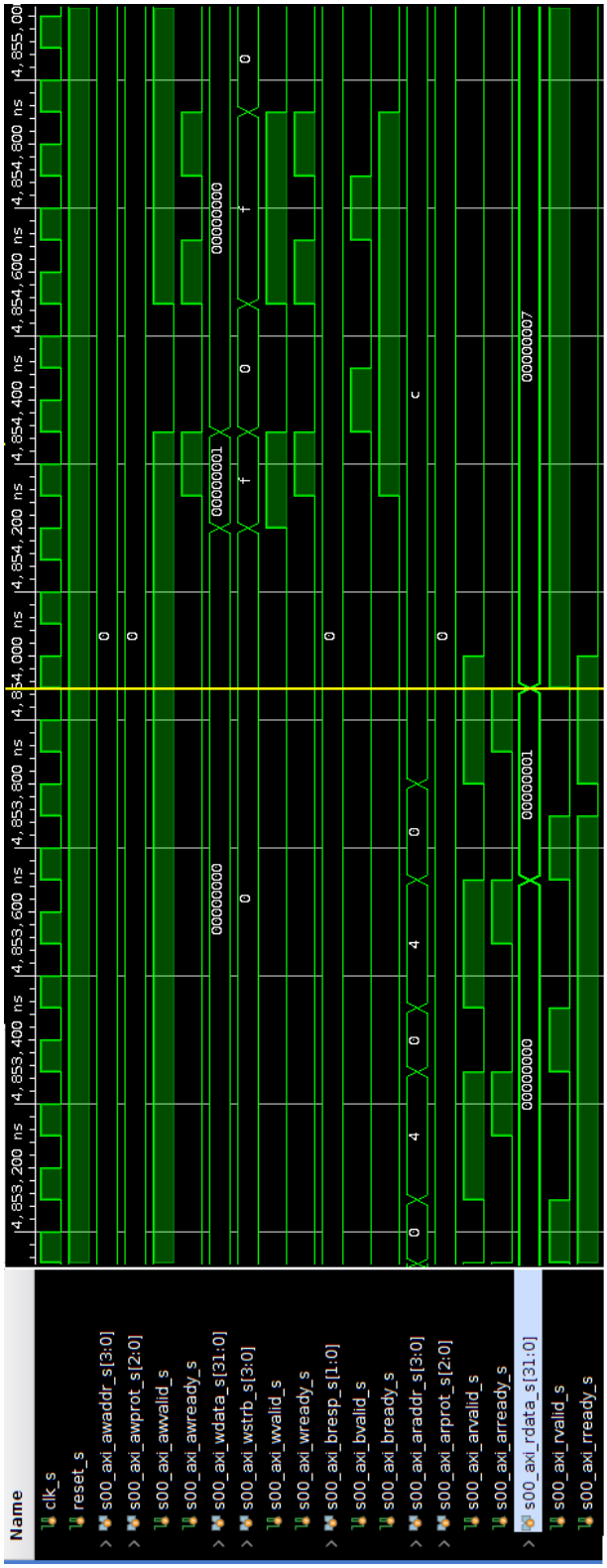
Na slici 4 su predstavljeni talasni oblici signala na *AXI Lite* interfejsu u trenutku nakon što su svi parametri prosleđeni jezgru i čeka se rezultat klasifikacije. Postavljanjem *s00_axi_araddr_s* signala na željenu adresu, kao i pratećih AXI signala na odgovarajuće vrednosti, čita se stanje *ready* registra na adresi 4 i čeka se da ono postaje 1, to jest da *s00_axi_rdata_s* signal dobije vrednost 1. Kada se to desi, čita se klasifikovani broj iz *cl_num* registra na adresi 12 ($0xC$ u heksadecimalnom zapisu). U ovom slučaju klasifikovani broj je 7. Nakon toga, može se započeti nova klasifikacija upisivanjem vrednosti 1 u *start* registar na adresi 0 - što se postiže postavljanjem signala *s00_awaddr_s* na vrednost 0 i *s00_axi_wdata_s* na 1. Nakon toga se pikseli slike i parametri neuronske mreže prosleđuju preko *AXI Stream* interfejsa.

4 Verifikacioni plan

Verifikacija će biti izvršena korišćenjem UVM metodologije. Alat koji će biti korišćen za simulaciju je *QuestaSim* (razvijen od strane *MentorGraphics*-a), a simulacija će se vršiti na jednom računaru. Provere funkcionalnosti koje treba izvršiti tokom verifikacije su:

1. Provera funkcionalnosti reseta sistema. Proveriti da li je jezgro nakon resetovanja u stanju spreman i da li pravilno prima i šalje kontrolne i statusne podatke.

2. Provera funkcionalnosti AXI Lite interfejsa. Proveriti da li jezgro postavlja podatke na ovaj interfejs poštujući protokol.



Slika 4: Talasni oblici signala od značaja

3. Provera ispravnosti klasifikacije.

Potrebno je napisati testove koji će uspeti da pobude i da provere sve gore-navedene funkcionalnosti. Verifikaciona strategija koja će se primeniti je randomizacija sa ograničenjima koja se usmerava prikupljanjem pokrivenosti (engl. $coverage-driven\ constraint\ random-based\ functional\ verification$), a dizajn će se posmatrati kao crna kutija (*Black Box* pristup). Podrška koju UVM pruža randomizaciji sa ograničenjima bila je odlučujući faktor za odabir ove strategije. Potrebno je stimuluse ograničiti na smislene transakcije koje će jezgro biti u stanju da pravilno protumači. Dakle, broj AXI Stream transfera i njihova sadržina treba da budu ograničeni na podatke koji predstavljaju piksele slike nekog broja, težine i *bias*-e. Stimulusi koji budu pobuđivali AXI *Lite* interfejs treba da budu dovoljno ograničeni da obezbede pravilan kontrolni tok. Pošto se sve funkcionalnosti mogu proveriti upisivanjem podataka i čitanjem preko definisanih (spoljašnjih) interfejsa, dizajn je dovoljno posmatrati kao crnu kutiju.

Dakle, napisani test treba da bude dovoljno usmeren, kako bi omogućio reset celog sistema, upis i čitanje iz svih AXI *Lite* registara i klasifikaciju većeg broja slika iz MNIST baze podataka. Na ovaj način biće osigurana verifikacija svih funkcionalnosti od značaja.

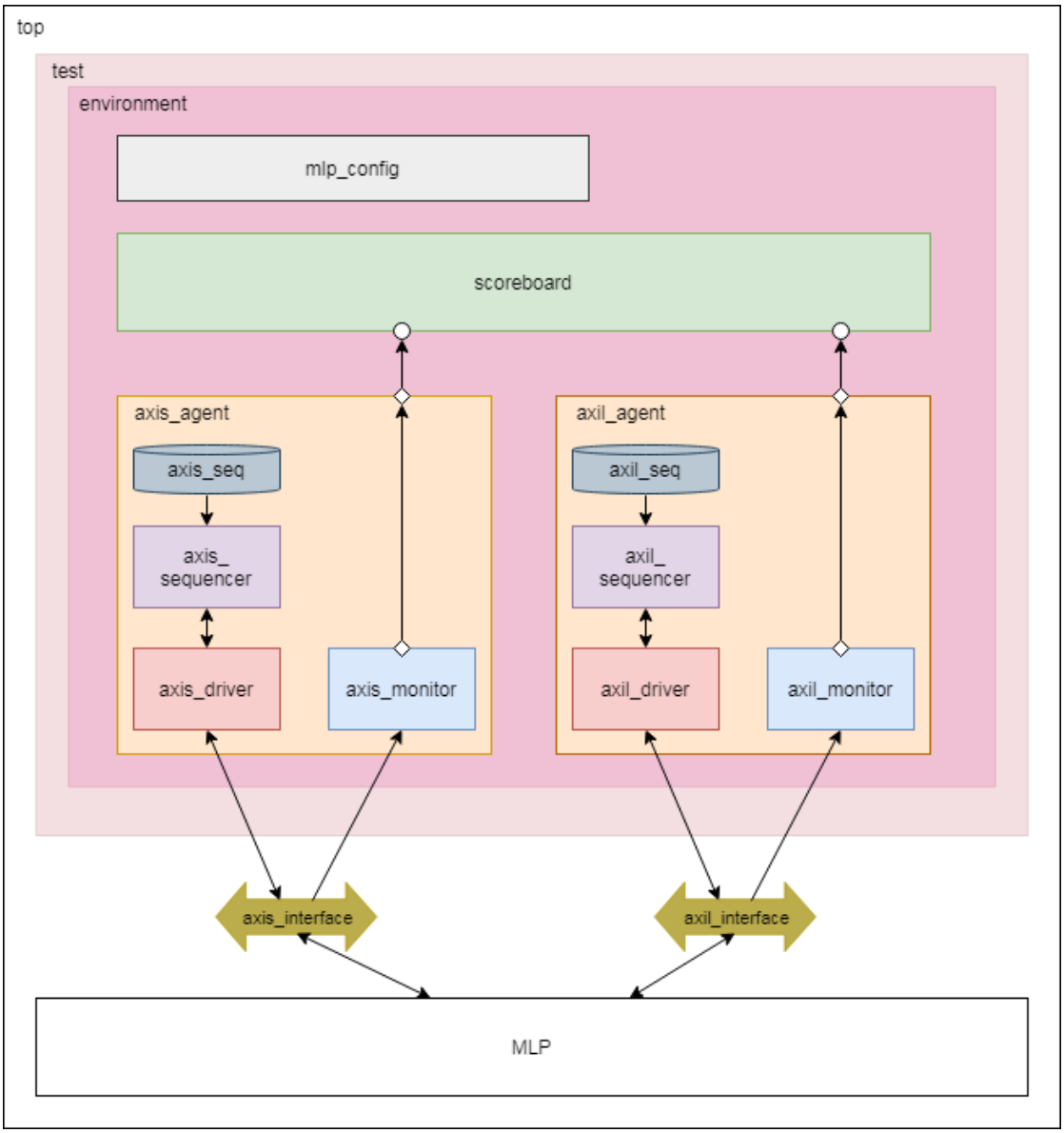
5 Verifikaciono okruženje

Verifikaciono okruženje je bazirano na univerzalnoj verifikacionoj metodologiji (UVM) i sadrži sledeće komponente: test, environment, scoreboard, configuration i agente za AXI *Stream* i AXI *Lite* interfejse, njima pripadajuće monitore, drajvere i sekvencere. Šematski prikaz verifikacionog okruženja je dat na slici 5. Svaka od komponenti predstavlja klasu koja nasleđuje odgovarajuću UVM klasu (nalaze se u UVM bilioteci) i opisane su pomoću *SystemVerilog* jezika.

5.1 axil_driver, axil_sequence, axil_sequencer, axil_monitor, axil_agent

Preko AXI Lajt (*engl. AXI Lite*) interfejsa jezgro dobija komandu za početak klasifikacije i daje informaciju o tome da li je spremno za novu klasifikaciju i rezultat klasifikacije.

Sekvencer ima ulogu stimulus generatora. U njemu se kontrolišu podaci koje će drajver slati dizajnu koji se verifikuje. Ova kontrola podataka se vrši u definisanim sekvencama koje određuju broj, sadržaj i redosled transakcija. [3] Kôd sekvencera je jednostavan, što se da videti na kodnom listingu 1. On se sastoji od $factory$



Slika 5: Verifikaciono okruženje MLP jezgra

registracije izvršene pomoću *'uvm_component_utils* makroa i definisanja konstruktora klase. Sva funkcionalnost sekvencera je nasleđena iz njegove roditeljske klase *uvm_sequencer*.

Listing 1: Kôd *AXI Lite* sekvencera

```
1  class axil_sequencer extends uvm_sequencer#(axil_frame);
2
3      'uvm_component_utils(axil_sequencer)
4
5      function new(string name = "axil_sequencer", uvm_component parent = null);
6          super.new(name,parent);
7      endfunction
8
9  endclass : axil_sequencer
```

Driver komponenta prevodi podatke sa nivoa transakcije (iz sekvence) na nivo interfejsa. Prilikom postavljanja signala na interfejs drajver vodi računa da AXI *Lite* protokol bude ispoštovan. Listing 2 prikazuje SystemVerilog kôd ove komponente. Na samom početku, prilikom deklarisanja klase, nasleđena je klasa *uvm_driver* i parametrizovana AXI *Lite* transakcijom. U 2. liniji koda iskorišćen je makro za *factory* registraciju komponente, a u narednoj liniji je deklarisan virtuelni interfejs. UVM *factory* omogućava jednostavnu zamenu postojeće klase izvedenom klasom bez menjanja ostatka okruženja, a virtuelni interfejs se koristi za povezivanje dizajna koji se testira i testbenča. [3] Posle toga, definisan je konstruktor klase i u *connect* fazi je preuzet interfejs koji će drajver koristiti i smešten je prethodno deklarisan virtuelni interfejs *vif*. U 18. liniji koda drajver u beskonačnoj petlji u *run* fazi koristi blokirajuću metodu *get_next_item* kako bi od sekvencera dobio transakciju iz sekvence na osnovu koje će postaviti signale na interfejs. Kada mu sekvencer prosledi sekvencu, drajver pomoću vrednosti *read_write* polja transakcije proverava da li je u pitanju transakcija čitanja ili upisa podataka. Na rastuću ivicu takta, u slučaju transakcije čitanja, postavlja podatke na adresnu i na magistralu podataka, kao i prateće signale, dok u slučaju transakcije upisa postavlja signale na adresnu magistralu i, nakon odbijanja odgovora od dizajna, podatak pročitan sa interfejsa smešta u *data* polje transakcije. Drajver osigurava da će protokol biti ispoštovan postavljanjem i čekanjem na aktiviranje odgovarajućih *ready* i *valid* signala. Nakon što završi sa postavljanjem signala na interfejs i dobije potrebne odgovore od dizajna koji se testira, on javlja sekvenceru da je sa obradom trenutne transakcije završio koristeći neblokirajuću metodu *item_done* u 49. liniji koda.

Listing 2: Kôd *AXI Lite* drajvera

```
1  class axil_driver extends uvm_driver#(axil_frame);
2     'uvm_component_utils(axil_driver)
```

```
3   virtual interface axil_if vif;
4
5   function new(string name = "axil_driver", uvm_component parent = null);
6       super.new(name,parent);
7   endfunction
8
9   function void connect_phase(uvm_phase phase);
10      super.connect_phase(phase);
11      if (!uvm_config_db#(virtual axil_if)::get(this, "*", "axil_if", vif))
12        `uvm_fatal("NOVIF",{"virtual interface must be set for: ",get_full_name(),".vif"})
13  endfunction : connect_phase
14
15  task run_phase(uvm_phase phase);
16      @(negedge vif.rst);
17      forever begin
18          seq_item_port.get_next_item (req);
19          `uvm_info (get_type_name(), $sformatf (" Driver sending...\n%s", req.sprint()),
                ↪ UVM_FULL)
20
21            @(posedge vif.clk) begin
22          if(req.read_write) begin //read = 0, write = 1
23                  vif.s_axi_awaddr = req.address;
24                  vif.s_axi_awvalid = 1;
25                  vif.s_axi_wdata = req.data;
26                  vif.s_axi_wvalid = 1;
27                  vif.s_axi_bready = 1;
28                  wait(vif.s_axi_awready && vif.s_axi_wready);
29                  wait(vif.s_axi_bvalid);
30                  vif.s_axi_wdata = 0;
31                  vif.s_axi_awvalid = 0;
32                  vif.s_axi_wvalid = 0;
33                  wait(!vif.s_axi_bvalid);
34                  vif.s_axi_bready = 0;
35              end
36              else begin
37                  vif.s_axi_araddr = req.address;
38                  vif.s_axi_arvalid = 1;
39                  vif.s_axi_rready = 1;
40                  wait(vif.s_axi_arready);
41                  wait(vif.s_axi_rvalid);
42                  vif.s_axi_arvalid = 0;
43                  vif.s_axi_araddr = 0;
44                  wait(!vif.s_axi_rvalid);
45                  req.data = vif.s_axi_rdata;
46                  vif.s_axi_rready = 0;
47              end
48            end
49          seq_item_port.item_done();
50      end
```

```
51      endtask : run_phase
52
53  endclass : axil_driver
```

Sekvence su UVM objekti koji sadrže logiku generisanja stimulusa. Bazna sekvenca je napisana tako da se u *pre_body task*-u podigne prigovor, a u *post_body task*-u prigovor spusti. Podizanje i spuštanje prigovora utiče na to kada će se određena faza u testu završiti. Svaka faza se završava kada nema prigovora za tu fazu što znači da se kraj testa kontroliše podizanjem i spuštanjem prigovora. [4] U baznoj sekvenci je deklarisan i sekvencer na kome će se bazna sekvenca (i sve sekvence koje je naslede) pokretati. Ovo je učinjeno pomoću makroa *‘uvm_declare_p_sequencer*. Njen kôd je dat listingom 3

Listing 3: Kôd *AXI Lite* bazne sekvence

```
1   class axil_base_seq extends uvm_sequence#(axil_frame);
2
3       ‘uvm_object_utils(axil_base_seq)
4       ‘uvm_declare_p_sequencer(axil_sequencer)
5
6       function new(string name = "axil_base_seq");
7           super.new(name);
8       endfunction
9
10      virtual task pre_body();
11          uvm_phase phase = get_starting_phase();
12          if (phase != null)
13              phase.raise_objection(this, {"Running sequence ’", get_full_name(), "’"});
14      endtask : pre_body
15
16      virtual task post_body();
17          uvm_phase phase = get_starting_phase();
18          if (phase != null)
19              phase.drop_objection(this, {"Completed sequence ’", get_full_name(), "’"});
20      endtask : post_body
21
22  endclass : axil_base_seq
```

AXI *Lite* sekvenca je napisana tako da obezbedi pravilan redosled upisa i čitanja u unutrašnje registre MLP jezgra. U 1. liniji koda u listingu 4 klasa *axil_seq* nasleđuje klasu *axil_base_seq*. U *axil_sequence* sekvenci nakon nasleđivanja bazne sekvence sledi *factory* registracija *axil_seq* objekta i definisanje njegovog konstruktora. Logika generisanja stimulusa nalazi se u *body task*-u. U 11. liniji koda se randomizuje promenljiva *num_of_images* koja određuje broj slika koje će se klasifikovati. Nakon toga, počevši od 15. linije koda, sledi niz *‘uvm_do_with* makroa. *‘uvm_do_with* makro implementira metode *start_item*, randomizaciju sa ograničenjima i *finish_item*. Sve ovo omogućava sinhronizaciju rada sekvencera i

drajvera. Naime, u sekvenci se poziva blokirajuća metoda *start_ item*, koja čeka da drajver zatraži sledeću transakciju (pozivanjem *get_next_item* metode). Nakon toga sekvenca randomizuje transakciju sa zadatim ograničenjima i poziva blokirajuću *finish_ item* metodu, obaveštavajući drajver da je transakcija randomizovana i da je spremna za procesiranje. Za detalje komunikacije sekvencera i drajvera pogledati [4]. Kao argumenti *uvm_do_with* makroa se navode transakcija koja će biti poslata drajveru i ograničenja randomizacije te transakcije. Upravo zahvaljujući ovim ograničenjima je moguće generisati stimuluse na pravilan način. Nakon što se pošalju dve transakcije čitanja iz registara jezgra, u 18. liniji se u *start* registar upisuje 1. U 23. liniji počinje beskonačna petlja u kojoj se čita stanje *ready* registra. Ukoliko je iz *ready* registra pročitana 1, šalje se transakcija koja obezbeđuje čitanje klasifikovanog broja i inkrementira se *image* promenljiva. Kada promenljiva *image* dostigne vrednost randomizovane *num_of_images* promenljive, dolazi do izlaska iz petlje. Na 31. i 32. liniji koda je u *start* registar upisana prvo 1, a zatim 0, kako se ne bi desilo da jezgro započne klasifikaciju nove slike bez eksplicitne dozvole. Ukoliko je iz *ready* registra pročitana 0, jednostavno se prelazi u novu iteraciju petlje. Ovim je osigurano da se sa čitanjem rezultata klasifikacije i započinjanjem nove klasifikacije sačeka sve dok jezgro zaista ne bude spremno.

Listing 4: Kôd *AXI Lite* sekvence

```
1  class axil_seq extends axil_base_seq;
2      int num_of_images = 0;
3      int image = 0;
4      `uvm_object_utils (axil_seq)
5
6      function new(string name = "axil_seq");
7          super.new(name);
8      endfunction
9
10     virtual task body();
11         assert (std::randomize(num_of_images) with {num_of_images > 0; num_of_images
               ↪ <= 10;});
12
13         `uvm_info (get_type_name(), $sformatf("%d images are being classified ",
               ↪ num_of_images), UVM_NONE);
14         //random reading from MLP
15         `uvm_do_with (req, {req.read_write == 0; req.data == 1; req.address == 4;});
16         `uvm_do_with (req, {req.read_write == 0; req.data == 1; req.address == 0;});
17         //start MLP
18         `uvm_do_with (req, {req.read_write == 1; req.data == 1; req.address == 0;});
19         //random reading from MLP
20         `uvm_do_with (req, {req.read_write == 0; req.data == 0; req.address == 0;});
21         `uvm_do_with (req, {req.read_write == 1; req.data == 0; req.address == 0;});
22         `uvm_info(get_type_name(), $sformatf("image %d is being classified ",image),
               ↪ UVM_NONE);
```

```
23          forever begin
24              `uvm_do_with (req, {req.read_write == 0; req.data == 1; req.address == 4;});
25               if (req.data == 1)begin
26                `uvm_do_with (req, {req.read_write == 0; req.data == 1; req.address == 12;});
27                 image ++;
28                 if (image == num_of_images)
29                     break;
30                 `uvm_info (get_type_name(), $sformatf("image number %d is being classified ",
                       ↪ image), UVM_NONE);
31                 `uvm_do_with (req, {req.read_write == 1; req.data == 1; req.address == 0;});
32                 `uvm_do_with (req, {req.read_write == 1; req.data == 0; req.address == 0;});
33             end
34         end
35     endtask : body
36
37 endclass : axil_seq
```

Monitor nadgleda interfejs i uočene promene pakuje u transakciju koju šalje *scoreboard*-u. U kodnom listingu 5 se primećuje da je pored *factory* registracije i deklarisanja virtuelnog interfejsa (što je bilo prisutno i u drajveru), u monitoru deklarisan i TLM port *item_collected_port* parametrizovan *axil_frame* transakcijom. Ovaj port služi kao deo TLM interfejsa za komunikaciju monitora i *scoreboard*-a. U konstruktoru na 10. liniji koda su pored same komponente kreirani i *item_collected_port* i dve grupe pokrivenosti, koje su iz ovog kodnog listinga izostavljene. Za detalje o implementaciji prikupljanja pokrivenosti pogledati glavu 6. Na 17. liniji je data *connect* faza u kojoj je preuzet interfejs i smešten u promenljivu *vif*. U *run* fazi je prisutna beskonačna petlja u kojoj se kreira nova transakcija pod nazivom *current_frame*. Razlog za kreiranje nove transakcije u svakoj iteraciji petlje je taj što se transakcija prosleđuje preko TLM porta *scoreboard*-u i radi nesmetanog, pravilnog rada *scoreboard*-a je bitno da se jednom prosleđena transakcija ne menja, već da se dodela novih vrednosti poljima transakcije vrši nad novokreiranom transakcijom. Nakon ulaska u *forever* petlju i kreiranja nove transakcije, na 26. liniji se čeka na rastuću ivicu takta, nakon čega se proveravaju tri uslova: da li je aktivan *s_axi_awready* signal, da li je aktivan *s_axi_arready* signal i da li je aktivan *s_axi_rvalid* signal. Ukoliko je na rastuću ivicu takta aktivan *s_axi_awready* signal, jasno je da se radi o transakciji upisa podataka, pa se dodeljuje vrednost odgovarajućoj *address* promenljivoj i *sample*-uje se *write_address* grupa pokrivenosti. Ukoliko je na rastuću ivicu takta aktivan *s_axi_arready* signal, radi se o transakciji čitanja podataka i u tom trenutku se prikuplja adresa sa koje se podatak čita. Ukoliko je aktivan *s_axi_rvalid*, to znači da je dizajn postavio pročitan podatak na interfejs (*s_axi_rdata* signal), pa je sada moguće *sample*-ovati *read_address* grupu pokrivenosti i postaviti *data* i *address* polja *current_frame* transakcije. Na kraju se, u 37. liniji koda, prikupljena transakcija prosleđuje *scoreboard*-u putem *item_collected* TLM porta.

Listing 5: Kôd *AXI Lite* monitora

```
1  class axil_monitor extends uvm_monitor;
2      bit [31:0] address;
3      uvm_analysis_port #(axil_frame) item_collected_port;
4      `uvm_component_utils(axil_monitor)
5      virtual interface axil_if vif;
6      axil_frame current_frame;
7
8      [... coverage code... ]
9
10     function new(string name = "axil_monitor", uvm_component parent = null);
11         super.new(name,parent);
12         item_collected_port = new("item_collected_port", this);
13         write_address = new();
14         read_address = new();
15     endfunction
16
17     function void connect_phase(uvm_phase phase);
18         super.connect_phase(phase);
19         if (!uvm_config_db#(virtual axil_if)::get(this, "*", "axil_if", vif))
20           `uvm_fatal("NOVIF",{"virtual interface must be set for: ",get_full_name(),".vif"})
21     endfunction : connect_phase
22
23     task run_phase(uvm_phase phase);
24         forever begin
25             current_frame = axil_frame::type_id::create("current_frame", this);
26             @(posedge vif.clk)begin
27                 if(vif.s_axi_awready )begin
28                     address = vif.s_axi_awaddr;
29                     write_address.sample();
30                 end
31                 if(vif.s_axi_aready)
32                     address = vif.s_axi_araddr;
33                 if(vif.s_axi_rvalid)begin
34                     read_address.sample();
35                     current_frame.data = vif.s_axi_rdata;
36                     current_frame.address = address;
37                     item_collected_port.write(current_frame);
38                 end
39             end
40         end
41     endtask : run_phase
42
43 endclass : axil_monitor
```

Agent enkapsulira drajver, sekvencer i monitor u jednu komponentu koja komunicira sa ostatkom verifikacionog okruženja. U listingu 6 primećuje se da se sve tri komponente kreiraju u *build* fazi u 13. liniji koda, a drajver i sekvencer

se povezuju pomoću odgovarajućih TLM portova u 22. liniji koda u *connect* fazi. Portove *seq_item_port* i *seq_item_export* nismo eksplicitno deklarisali, već su oni deo UVM-a i deklarisane su u roditeljskim klasama. Povezivanje monitora i *scoreboard*-a se vrši u *build* fazi *environment* komponente, a ne agenta, s obzirom na to da *scoreboard* ne pripada agent komponenti.

Listing 6: Kôd *AXI Lite* agenta

```
1  class axil_agent extends uvm_agent;
2      axil_driver axil_drv;
3      axil_monitor axil_mon;
4      axil_sequencer axil_seqr;
5
6      `uvm_component_utils_begin (axil_agent)
7      `uvm_component_utils_end
8
9      function new(string name = "axil_agent", uvm_component parent = null);
10         super.new(name,parent);
11     endfunction
12
13     function void build_phase(uvm_phase phase);
14         super.build_phase(phase);
15         axil_drv = axil_driver::type_id::create("axil_drv", this);
16         axil_seqr = axil_sequencer::type_id::create("axil_seqr", this);
17         axil_mon = axil_monitor::type_id::create("axil_monitor", this);
18     endfunction : build_phase
19
20     function void connect_phase(uvm_phase phase);
21         super.connect_phase(phase);
22         axil_drv.seq_item_port.connect(axil_seqr.seq_item_export);
23     endfunction : connect_phase
24
25 endclass : axil_agent
```

5.2 axis_driver, axis_sequence, axis_sequencer, axis_monitor, axis_agent

AXI *Stream* bazna sekvenca, slično kao u AXI *Lite* slučaju, deklariše sekvencer na kome će se pokretati i podiže i spušta prigovore u *pre_body* i *post_body task*-ovima, respektivno. U baznoj sekvenci je implementirana i funkcija *extract_data*, koja čita piksele slika za klasifikaciju i parametre neuronske mreže iz odgovarajućih fajlova i smešta ih u redove. Na taj način, svaka sekvenca koja nasledi baznu sekvencu ima pristup ovim podacima. Njen kod može se pronaći u pratećim fajlovima.

Listing 7 predstavlja kôd AXI *Stream* sekvence. U *body task*-u, na liniji 11,

počinje *for* petlja koja iterira deset puta i predstavlja slanje 10 slika za klasifikaciju. Napomenuto je da pravi broj slika koje će jezgro klasifikovati zavisi od randomizovane promenljive u AXI *Lite* sekvenci, a koja je ograničena na opseg od 1 do 10. U 14. liniji koda kreiramo *req* AXI *Stream* transakciju i pozivamo metodu *start_item* (objašnjeno u poglavlju o AXI *Lite* sekvenci). Ova transakcija za polje ima *dataQ* red u koji se sada smešta 784 piksela slike za klasifikaciju. Nakon toga se poziva *finish_item* metoda kao deo standardne komunikacije sa drajverom. Dalje se na sličan način pojedinačno kreiraju i šalju transakcije koje predstavljaju težine i *bias* za svaki neuron. Sekvenca obezbeđuje da se sve transakcije pošalju u tačno definisanom redosledu, kako bi jezgro podatke protumačilo na pravilan način. U ovoj sekvenci, iako se transakcije kreiraju i pozivaju se metode *start_ item* i *finish_ item*, ipak izostaje randomizacija. Razlog tome je što nije bilo polja u transakciji koja bi bilo potrebno randomizovati. Jedini način da se dobije smisleni, usmeren test je da se jezgru šalju podaci smislenog sadržaja i broja.

Listing 7: Kôd *AXI Stream* sekvence

```
1  class axis_seq extends axis_base_seq;
2
3      `uvm_object_utils (axis_seq)
4      int i = 1;
5      function new(string name = "axis_seq");
6          super.new(name);
7      endfunction
8
9      virtual task body();
10         `uvm_info(get_type_name(), $sformatf("Sequence starting..."), UVM_HIGH)
11         for (image = 0; image < 10; image ++)
12         begin
13             `uvm_info(get_type_name(), $sformatf("Sending image number %d.",image),
                   ↪ UVM_HIGH);
14             req=axis_frame::type_id::create("req");
15             start_item(req);
16             for (i = 0; i < IMG_LEN; i ++)
17                 req.dataQ.push_back (yQ [image * 784 + i]);
18             finish_item(req);
19             for (int layer = 1; layer < 3; layer ++)
20             begin
21                 `uvm_info(get_type_name(), $sformatf("Layer number %d calculating",layer),
                       ↪ UVM_NONE);
22                 for (int neuron = 0; neuron < neuron_array[layer]; neuron++)
23                 begin
24                     //send weights
25                     req=axis_frame::type_id::create("req");
26                     start_item(req);
27                     for(i=0; i<neuron_array[layer - 1]; i++)
28                         req.dataQ.push_back(weightQ[layer-1][neuron*neuron_array[layer-1] + i])
```

```
                    ↪ ;
29                  finish_item(req);
30
31                  //send bias
32                  req=axis_frame::type_id::create("req");
33                  start_item(req);
34                  req.dataQ.push_back(biasQ[layer-1][neuron]);
35                  finish_item(req);
36              end
37          end
38      end
39    endtask : body
40
41  endclass : axis_seq
```

U listingu 8 prikazan je segment koda AXI *Stream* drajvera. Radi preglednosti data je samo *run* faza, pošto je ostatak koda sličan AXI *Lite* drajveru. U beskonačnoj petlji se prvo postavljaju signali *s_axis_tlast* i *s_axis_tvalid* na 0, a potom se zahteva transakcija iz sekvence pomoću metode *get_next_item*. Na 12. liniji koda se nalazi petlja koja prolazi kroz svaki podatak iz reda *dataQ* iz transakcije. Ukoliko je u pitanju poslednji podatak u redu, aktivira se signal *s_axis_tlast*. Za svaki podatak se, nakon kašnjenja od 1 vremenske jedinice, aktivira signal *s_axis_tvalid* i dodeljuje se vrednost podatka iz reda signalu *s_axis_tdata*, a potom se čeka na odgovor jezgra u vidu aktiviranja *s_axis_tready* signala. Nakon što su svi podaci iz reda procesirani, poziva se metoda *item_done*, koja omogućava formiranje sledeće transakcije u sekvenci.

Listing 8: Kôd *AXI Stream* drajvera

```
1   class axis_driver extends uvm_driver#(axis_frame);
2
3     [...]
4     task run_phase(uvm_phase phase);
5         forever
6         begin
7       @(negedge vif.clk);
8       vif.s_axis_tlast=0;
9       vif.s_axis_tvalid=0;
10      seq_item_port.get_next_item(req);
11
12      foreach (req.dataQ[i])
13      begin
14          #1 if(i==(req.dataQ.size-1))
15              vif.s_axis_tlast=1;
16          else
17              vif.s_axis_tlast=0;
18          vif.s_axis_tvalid=1;
19          vif.s_axis_tdata=req.dataQ[i];
```

```
20          @(posedge vif.clk iff vif.s_axis_tready);
21        end
22        seq_item_port.item_done();
23         end
24     endtask : run_phase
25   [...]
26
27 endclass : axis_driver
```

Monitor nadgleda AXI *Stream* interfejs. Ovaj monitor sličan je AXI *Lite* monitoru. Kodni listing 9 prikazuje da u *run* fazi AXI *Stream* monitor prati s_axis_tready i s_axis_tvalid signale i na onu rastuću ivicu takta, kada oni istovremeno budu aktivni, on skladišti podatak sa s_axis_tdata magistrale u red *dataQ*. Kada uoči signal s_axis_tlast za poslednji podatak u transakciji, sakupljenu transakciju šalje komponenti *scoreboard* preko $item\_collected\_port$ porta.

Listing 9: Kôd *AXI Stream* monitora

```
1  class axis_monitor extends uvm_monitor;
2
3      uvm_analysis_port #(axis_frame) item_collected_port;
4      `uvm_component_utils_begin(axis_monitor)
5      `uvm_component_utils_end
6      virtual interface axis_if vif;
7      axis_frame current_frame;
8
9      function new(string name = "axis_monitor", uvm_component parent = null);
10         super.new(name,parent);
11         item_collected_port = new("item_collected_port", this);
12     endfunction
13
14     function void connect_phase(uvm_phase phase);
15         super.connect_phase(phase);
16         if (!uvm_config_db#(virtual axis_if)::get(this, "*", "axis_if", vif))
17             `uvm_fatal("NOVIF",{"virtual interface must be set for: ",get_full_name(),".vif"
                   ↪ })
18     endfunction : connect_phase
19
20     task run_phase(uvm_phase phase);
21          current_frame = axis_frame::type_id::create("current_frame", this);
22         forever begin
23             @(posedge vif.clk iff (vif.s_axis_tready && vif.s_axis_tvalid));
24             current_frame.dataQ.push_back(vif.s_axis_tdata);
25             if(vif.s_axis_tlast)
26             begin
27                 item_collected_port.write(current_frame);
28             end
29         end
30     endtask : run_phase
```

```
31
32 endclass : axis_monitor
```

AXI *Stream* agent gore navedene komponente okuplja u jednu. Zbog sličnosti sa ranije objašnjenim AXI *Lite* agentom, AXI *Stream* agent neće biti posebno analiziran.

5.3 scoreboard

Unutar ove komponente implementiran je referentni model MLP jezgra. Na osnovu podataka koje dobija od monitora, *scoreboard* vrši proračune i upoređuje svoj rezultat klasifikacije sa onim koje je dalo jezgro. Listingom 10 dat je skraćeni kod *scoreboard* komponente. Prve 2 linije koda čine makroi '*uvm_analysis_imp_decl*() kojima se automatski deklarišu implementacioni portovi naziva *uvm_analysis_imp_axis* i *uvm_analysis_imp_axil* i njihove *write* funkcije. Ovi portovi će u *environment*-u biti povezani sa portovima na monitorima. Monitori nadgledaju interfejse i svaki put kada uoče smislenu transakciju, oni je prosleđuju scoreboardu preko TLM portova. Svaki put kada transakcija pristigne na port koji je povezan sa monitorom, aktiviraće se njemu pripadajuća *write* funkcija.

Nakon deklaracije klase, nasleđivanja *uvm_scoreboard* klase i definisanja potrebnih promenljivih i redova, na 17. i 18. liniji se definišu ranije deklarisani implementacioni portovi *port_axis* i *port_axil*. Oni su parametrizovani tipom transakcije i komponentom. U konstruktoru klase potrebno je napraviti ove portove (linija 25. i 26.).

write_axis funkcija na 29. liniji sakuplja sve pristigle transakcije i tumači ih kao ulazne piksele, težine ili *bias*-e. I referentnom modelu je, kao i jezgru, bitno da se podaci šalju u predefinisanom redosledu (što je obezbeđeno u sekvenci) radi njihovog ispravnog tumačenja. Jednu AXI *Stream* transakciju čine ili pikseli slike ili težine za jedan neuron ili bias za jedan neuron. Monitor će svaki od nabrojanih slučaja tretirati kao zasebnu transakciju koju će poslati komponenti *scoreboard*, što će svaki put dovesti do aktiviranja *write_axis* funkcije. Zato je neophodno korišćenjem pomoćnih promenljivih(*img*, *is_bias*, *neuron*, *layer*) i proverom brojnih uslova tačno odrediti koji podaci su već stigli, a koji trenutno stižu. Kada je i poslednji podatak primljen, *write_axis* poziva funkciju *calc_res* u kojoj je implementiran referentni model jezgra. Ova funkcija koristi isti algoritam kao i MLP IP *core* i rezultat klasifikacije smešta u promenljivu *result*, a za računanje koristi prethodno pristigle podatke skladištene u odgovarajuće redove. Kôd ove funkcije se može pronaći u pratećim fajlovima, a zbog jednostavnosti i strogog praćenja MLP algoritma, ovde neće biti detaljnije razmatran.

Kada AXI *Lite* monitor pošalje transakciju, aktivira se *write_axil* funkcija. U njoj se proverava sadržaj transakcije na osnovu kog se saznaje stanje jezgra i

rezultat klasifikacije. Proverava se adresno polje i polje podataka. Na 65. liniji, ako sadržaj pristigle transakcije ukazuje na to da je iz registra na adresi 4 (ready registra) pročitana vrednost 1, promenljiva *res_ready* se postavlja na 1, kako se prilikom sledećeg aktiviranja funkcije *write_axil* ne bi izgubila informacija da je jezgro spremno za rad. U suprotnom, proverava se da li je pročitan *cl_num* registar i da li je *res_ready* na 1 (što bi značilo da je pre ovoga već jednom pročitano da je *ready* registar setovan). Ako to jeste slučaj, na 72. liniji koda klasifikovani broj se upoređuje sa rezultatom do kog je došao referentni model i prijavljuje se greška ukoliko je došlo do neslaganja.

Funkcija *clear_queues* briše podatke iz redova.

Listing 10: Kôd *scoreboard*-a

```
1  `uvm_analysis_imp_decl(_axis)
2  `uvm_analysis_imp_decl(_axil)
3  class scoreboard extends uvm_scoreboard;
4      localparam bit[9:0] neuron_array[0:2] = {10'd784, 10'd30, 10'd10};
5      localparam bit[9:0] IMG_LEN = 10'd784;
6      bit res_ready = 0;
7      bit[3:0] result =0;
8      int img = 0;
9      int layer = 1;
10     int neuron = 0;
11     int is_bias = 0;
12     logic[17 : 0] yQ[$];
13     logic[17 : 0] weightQ[2][$];
14     logic[17 : 0] biasQ[2][$];
15     int num_of_assertions = 0;
16
17     uvm_analysis_imp_axis#(axis_frame, scoreboard) port_axis;
18     uvm_analysis_imp_axil#(axil_frame, scoreboard) port_axil;
19
20     int num_of_tr;
21     logic [31:0] reference_model_image [784];
22
23     function new(string name = "scoreboard", uvm_component parent = null);
24         super.new(name,parent);
25         port_axis = new("port_axis", this);
26         port_axil = new("port_axil", this);
27     endfunction : new
28
29     function write_axis (axis_frame tr);
30      if (img == 0)
31      begin
32          clear_queues();
33          img = 1;
34          foreach (tr.dataQ[i])
35          yQ.push_back (tr.dataQ[i]);
```

```
36      end
37      else begin
38          if (is_bias == 0)
39          begin
40              foreach(tr.dataQ[i])
41              weightQ[layer-1].push_back(tr.dataQ[i]);
42              is_bias = 1;
43          end
44          else begin
45              biasQ[layer-1].push_back(tr.dataQ[0]);
46              neuron ++;
47              is_bias = 0;
48              if (neuron == neuron_array[layer])
49              begin
50                  layer++;
51                  neuron = 0;
52                  if (layer == 3)
53                   begin
54                      calc_res();
55                      `uvm_info(get_type_name(), $sformatf("Classified number is: %d",result
                            ↪ ), UVM_LOW);
56                      img = 0;
57                      layer = 1;
58                   end
59              end
60          end
61        end
62      endfunction : write_axis
63
64      function write_axil (axil_frame tr);
65           if(tr.address==4 && tr.data==1)
66           begin
67              res_ready=1;
68           end
69           else if(tr.address==12 && res_ready)
70           begin
71              res_ready=0;
72              assert(tr.data==result)
73              else
74              begin
75                  `uvm_error(get_type_name(), $sformatf("res mismatch, reference model: %d \
                        ↪ t mlp: %d", result, tr.data));
76                  num_of_assertions++;
77              end
78           end
79      endfunction : write_axil
80
81      function void clear_queues ();
82          yQ={};
```

```
83          for(int l=0; l<2; l++)
84          begin
85              weightQ[l]={};
86              biasQ[l]={};
87          end
88      endfunction
89
90      function void calc_res ();
91       [...]
92       result = cl_num;
93      endfunction
94
95  endclass : scoreboard
```

5.4 environment i test

Environment instancionira i enkapsulira sve prethodno opisane komponente (listing 11). U *build* fazi *environment* kreira agente i *scoreboard* i preuzima konfiguraciju. U *connect* fazi se povezuju *item_collected* portovi AXI *Lite* i AXI *Stream* monitora sa implementacionim portovima *port_axil* i *port_axis* na *scoreboard*-u.

Listing 11: Kôd *environment*-a

```
1   class env extends uvm_env;
2       axil_agent mlp_axil_agent;
3       axis_agent mlp_axis_agent;
4       scoreboard scbd;
5       mlp_config cfg;
6
7       `uvm_component_utils (env)
8
9       function new(string name = "env", uvm_component parent = null);
10          super.new(name,parent);
11      endfunction
12
13      function void build_phase(uvm_phase phase);
14          super.build_phase(phase);
15          mlp_axis_agent = axis_agent::type_id::create("mlp_axis_agent", this);
16          mlp_axil_agent = axil_agent::type_id::create("mlp_axil_agent", this);
17          scbd = scoreboard::type_id::create("scbd", this);
18          if(!uvm_config_db#(mlp_config)::get(this, "", "mlp_config", cfg))
19            `uvm_fatal("NOCONFIG",{"Config object must be set for: ",get_full_name(),".cfg"
                ↪ })
20      endfunction : build_phase
21
22      function void connect_phase(uvm_phase phase);
23          super.connect_phase(phase);
24          mlp_axil_agent.axil_mon.item_collected_port.connect(scbd.port_axil);
```

```
25        mlp_axis_agent.axis_mon.item_collected_port.connect(scbd.port_axis);
26    endfunction : connect_phase
27
28 endclass : env
```

Test instancionira *environment* i pokreće sekvence na sekvencerima. Zbog svoje jednostavnosti kôd baznog testa nije prikazan ovde, a može se naći u pratećim fajlovima. U baznom testu *test_base* u *build* fazi se kreira *environment* i konfiguracija, koja se i setuje. U *build* fazi testa *test_mlp_simple* se kreiraju sekvence *axil_seq* i *axis_seq*, a u *run* fazi se podiže prigovor, sekvence se pokreću na svojim sekvencerima i prigovor se spušta. Listing 12 prikazuje kôd ovog testa.

Listing 12: Kôd testa

```
1  class test_mlp_simple extends test_base;
2
3      `uvm_component_utils(test_mlp_simple)
4      axil_seq mlp_axil_seq;
5      axis_seq mlp_axis_seq;
6
7      function new(string name = "test_mlp_simple", uvm_component parent = null);
8          super.new(name,parent);
9      endfunction : new
10
11     function void build_phase(uvm_phase phase);
12         super.build_phase(phase);
13         mlp_axil_seq = axil_seq::type_id::create("mlp_axil_seq");
14         mlp_axis_seq = axis_seq::type_id::create("mlp_axis_seq");
15     endfunction : build_phase
16
17     task run_phase(uvm_phase phase);
18         phase.raise_objection(this);
19         phase.phase_done.set_drain_time(this, 1000);
20       if(1==1) begin
21         fork
22           mlp_axis_seq.start(mlp_env.mlp_axis_agent.axis_seqr);
23           mlp_axil_seq.start(mlp_env.mlp_axil_agent.axil_seqr);
24         join_any
25        end
26         phase.drop_objection(this);
27     endtask : run_phase
28
29 endclass : test_mlp_simple
```

6 Verifikaciona pokrivenost

Scenariji koje je potrebno proveriti za pokrivenost su vezani za statusne i kontrolne registre kojima se pristupa AXI *Lite* interfejsom. Praćenjem podataka na AXI *Lite* magistralama mogu se zapravo ispratiti stanja u koja dovodimo jezgro, pošto se ovim interfejsom suštinski njime upravlja. Sakupljanje pokrivenosti je zato implementirano u *axil_monitor* komponenti i prikazano je u listingu 13. U ovu svrhu su kreirane odgovarajuće grupe pokrivenosti (engl. *covergroups*) i tačke pokrivenosti (engl. *coverpoints*)

Listing 13: Kôd za prikupljanje pokrivenosti

```
1  covergroup write_address;
2        option.per_instance = 1;
3        write_address: coverpoint address{
4           bins write_address_bin = {0};
5        }
6        data_write: coverpoint vif.s_axi_wdata {
7           bins start_0 = {0};
8           bins start_1 = {1};
9        }
10    endgroup
11
12    covergroup read_address;
13       option.per_instance = 1;
14       read_address: coverpoint address{
15          bins start_address_bin = {0};
16          bins ready_address_bin = {4};
17       }
18       data_read: coverpoint vif.s_axi_rdata{
19          bins data_bin_ready = {1};
20          bins data_bin_not_ready = {0};
21       }
22       cp_cross: cross read_address, data_read;
23    endgroup
```

Provereni su sledeći slučajevi:

- Čitanje iz svih internih registara. *Coverpoint* read_address proverava da li se desilo čitanje registara od značaja. Na osnovu specifikacije, radi se o registrima na adresama 0 i 4.
- Upis u sve interne registre. *Coverpoint* write_address proverava da li se desio upis na sve registarske adrese od značaja. Značajan je upis u *start* registar, te je samo njegova adresa i navedena.
- Čitanje posle upisa.

- Upis posle čitanja.
- Upis posle upisa.
- Čitanje posle čitanja.

Coverpoint cp_cross proverava poslednja četiri slučaja. Naime, ovaj *cross coverpoint* omogućava da se proveri da li su sa svih adresa navedenih u *read_address* tački pokrivenosti pročitane sve vrednosti navedene u *data_read* tački pokrivenosti. Ukoliko su iz registra *start* na adresi 0 pročitane i vrednost 0 i vrednost 1, to znači da smo ga jednom pročitali pre slanja komande za početak klasifikacije i još jednom posle slanja te komande. Isti je slučaj i sa *ready* registrom. Dakle, ova pokrivenost osigurava da su stimulusi za jezgro dobro napisani i da iniciraju klasifikaciju.

Opcija *option.per_instance* = 1 omogućava praćenje pokrivenosti za svaku instancu grupe pokrivenosti ponaosob. Ovo bi došlo do izražaja da postoji više instanci monitora u verifikacionom okruženju.

7 Tok verifikacije i rezultati prikupljanja pokrivenosti

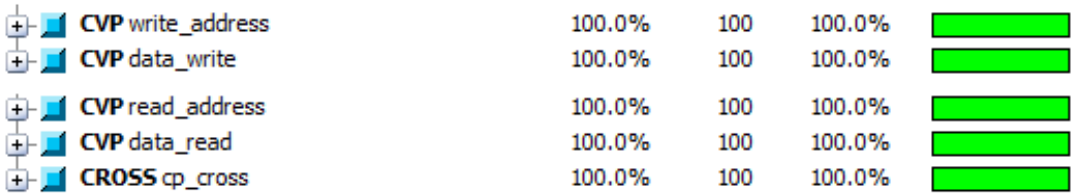
Izvršen je test *test_mlp_simple*. Prvo je bilo neophodno ukloniti pronađene greške u samom verifikacionom okruženju. One su se ticale povezanosti TLM portova i ponašanja *write* funkcija u *scoreboard* komponenti. Nakon toga, uz obezbeđene prateće fajlove sa pikselima slika i parametrima neuronske mreže, pokrenuta je simulacija korišćenjem *QuestaSim* alata. Nije bilo prijavljenih upozorenja niti grešaka, a ispisivane poruke pokazivale su poklapanje rezultata dobijenih od jezgra i referentnog modela. Dobijanje ovakvih ispravnih rezultata je prvi pokazatelj da jezgro na pravilan način tumači i obrađuje podatke dobijene putem AXI interfejsa, kao i da putem istih interfejsa daje odgovore u pravilnom formatu. Naime, monitor ne bi uspeo da prikupi transakcije,a *scoreboard* da ih protumači, da odgovor jezgra nije pratio protokol.

Potom se pristupilo analizi talasnih oblika signala. Na slici 7 vide se AXI Lite signali u trenutku nakon klasifikacije prve cifre. Kada je sa AXI *Lite* adrese 4 (*ready* registar) pročitana 1, poslata je komanda za čitanje sa adrese 12 (*cl_num* registar). Klasifikovani broj se pojavljuje na magistrali *s_axi_rdata* i primećuje se broj 7. Jezgro je aktiviralo signale poput *s_axi_arready*, što znači da je AXI Lite protokol ispoštovan. Vidimo da odmah nakon čitanja rezultata, signal *s_axi_wdata* postavlja na 1, a *s_axi_awaddr* ima vrednost 0 (adresa *start* registra), što znači da započinje nova klasifikacija.

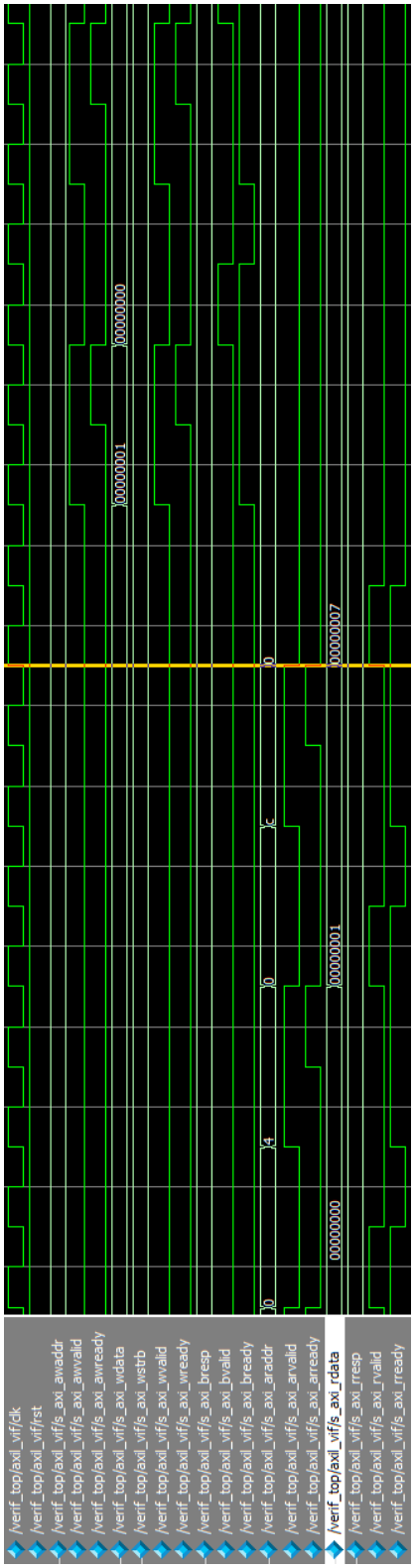
Na isti način, posmatranjem talasnih oblika signala, ustanovljeno je poštovanje AXI *Stream* protokola.

Na samom početku simulacije talasni oblici signala su pokazali da je jezgro resetovano. Pošto je nakon toga nastavilo sa ispravnim klasifikovanjem, to znači da je i funkcionalnost reseta zadovoljena.

Rezultati analize pokrivenosti su prikazani na slici 6. Pokrivenost svih tačaka je 100%, što znači da je test scenario dobro napisan i da je uspeo da izverifikuje funkcionalnost jezgra u željenoj meri.



Slika 6: Rezultati prikupljanja pokrivenosti



Slika 7: Talasni oblici signala od značaja - simulacija

8 Zaključak

U ovom radu je pomoću UVM metodologije uspešno verifikovano IP jezgro koje implementira MLP algoritam za prepoznavanje cifara. MLP IP jezgro je dalo očekivan odziv na primenjene stimuluse i svaka poslata slika je bila klasifikovana na isti način i od strane jezgra, kao i od referentnog modela. Na osnovu simulacionih rezultata i rezultata prikupljanja pokrivenosti zaključeno je da projektovani hardver ispunjava zahteve funkcionalne specifikacije i da su napisani testovi u dovoljnoj meri izverifikovali postavljene zahteve.

U verifikacionom okruženju nisu uočeni propusti. Komponente okruženja pisane su na sistematičan način, nezavisne jedna od druge i od dizajna koji se verifikuje. Komponente verifikacionog okruženja korišćenog u ovom radu se mogu ponovo koristiti u verifikaciji drugih IP jezgara koja poseduju AXI *Lite* ili AXI *Stream* interfejse.

Potencijalna poboljšanja mogla bi biti u pisanju novih, raznovrsnijih testova i pisanju čeker komponenti. Dodavanje *assert* naredbi za praćenje poštovanja AXI protokola moglo bi takođe doprineti kvalitetu verifikacionog okruženja. Pošto MNIST baza podataka u sebi sadrži i oznake cifara na fotografijama, moguće je primeniti i drugi verifikacioni pristup - korišćenjem zlatnih vektora.

Literatura

[1] Vladimir Crnojević. *Prepoznavanje oblika za inženjere.* Fakultet tehničkih nauka, Novi Sad, 2014.

[2] Ivan Nunes da Silva, Danilo Hernane Spatti, Rogerio Andrade Flauzino, Luisa Helena Bartocci Liboni, and Silas Franco dos Reis Alves. *Artificial Neural Networks: A Practical Course.* Springer, 2017.

[3] Andrea Erdeljan. Materijal za vežbe iz predmeta Funkcionalna verifikacija hardvera - vežba 5, 2019.

[4] Andrea Erdeljan. Materijal za vežbe iz predmeta Funkcionalna verifikacija hardvera - vežba 6, 2019.

[5] Michael A. Nielsen. *Neural Networks and Deep Learning.* Determination Press, 2015.

[6] Rastislav Struharik. *Projektovanje složenih digitalnih sistema : računarske vežbe.* Fakultet tehničkih nauka, Novi Sad, 2017.

[7] Bruce Wile, C. John Goss, and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle.* Morgan Kaufmann, San Francisco, 2005.