Funkcionalna verifikacija hardvera

NAZIV PROJEKTA:

Audio Scrambler System

TEKST ZADATKA:

Napraviti verifikaciono okruženje na osnovu kog ce se verifikovati Audio signal skrembler i deskrembler dizajn

MENTOR PROJEKTA:

Nikola Kovačević

PROJEKAT IZRADILI:

Jelena Vujaković EE52/2015

DATUM ODBRANE PROJEKTA:

20.11.2020

Sadržaj

[Funkcionalna verifikacija hardvera 1](#_Toc56779503)

[1. Uvod u funkcionalnu verifikaciju hardvera 3](#_Toc56779504)

[2. Realizacija projekta – Audio Scrambler System 5](#_Toc56779505)

[2.1. Scrambler IP Core 6](#_Toc56779506)

[2.2. Razvoj Linux Drajvera za projektovano jezgro 6](#_Toc56779507)

[2.3. Teorijska analiza Audio Scrambler System modula 7](#_Toc56779508)

[Format .WAV 8](#_Toc56779509)

[Algoritam skemblovanja 8](#_Toc56779510)

[2.4. Implementacija hardvera pomoću Vivado IP integratora 9](#_Toc56779511)

[2.5. Definisanje interfejsa 11](#_Toc56779512)

[2.6. AXI Interfejs 12](#_Toc56779513)

[3. *UVM* metodologija verifikacije projektovanih digitalnih sistema 15](#_Toc56779514)

[3.1. UVM hijerarhija 16](#_Toc56779515)

[3.2. Verifikaciono okruženje 17](#_Toc56779516)

[3.3. Verifikacioni plan scrambler\_ip jezgra 17](#_Toc56779517)

[3.4. Top testbenč 18](#_Toc56779518)

[4. *UVM* environment klasa 19](#_Toc56779519)

[4.1. BRAM A agent 19](#_Toc56779520)

[Standardne *UVM* faze BRAM A *agent*a tokom simulacije 20](#_Toc56779521)

[BRAM A driver i BRAM A sekvencer 22](#_Toc56779522)

[BRAM A monitor i scrambler\_ip\_scoreboard 23](#_Toc56779523)

[4.2. BRAM B agent 24](#_Toc56779524)

[BRAM B monitor i scrambler\_ip\_scoreboard 25](#_Toc56779525)

[4.3. Rezultati simulacije BRAM A i BRAM B memorije 26](#_Toc56779526)

[4.4. AXI-LITE agent 26](#_Toc56779527)

[AXI-LITE monitor 27](#_Toc56779528)

[AXI-LITE drajver, sequencer, sequence 27](#_Toc56779529)

[4.5. Virtualni sekvencer 30](#_Toc56779530)

[4.6. Scoreboard 31](#_Toc56779531)

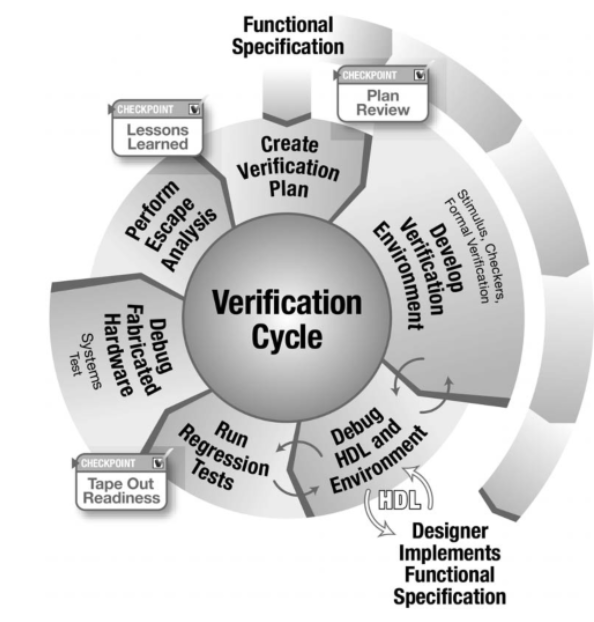
[4.7. Verifikaciona pokrivenost scrambler\_ip jezgra 32](#_Toc56779532)

[5. Literatura 34](#_Toc56779533)

# Uvod u funkcionalnu verifikaciju hardvera

Nakon projektovanja *IP* jezgra potrebno je proveriti da li se njegova funkcionalnost poklapa sa specifikacijom. Cilj verifikacije hardvera je da otkrije i lokalizuje skrivene greske (bagove) ukoliko oni postoje u dizajnu. Drugim rečima, zadatak verifikacije hardvera je da na efikasan način pretraži ogroman prostor stanja i detektuje svako odstupanje od specifikacije. Tekuće stanje određeno je trenutnim sadržajem svih memorijskih elemenata koji se nalaze u dizajnu. Naredno stanje, određeno trenutnim stanjem i trenutnim vrednostima ulaznih portova može biti 2n. Ako se uzme u obzir činjenica da broj mogućih tranzicija u sistemu sa n bita memorije raste kao eksponencijalna funkcija od n i iznosi dolazimo do zaključka da je potrebno podeliti sitem na manje celine. Na taj način sistem postaje modularan,svaki modul se verifikuje posebno, zatim se verifikovani modul spaja u veće celine, pri čemu se takođe proverava korektnost rada.

Važan koncept pri verifikaciji je verifikacioni ciklus. Polazeći od funkcionalne specifikacije verifikacioni ciklus nakon jasno definisanih koraka treba da obezbedi verifikovan dizajn bez bagova. Funkcionalna specifikacija predstavlja formalan opis funkcionalnosti koje sistem treba da poseduje. Sadrži specifikaciju interfejsa koje sistem treba da poseduje, pomoću kojih će komunicirati sa svojim okruženjem, spisak funkcija koje sistem treba da poseduje, performanse ( cena, potrošnja, brzina rada ) koje sistem treba da ispuni.



*Slika 1. Na slici je prikazan tok verifikacije*

Na osnovu funkcionalne specifikacije pravi se detaljan opis verifikacionog procesa koji će biti implementiran, verifikacioni plan. Verifikacioni plan sadrži

1. Opis testova i metoda koje će biti korišćene prilikom verifikacije dizajna
2. Potrebne alate koji će biti korišćeni i procesu verifikacije
3. Uslove za završetak verifikacije
4. Potrebne resuse kao i vremenski redosled izvršavanja procesa verifikacije
5. Spisak funkcija koje će biti verifikovane
6. Spisak funkcija koje neće biti verifikovane

Nakon razvijenog verifikacionog plana pristupa se razvoju verifikacionog okruženja. Verifikaciono okruženje predstavlja skup softverskih modula (testbenčeva) i alata koja verifikacionom timu omogućavaju da pronađe bagove koji su prisutni u dizajnu. Projekat je realizovan na osnovu standardizovane *UVM* metodologije ( *engl. Universal Verificarion Metodology* ). *UVM* metodologija se između ostalog sastoji od pomoćne biblioteke u *SystemVerilog* jeziku. Objektno-orjentisani dizajn, kao glavna karakteristika *SystemVerilog* jezika omogućava lako kreiranje verifikacionih komponenti. Ponovna upotreba testbenčeva i jednostavno kreiranje verifikacionih komponenti su doprineli su širokoj upotrebi ove metodologije.

Univerzalna verifikaciona komponenta *UVC* ( *engl. Universal Verification Component*) predstavlja komponentu koja enkapsulira osnovne podkomponente kao što su monitor, drajver, sekvencer. Zahvaljujući takvom načinu grupisanja podkomponenti u jednu komponentu moguće je na jednostavan način prilagoditi i ponovo iskoristiti već projektovan *UVC* kao nezavisnu komponentu koja će biti deo većeg, kompleksijeg sistema. Ponovnom upotrebom manjeg, već projektovanog sistema štedi se na vremenu koje bi bilo utrošeno ukoliko bi se komponenta ponovo projektovala. *UVM* obezbeđuje framework za verifikaciju zasnovan na funkcionalnoj pokrivenosti (*engl. Coverage Driven Verification,* *CDV* ). Na osnovu zadatih parametara dobija temeljna verifikacija.

U prvom uvodnom poglavlju date su osnovne informacije o realizaciji projekta. Ukratko je prikazan kontekst toka verifikacije digitalnih sistema, osnovne informacije o korišćenoj metodologiji, takođe je dat pregled korišćenih softverskih alata kao i razlog uvođenja jezika za verifikaciju hardvera.

U drugom poglavlju detaljnije je analizirana realizacija projekta. Prikazan je način realizacije algoritma. Analizirani su koraci specifikacije koji su doveli do realizacije *HDL* modela i scrambler\_ip jezgra.

U trećem poglavlju dat je prikaz realizacije projekta na osnovu *UVM* metodologije. Objašnjena je *UVM* hijerarhija i verifikaciono okruženje. Dat je prikaz verifikacionog plana i prikaz hijerarhije na najvišem nivou.

U četvrtom poglavlju objašnjen je način implementacije verifikacionog plana i verifikacionog okruženja. Detaljno su opisane komponente koje čine verifikaciono okruženje kao i podkomponete od kojih su sastavljeni. Takođe je opisan način rada svake komponente.

U petom poglavlju navedena je korišćena literatura.

# Realizacija projekta – Audio Scrambler System

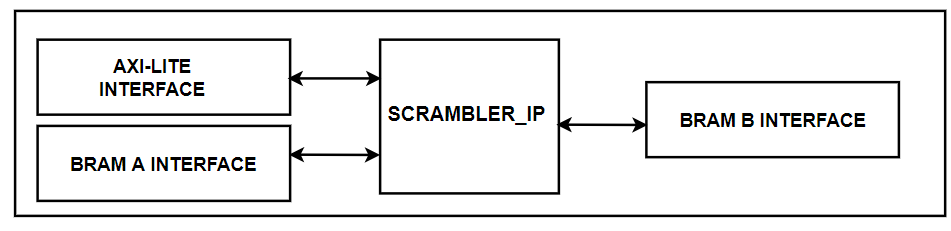
Početna tačka realizacije projekta je izvršna specifikacija, u kojoj je definisana funkcionalnost čitavog sistema, napisana u *Matlab* i *C++* programskom jeziku. Cilj implementacije algoritma u *C++* programskom jeziku je da se algoritam lakše prilagodi SystemC standardnom jeziku za modelovanje na nivou sistema. Jezik SystemC čini *C++* biblioteku koja uključuje mogućnost konkurentnosti i komunikaciju baziranu na transakcijama što dovodi do povećanja produktivnosti.

Tehnikom profajliranja testirana je izvršna specifikacija projekta. Pomoću *Callgrind [[1]](#footnote-1)* alata unutar *Valgrind* [[2]](#footnote-2) distribucije izvršeno je profajliranje funkcija koje služi za otkrivanje i optimizaciju najčešće pozivanih funkcija. Cilj profajliranja je prikupljanje podataka o procentu utrošenog vremena u bloku koda. Blokovi koda mogu biti funkcije ili deo kod u kome se nalaze instrukcije grananja ili skokova. Takođe, bitan podatak koji se dobija pomoću *Memcheck* alata unutar *Valgrind* disribucije je podatak o alokaciji memorije na osnovu kog je utvrđeno da nema curenja memorije, neispravnog oslobađanja *heap* memorije, pristupanja memoriji koja je već oslobođena tokom izvršavanja programa.

Na osnovu analize rezultata profajliranja sakupljeni su podaci o broju pozvanih funkcija, broju izvršenih instrukcija čime je utvrđeno da funkcije u kojima se vrši operacija skremblovanja i deskreblovanja podataka troše najviše vremena. Nakon toga je izvršena podela sistema na hardversku i softversku komponentu. U okviru hardverske komponente implementiran je algoritam skremblovanja i deskremblovanja podataka. Softverska komponenta modeluje ponašanje procesora. Model celokupnog elektronskog sistema modelovan je u SystemC standardnom jeziku za modelovanje na nivou sistema. Sastoji se od procesora, *IP* jezgra i podsistema za povezivanje. Komponenta za povezivanje na osnovu traženih adresa prosleđuje transakciju memoriji, koja se u ovom delu projekta nalazi u okviru procesora, ili *IP* jezgru. Za povezivanje u okviru sistema koristi se *TLM* [[3]](#footnote-3) standard koji ceo sistem realizuje transakcijama pomoću standardnih *TLM* interfejsa. Cilj modelovanja celokupnog elektronskog sistema je razvoj virtualne platforme. Virtualna platforma predstavlja softverski model čija funkcionalnost je identična hardverskom prototipu koji je tokom dalje realizacije projekta razvijen. Nakon kreiranog softverskog modela virtualne platforme projektovano je *IP* jezgro namenjeno implementaciji hardverske komponente sistema. Naredni korak je razvoj *Linux* drajvera za projektovano IP jezgro i kreiranje aplikacije za testiranje rada drajvera. Zatim je projektovano *IP* jezgro verifikovano pomoću *UVM* metodologije.

## Scrambler IP Core

*Scambler IP* jezgro komunicira sa dve spoljašnje *BRAM* (*eng. Block Random Access Memory*) memorije. U *BRAM* memoriju smeštaju se adrese na kojima se nalaze podaci koji predstavljaju obirke audio signala koji se obrađuje. Izbor sistemske magistrale zavisi od procesorskog jezgra koje se nalazi na razvojnoj ploči. U realizaciji ovog projekta korišćena je *Zybo* *Zynq* razvojna ploča koja se sastoji od dvojezgarnog *ARM Cortex-A9* procesora u procesorskom delu razvojne ploče. *ARM* procesori direktno podržavaju *AMBA* ( *engl. Advanced Mictocontroller Bus Architecture* ) ili *AXI* sistemske magistrale. Komunikacija između *IP* jezgra, dve spoljašnje *BRAM* memorije i procesora odvija se pomoću standardizovane *ARM* *AMBA* sistemske magistrale. *AXI-Lite* protokol pripada ovoj grupi familije sistemskih magistrala. Na osnovu *AXI-Lite* protokola na jasan način definisana je razmena podataka, komandi i statusnih signala. Procesor je inicijator transakcije, on šalje zahtev za početak rada preko *AXI-Lite* interfejsa. Zatim šalje adresu na kojoj se nalazi podatak. Adresa na kojoj se nalazi podatak smešta se u prvi *BRAM* tačnije u *BRAM* *PORT* A koji predstavlja pristup za čitanje. Oba *BRAM*-a su dvopristupna ( jedan pristup za čitanje i jedan pristup za pisanje). Zatim se podaci šalju dok se ne popuni kapacitet prve *BRAM* memorije, koji se sastoji od 32768 memorijskih lokacija. U svakoj transakciji šalje se po jedan podatak koji predstavlja jedan odbirak i sastoji se od 4 bajta. Algoritam skremblovanja i deskremblovanja je implementiran nad blokom od 8192 obirka.



*Slika 2. Šematski prikaz implementiranog IP jezgra*

## Razvoj Linux Drajvera za projektovano jezgro

*Scrambler IP* predstavlja platformski uređaj koji je neophodno povezati sa procesorom. Procesor nema mogućnost automatskog detektovanja *scrambler\_ip* jezgra sa kojom komunicira preko *AXI* interfejsa. Implementirani drajver sakriva detalje funkcionisanja uređaja i sadrži informacije o hardveru koji se nalazi na ploči ( na koju magistralu je priključen uređaj, širina adresne magistrale, imena i broj uređaja pomoću kog kernel povezuje uređaj sa drajverom ). Na taj način drajver se može posmatrati kao softverski sloj koji se nalazi između uređaja koji se nalazi u kernel prostoru i aplikacije koja se nalazi u korisničkom prostoru. *BRAM A, BRAM B* memorija i *scrambler\_ip* predstavljaju periferije (uređaje) koji se povezuju sa drajverom. Aplikacija testira ove uređaje tako što šalje podatke u skladu sa specifikacijom. Postavlja vrednosti AXI registara, šalje podatke iz uređaja *BRAM A* u *scrabler\_ip* jezgro, koji šalje rezultate obrade u *BRAM B* memoriju. Na osnovu poređenja podataka aplikacija vrši testiranje rada drajvera i čitavog sistema. Deskremblovani podaci trebaju biti identični ulaznim podacima.

## Teorijska analiza Audio Scrambler System modula

Hardverska implementacija sistema izvršena je korišćenjem *FPGA* programabilnih kola. Pomoću *HDL* modela sistema i *Vivado Design Suite* alata za automatsku sintezu projektovano je *IP* jezgro koje predstavlja modul koji implementira algoritam. Digitalna obrada signala ( engl. *Digital Signal Processing, DSP* ) predstavlja obradu digitalno reprezentovanih signala. Obzirom da bi analiza, projektovanje i sinteza digitalnih sistema koji obrađuju digitalne signale bila vrlo komplikovana u praksi se sistemi za digitalnu obradu projektuju kao diskretni sistemi. Obrada audio signala vrši se pomoću diskretnih signala. Nakon što se projektuje odgovarajući diskretni sistem, efekti kvantizacije vrednosti diskretnih signala analiziraju se odvojeno u fazi implementacije sistema. Diskretni sistem predstavlja numerički algoritam koji transformiše vremenski diskretni ulazni signal x(n) ( koji je predstavljen kao sekvenca brojeva konačne dužine ) u vremenski diskretan izlazni signal y(n) ( koji je takođe predstavljen kao sekvenca brojeva konačne dužine ). Algoritam realizovan u ovom projektu predstavlja diskretni sistem bez memorije, jer trenutna vrednost izlazng signala zavisi samo od ulaznog signala. Relacija ulaz-izlaz diskretnog signala ima sledeći oblik:

y(n) = H{x (n)} , H označava matematički operator koji opisuje rad sistema

Da bi digitalna elektronska kola mogla obrađivati signale oni moraju imati amplitude predstavljene sa konačnom preciznošću, koristeći formate fiksne ili pokretne tačke. Digitalni signali su predstavljeni pomoću niza logičkih vrednosti, bita, koje odgovaraju ciframa binarne reprezentacije posmatrane numeričke vrednosti. Binarni vektor b1, b2 .. bB-1, bb, gde B binarnih cifara, bita uzima vrednosti 0 ili 1 može se iskoristiti za reprezentaciju celobrojne binarne vrednosti iz skupa [0, 1, ..., 2B-1]

D = b12B-1 + b22B-2 + ... + bB-121 + bB20

Fizička implementacija diskretnih sistema može se realizovati softverski ili hardverski. Ovi sistemi predstavljaju digitalna elektronska kola koja su projektovana da manipulišu vrednostima reprezentovanim u digitalnom obliku pomoću binarnih električnih signala.

Spregu između analognog i digitalnog signala čine dva procesa. Proces odabiranja vrši se konverzija vremenski kontinulanog signala u diskretni signal, odabirajući vrednosti kontinulanog signala u regularnih intervalima vremena. Drugi korak je proces kvantizacije. Ovim procesom vrši se konverzija kontinualne amplitude x u diskretnu amplitudu xd. Nakon ova dva koraka na izlazu A/D konvertora pojavljuje se digitalni signal xd[n], koji se od diskretnog x[n] signala razlikuje po tome što je njegova amplituda diskretizovana na 2B vrednosti.

### Format .WAV

*WAV* *( engl. Waveform Audio Format )* je matični format zvuka u *Windows* operativnom sistemu. Ovaj format baziran je na *RIFF* specifikaciji koja govori o načinu skladištenja multimedijalnih datoteka. Svaka *RIFF* kompatibilna datoteka započinje sa zaglavljem nakog kog slede sekvence sastavljene od blokova podataka. *WAV* datoteka je najčešće *RIFF* datoteka sa jednim *WAV* blokom koji se sastoji od dva manja bloka. U prvom bloku se vrši specifikacija formata podataka dok se u drugom bloku zapisuju zvučni uzorci, semplovi. Pomoću *WAV* datoteke moguće je zapisati jednokanalni ili višekanalni (mono/stereo), različitim rezolucijama (npr. 8 ili 16 bita ) i različitim frekvencijama uzrokovanja ( npr. 44.1 kHz ).

Fajl je sačuvan u nekompresovanom *WAV* formatu zvučnog zapisa. Ovaj format ima svoju metodologiju na osnovu koje kreira strukturu datoteke u kojoj organizuje bajtova digitalizovanog zvuka u datoteku podataka. Svaki deo sekunde kvantizer zvuka uzima i snima kao digitalnu informaciju u bitovima i bajtovima. Kvalitet ovog digitalnog zapisa zavisi od toga koliko često se uzimaju uzorci ( brzina kvantizovanja ili frekvencija, merena u kilohercima kHz ili hiljadama uzoraka u sekundi ) i sa koliko brojeva se prezentuje vrednost svakog uzorka ( veličina uzorka, bitdepth, rezolucija ili dinamički opseg ). Što se češće uzima uzorak i što se više podataka zapamti o uzorku, biće finija rezolucija i veći kvalitet zvuka. Frekvencija korišćena u realizaciji ovog projekta je 44,1 kHz, 16 kHz, 8 kHz uzoraka u sekundi. Što je veća veličina uzorka, prikupljeni podaci če bolje da opišu snimljeni zvuk. Osmobitni uzorak daje 256 jednakih jedinica za opis dinamičkog raspona ( amplitude ) nivo zvuka u tom trenutku od dela uhvaćenog zvuka. Šesnaestobina veličina uzorka daje 65 536 jednakih jedinica za opis dinamičkog raspona. Delovi analognog talasnog oblika uzorkuju se u različitim frekvencijama, a svaki diskretni uzorak je zapamćen kao 16 bita podataka. Vrednost svakog uzorka se zaokružuje na najbliži ceo broj ( kvantizacija ).

### Algoritam skemblovanja

*Audio scrabler system* je realizovan na osnovu algoritma koji se sastoji iz dve faze, faze skremblovanja i faze deskrembovanja. Razlika između faze skremblovanja i faze deskremblovanja je u tome što se process skremblovanja radi nad učitanim ulaznim odbircima audio zapisa, dok se process deskremblovanja vrši nad prethodno skremblovanim odbircima. Nakon procesa deskremblovanja odbirci treba da budu identični ulaznim odbircima.

Ovako osmisljen algoritam sastoji se iz faze skremblovanja i faze deskrembovanja. Faza deskremblovanja je inverzna fazi skremblovanja. Skremblovanje i deskremblovanje u algoritmu implementirano je nad blokovima od 8192 odbirka. Čitav signal zvuka nakon učitavanja podeljen na blokove od 8192 odbirka koji su unutar sebe podeljeni na blokove od 128 delova koji sadrze 64 odbirka, zatim se 64 odbirka dele na 16 blokova od 4 odbirka. Skremblovanje je urađeno tako što su prva 64 odbirka u bloku od 8192 odbirka zamenjena sa zadnjih 64 odbirka, drugih 64 odbirka su zamenjena sa predzadnja 64 odbirka i tako dalje sve dok ne dodje do zamene svih odbiraka u jednom bloku po tom principu. Dalje se vrši skremblovanje svakog dela od 64 odbirka. Skremblovanje se vrši na način da se svaki blok od 64 odbirka podeli na 16 blokova po 4 odbirka. Prvi blok od 4 odbirka se zamenjuje sa zadnja 4 odbirka u bloku od 64 odbirka, drugi blok od 4 odbirka se zamenjuje sa predzadnja 4 odbirka u bloku od 64 odbirka. Radi se po tom principu dok se ne izvrši premeštanje svih odbiraka u svim blokovima.

## Implementacija hardvera pomoću Vivado IP integratora

*ZedBoard* ploča bazirana na *Xilinx Zynq-7000 All Programmable SOC ( AP SoC )* korišćena je u realizaciji ovog projekta. Za konfigurisanje multiprocesorskog *ARM* sistema *Dual Cortex-A9* unutar procesorskog sistema čipa korišćen je *Zynq7 PS* blok u okviru *Vivado IP Integrator alata*. *Zynq7 PS* blok sadrži podrazumevane pristupe pomoću kojih komponenta komunicira sa ostatkom sistema. *FIXED\_IO* pristup služi za komunikaciju FPGA čipa sa spoljnim komponentama. Ovaj pristup sadrži *UART ( Universal Asynchronous Receiver Transmitter )* komunikaciju. Pri implementaciji softvera na ploči korišćen je *UART* terminal za ispis. Pomoću M\_*AXI*\_GPO pristupa koji predstavlja *AXI* magistralu ostvaruje se komunikacija između procesorskog dela čipa i programabilnog dela čipa. Za komunikaciju sa *DDR* memorijom koristi se *DDR* pristup. Na *M\_AXI\_GPO\_CLK* pristup dovodi se sinhronizacioni signal. Sinhronizacioni signal za programabilnu logiku koja se nalazi na čipu dovodi se preko *F\_CLK\_CLKO* pristupa, dok se reset dovodi preko *F\_CLK\_RESETO\_N* pristupa.

Unutar programabilnog dela čipa nalazi se projektovano scrambler\_*IP* jezgro, *AXI Interconnect* i komponente koje predstavljaju *BRAM* *A* i *BRAM* *B* memoriju *Block Memory Generator* ( *axi\_bram\_ctrl0 i axi\_bram\_ctrl1*), *AXI Block RAM Controller-a* ( *blk\_mem\_gen\_ctrl0 i blk\_mem\_gen\_ctrl1* ).

*Block Memory Generator IP* koristi već postojeće memorijske blokove ( *Block Memory )* koji se nalaze na programabilnom delu *FPGA* čipa. Ovaj *IP Core* je konfigurisan tako da obe *BRAM* memorije imaju interfejs za čitanje ( *PORT A* ) i interfejs za upis ( *PORT B* ). Kompatibilan je sa *AXI* Lite protokolom. Parametar koji definiše dubinu memorije postavljeni su u skladu sa specifikacijom na 32764 memorijske lokacije, dok je parametar za širinu postavljen na 32 bita u skladu definisanim *AXI* Lite interfejsom.

Komponente koje omogućavaju integraciju *BRAM* A i *BRAM* B memorije u sistem koriste instance *AXI Block RAM Controller-a*. *Xilinx Vivado Design Suite* sadrži *AXI Block RAM (BRAM) Controller IP* jezgro. Pomoću ovog *IP* jezgra ostvaruje se komunikacija sa *BRAM* memorijom. *AXI Block RAM (BRAM) Controller IP* je konfigurisan za rad sa True Dual *PORT* *BRAM* A i *BRAM* B memorijom . Kompatibilan je sa *AXI-Lite* interfejsom. Kompatibilnost se ogleda u činjenici da se komunikacija sa *AXI master* komponentama ostvaruje se pomoću pet kanala koji čine *AXI-Lite* interfejs. U zavisnosti od tipa transakcije koji je iniciran aktivira se u slučaju upisa *Write Address Channel,* zatim *Write Data Channel* i *Write Response Channel.* U slučaju čitanja aktivira se *Read Address Channel* i *Read Data Channel* kao odgovor. Pomoću *AXI\_wvalid* i *AXI\_rvalid* signala *AXI* *Block RAM (BRAM) Controller IP* kontroliše tip operacije i proverava da *read* i *write* operacije nikad nisu istovremeno aktivirane. *Block RAM (BRAM) Controller IP* implementiran je kao *AXI slave IP* koji je povezan sa *AXI Interconnect* modulom.

*AXI Interconnect* modul se nalazi unutar sistema omogućava komunikaciju memorijski mapiranih *AXI master* i memorijski mapiranih *AXI slave* komponenti. U *AXI master* komponente spada *Zynq Processing System* *processing\_system7\_0* dok *AXI slave* komponente čine *scrambler\_axi\_ip, blk\_mem\_gen0, blk\_mem\_gen1, axi\_bram\_ctrl0 i axi\_bram\_ctrl1*. *AXI master* komponeta u ovom slučaju procesor je inicijator transakcija, dok *AXI slave* komponente služe kao komponente koje treba da na odgovarajuć način odgovore na započete transakcije.

*Vivado IP Integrator block design* sadrži *AXI Interconnect core* u okviru *Vivado Design Suite* unutar *Vivado IP* kataloga. *AXI Interconnect core* je konfigurisan tako da podržava *AXI-Lite* protokol. Unutar *AXI Interconnect* komponente nalazi se *AXI Crossbar* komponenta kad u sistemu postoji više od jedne *AXI master* komponente ili više od jedne *AXI slave* komponente.

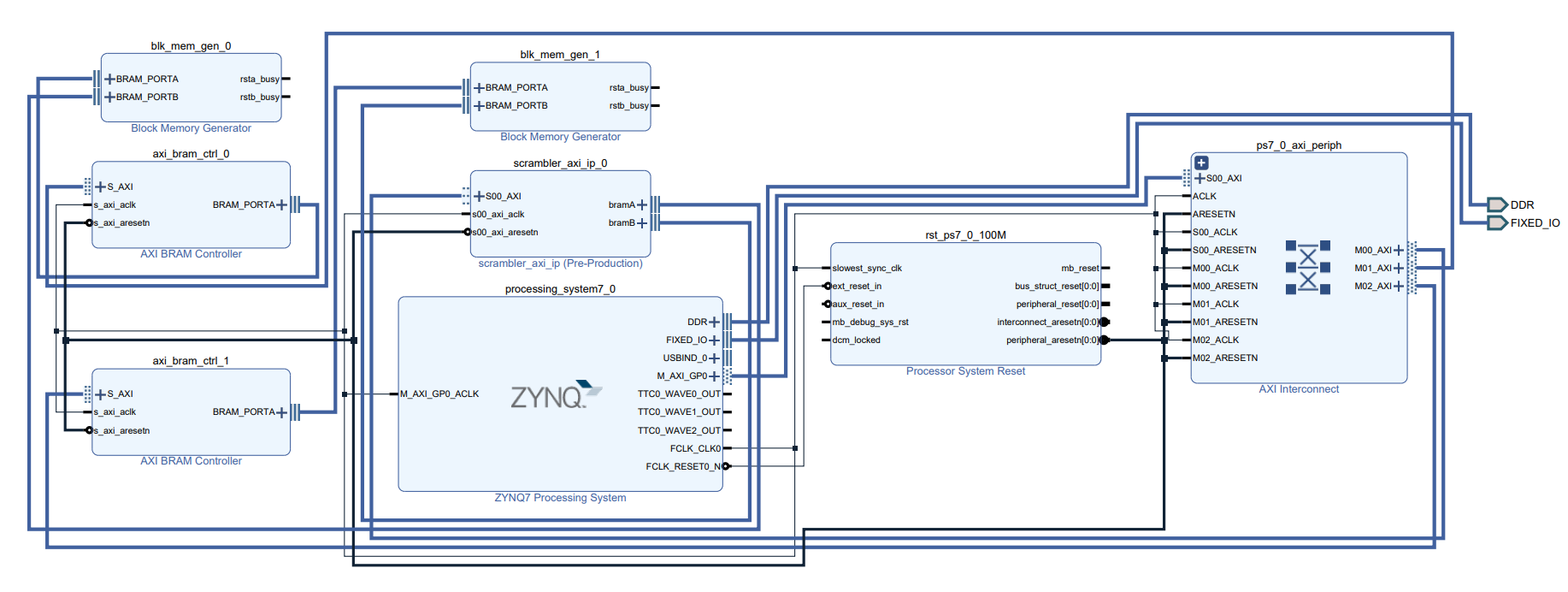
*AXI Crossbar* komponenta je zadužena za rutiranje transakcija između *AXI master* i *AXI slave* komponenti. Analizirajući ciljnu adresu transakcije, koja je deo transakcije, komponenta odlučuje kom od priključenih slejvova je namenjena transakcija i prosleđuje je na odgovarajući port. Funkcionalnosti ove komponente su brojne kao na primer konvertovanje širine podataka, konverzija *AXI4* u *AXI3* protokol, i obrnuto, zadužena je za upravljanje memorijom ( *engl. Memory Menagement Unit, MMU* ).

Način projektovanja *IP* jezgra zasnovan je na *Register Transfer (RT)* metodologiji. *Register Transfer* metodologija predstavlja formalizovan postupak projektovanja digitalnog elektronskog sistema koji implementira izabrani algoritam. Grupisanje signala i njihovo interpretiranje kao posebni tip podataka ( neoznačeni celobrojni tip ili stanje sistema ) dovode do povećanja apstraktnosti reprezentacije podataka na *RT* nivou.

Nakon što je razvijen model celokupnog elektronskog sistema u *SystemC* standardnom jeziku za modelovanje na nivou sistema, naredni korak je projektovanje hardverskog *IP* jezgra.

U skladu sa *RT* metodologijom projektovana su Podsistem za obradu podataka (*engl. Datapath*) i Upravljački podsistem ( *engl. Controlpath*).

Zatim je izvršeno povezivanje *IP* jezgra sa procesorom pomoću *Vivado IP Packager* alata. Implementacija hardvera može se posmatrati kao mapiranje već razvijene virtualne platforme na platformu namenjenu implementaciji, pri čemu se umesto *TLM* magistrala koristi *Advanced Extensible Interface ( AXI ) magistrale*.



*Slika 5. Implementirani sistem*

## Definisanje interfejsa

Projektovano scrambler*\_ip* jezgro sastoji se od *AXI-LITE* interfejsa i *BRAM* interfejsa. Komponente unutar sistema koje sadrže *BRAM* interfejs komuniciraju preko njega sa *BRAM* memorijom radi učitavanja ulaznih podataka ili radi smeštanja rezultata obrade. Signali koji čine *BRAM* A i *BRAM* B interfejs dati su u nastavku.

*Tabela 1. Signali BRAM A interfejsa*

|  |  |
| --- | --- |
| Naziv signala | Opis signala |
| a\_addr\_o | tipa STD\_LOGIC\_VECTOR (WADDR-1 downto 0) – predstavlja adresu koja se dobija iz prve *BRAM* memorije. |
| a\_en\_o | tipa STD\_LOGIC – predstavlja jednobitni signal koji se dovodi na ulazni *PORT* za selekciju rastućih ivica clk *PORT*a na koje *BRAM* A memorija treba da se aktivira. |
| a\_data\_i | tipa STD\_LOGIC\_VECTOR (WADDR-1 downto 0) – predstavlja podatak koji se nalazi na adresi koja se dobija iz prve *BRAM* memorije. |

*Tabela 2. Signali BRAM B interfejsa*

|  |  |
| --- | --- |
| Naziv signala | Opis signala |
| b\_addr\_o | tipa STD\_LOGIC\_VECTOR (WADDR-1 downto 0) – predstavlja adresu koja se šalje iz jezgra ka *BRAM* B memoriji. |
| b\_wr\_o | tipa STD\_LOGIC\_VECTOR (WADDR-1 downto 0) – predstavlja podatak koji se nalazi na adresi koja se šalje iz jezgra ka *BRAM* B memoriji. |
| b\_data\_o | tipa STD\_LOGIC – predstavlja jednobitni signal koji se dovodi na ulazni *PORT* za selekciju rastućih ivica clk *PORT*a na koje *BRAM* B memorija treba da dozvol upis. |

*Tabela 3. Komandni i statusni signali interfejsa*

|  |  |
| --- | --- |
| Naziv signala | Opis signala |
| Start | tipa STD\_LOGIC – predstavlja jednobitni komandni signal kojim procesor kontroliše početak rada |
| Ready | tipa STD\_LOGIC – predstavlja jednobitni signal koji daje indikaciju kad je jezgro spremno za novu transakciju. |

## AXI Interfejs

*AXI* ( *engl. Advanced Extensible Interface*) magistrale predstavljaju poslednju evoluciju *ARM* *AMBA* ( *engl. Advanced Microcontroller Bus Architecture*) magistrale. Trenutno je aktuelna četvrta verzija *AXI* protokola, *AXI4*, standardizovana 2010. godine. *AXI* sistemska magistrala namenjena je za korišćenje unutar SoC sistema visokih performansi, koji rade na visokim učestanostima. Neke od karakteristika *AXI* komunikacionog protokola:

* Adresne i kontrolne faze odvojene su od faze prenosa podataka
* Kontroni signali validnosti pojedinačnih bajtova prilikom prenosa podataka (Byte Strobes) omogućavaju neporavnate prenose podataka
* Podržan je prenos podataka u blokovima (*Burst-based Transaction*)*,* pri čemu je neophodno specificirati samo početnu adresu bloka
* Kanali za upis i čitanje podataka su razdvojeni, omogućavajući jednostavnu implementaciju *DMA* prenosa
* Inicirane transakcije mogu se kompletirati u redosledu koji je različit od redosleda u kom su zadate (*Out-of-order Completion*)
* Moguće je jednostavno uvođenje protočne obrade, u cilju zadovoljavanja zahtevanih vremenskih karakteristika sistema

Postoje tri varijante *AXI4* interfejsa (*AXI4-Full, AXI4-Lite, AXI4-Stream*). U realizaciji ovog projekta korišćen je *AXI4*-Lite protokol koji predstavlja pojednostavljenu verziju *AXI4*-*Full* interfejsa. Koristi se za komunikaciju sa memorijski mapiranim modulima, kod kojih ne postoji mogućnost prenosa podataka u blokovima, već je moguće prenositi samo jedan podatak po transakciji. Usled toga su hardverski resursi neohodni za implementaciju *AXI4-Lite* kontrolera znatno manji od resursa potrebnih za implementaciju *AXI4-Full* kontrolera, kod kojih je podržan prenos podataka u blokovima od maksimalno 256 transfer ciklusa. Takođe je pojednostavljen I sam interfejs, jer je dobar deo signala iz *AXI4*-*Full* interfejsa nepotreban.

*AXI-Lite* interfejsi sastoje se iz sledećih pet kanala:

1. Adresnog kanala za čitanje (*Read Address Channel*)
2. Adresnog kanala za upis (*Write Address Channel*)
3. Kanala pročitanih podataka (*Read Data Channnel*)
4. Kanala upisanih podataka (*Write Data Channel*)
5. Statusog kanala upisa (*Write Response Channel*)

Sve tri varijante *AXI* interfejsa *AXI* kompatibilne periferije dele se u dve grupe: *AXI* *master*e i *AXI* slejvove. *AXI* *master*i iniciraju transakciju, dok *AXI* slejvovi mogu samo da odgovaraju na započetu transakciju.

*Tabela 4. Signali AXI-Lite interfejsa*

|  |  |
| --- | --- |
| Naziv signala | Opis signala |
| s\_axi\_clk | Sinhronizacioni signal AXI interfejsa |
| s\_axi\_aresetn | Reset signal AXI interfejsa |
|  | Adresni kanal za upis |
| s\_axi\_awaddr | Početna adresa bloka od koje će biti upisivani podaci |
| s\_axi\_awprot | Vrsta pristupa. Retko se koristi |
| s\_axi\_awvalid | Indikacija od validnosti adrese. Kada je visok, trenutna adresa je validna |
| s\_axi\_awready | Indikacija da je AXI ***slave*** spreman da prihvati adresu i ostale kontrolne signale iz kanala |
|  | Kanal upisanih podataka |
| s\_axi\_wdata | Podatak koji treba upisati |
| s\_axi\_wstrb | Signal validnosti pojedinačnih bajtova unutar podatka koji se upisuje. Svaki bit je koji je postavljen znači da je njemu pridužen bajt u podatku validan |
| s\_axi\_wvalid | Indikacija validnosti podatka koji se upisuje. Kada je visok trenutni podatak je validan |
| s\_axi\_wready | Indikacija da AXI ***slave*** spreman da prihvati podatak |
|  | Statusni kanal upisa |
| s\_axi\_bresp | Status kompletirane transakcije upisa |
| s\_axi\_bvalid | Indikacija da je podatak u statusu transakcije validan |
| s\_axi\_bready | Indikacija da je AXI ***master*** spreman da prihvati podatak o statusu |
|  | Adresni kanal za čitanje |
| s\_axi\_araddr | Početna adresa bloka od koje će biti čitani podaci |
| s\_axi\_arprot | Vrsta pristupa. Retko se koristi |
| s\_axi\_arvalid | Indikacija validnosti adrese |
| s\_axi\_arready | Indikacija da je AXI ***slave*** spreman da prihvati adresui ostale kontrolne signale iz kanala |
|  | Kanal pročitanih podataka |
| s\_axi\_rdata | Podatak koji je pročitan |
| s\_axi\_rresp | Status kompletirane transakcije čitanja |
| s\_axi\_rvalid | Indikacija validnosti podatka koji se čita. Kada je visok trenutni podatak je validan |
| s\_axi\_rready | Indikacija da je AXI ***master*** spreman za prihvati podatak |

Na slici 3. prikazana je transakcija upisa podatka korišćenjem *AXI-Lite* interfejsa. Aktivna su tri kanala, adresni kanal za upis, kanal upisanih podataka i statusni kanal. *AXI* *master* inicira transakciju upisa podatka tako što šalje adresu podatka koji će biti upisan. Kad *AXI* *slave* prihvati ovu informaciju, *AXI* *master* šalje podatak preko kanala upisanih podataka. Nakon čto *AXI* slejv završi upis, *AXI* slejv šalje informaciju o završenom stanju operacije upisa *AXI* *master*u preko statusnog kanala upisa.



*Slika 3. Transakcija upisa podatka korišćenjem AXI-Lite interfejsa*

Na slici 4. prikazan je primer transakcije čitanja podatka korišćenjem *AXI-Lite* interfejsa. Prilikom čitanja aktivna su samo dva kanala, adresni kanal za čitanje i kanal pročitanih podataka. Statusni kanala je suvišan. *AXI* *master* inicira transakciju čitanja podatka tako što pošalje adresu podatka koji će biti pročitan. Kad *AXI* slejv prihvati ovu informaciju, počinje sa slanjem podatka ka *AXI* *master*u, koristeći kanal pročitanih podataka. Kad *AXI* *master* prihvati podatak, transakcija čitanja je kompletirana.



*Slika 4. Transakcija čitanja podatka korišćenjem AXI-Lite interfejs*

# *UVM* metodologija verifikacije projektovanih digitalnih sistema

*UVM* je standardizovana metodologija koja omogućava razvoj verifikacionog okruženja pomoću kog se utvrđuje ispravna funkcionalnost sistema, što je cilj ovog projekta. *UVM* biblioteka sastoji se od *UVM* baznih klasa na osnovu kojih se razvijaju komponente koje čine verifikaciono okruženje. Prednost *UVM* metodologije je mogućnost korišćenja randomizacije. Randomizacija se između ostalog može vršiti nad podacima koji čine stimulus.Druga prednosti *UVM* metodologije mogućnost korišćenja ograničenja nad podacima, adresama, tipovima transakcija koje se generišu ( operacija za čitanje ili operacija za upis ). Veoma bitna prednost koju pruža korišćenje *UVM* metodologije koja predstavlja jedan od parametara uspešnosti plana verifikacije je podrška za funkcionalnu pokrivenost ( *engl. Coverage* ).

*UVM* metodologija sastoji se od pomoćne biblioteke u *SystemVerilog* jeziku. *SystemVerilog* jezik koji je nastao kao nadogradnja na *Verilog 2001* jezik, kombinuje mogućnosti jezika za opis hardvera ( *VHDL* i *Verilog* ) sa mogućnostima koje pružaju programski jezik *C* i *C++.* Posebnu fleksibilnost *UVM* metodologiji daju osobine objektno-orjentisanog *C++* programskog jezika.

Koncept *UVM* fabrike *( engl. UVM factory* ) predstavlja aspekat objektno-orjentisanog programiranja. *Factory* metoda je jedna od osnovnih softverskih dizajn šablona ( engl. Design pattern ). Da bi se verifikaciono okruženje na jednostavan način prilagodilo zahtevima verifikacionog plana *UVM factory* omogućava jednostavan način zamene postojeće klase izvedenom klasom bez menjaja ostatka okruženja koristeći mehanizam *factory override*. Svodi se na mogućnost objekta da instancira drugi objekat. Da bi se na jednostavniji način moglo manipulisati sa objektom, na primer da bi se kreirao objekat na različite načine, a zatim i instancirao sa određenim imenom i određenim tipom potrebno je da se objekat registruje kao deo *UVM* fabrike. U ovom projektu komponente od kojih se sastoji verifikaciono okruženje predstavljaju dinamičke objekte koji nastaju kao instance klase tokom simulacije. Pri tome, svi objekti su registrovani i čine deo *UVM* fabrike. Da bi se izbegli dodatni koraci prilikom registrovanja *UVM* objekti i komponente koriste *factory registration makroe* za registraciju. U zavisnosti od tipa parametra kog je potrebno registrovati postoje četiri vrste makroa. U ovom projektu korišćeni su makroi za UVM komponente i *UVM* objekte bez parametrizacije.

1. ’uvm\_component\_utils(name) – za *UVM* komponente
2. ’uvm\_component\_param\_utils(name) – za parametrizovane  *UVM* komponente
3. ’uvm\_object\_utils – za *UVM* objekte, *field* makroi za polja u klasi
4. ’uvm\_object\_param\_utils(name) – za parametrizovane *UVM* objekte

## UVM hijerarhija

Hijerarhija verifikacionog okruženja određena je *UVM* metodologijom. Na osnovu *UVM* metodologije svaka komponenta ima tačno definisanu ulogu, način povezivanja sa ostalim komponentama i mesto u hijerarhiji verifikacionog okruženja. *UVM* *agent* enkapsulira komponente kao što su drajver, monitor, sekvencer u jednu komponentu. Ovakav način implementacije dovodi do jednostavnije ponovne upotrebe *UVM* *agent* komponente, kao i do apstraktnog pristupa interfejsu. Verifikacione komponente čine *UVM* testbenč. Hijerahija testbenča je jasno definisana za svaku verifikacionu komponentu. Svaka verifikaciona komponenta ima specifičnu ulogu u hijerarhiji, nalazi se u određenom delu hijerarhije, povezana je na način određen *UVM* metodologijom sa ostalim komponentama, komunicira sa ostalim komponentama takođe prema pravilima određenim *UVM* metodologijom. Realizacija testbenč-a se zasniva na činjenici da komponente na višem nivou hijerarhije obuhvataju komponente na nižem nivou. Na najvišem nivou hijerarhije nalazi se test koji na osnovu baznog testa instacira i konfiguriše okruženje dok se na osnovu jedinstvenog testa pokreće redosled sekvenci, prateći ograničenja i randomizaciju verifikaciono okruženje testira određeni dizajn ( *engl. Design Under Test, DUT* ).

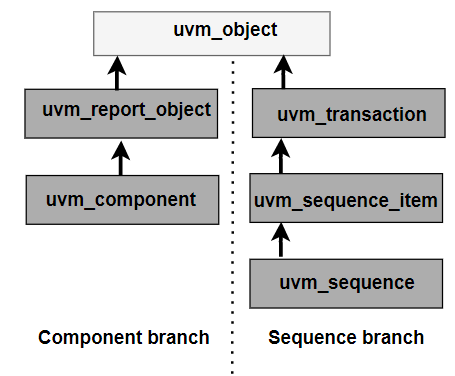
Na slici 2. prikazana je hijerarhija testbenča koji je namenjen verifikaciji *Audio Scrambler System* dizajna. Kao što je već rečeno na najvišem vrhu hijerarhije nalazi se test, njegovim pokretanjem instacira se verifikaciono okruženje koje obuhvata komponente kao što su *BRAM* A, *BRAM* B i *AXI-Lite* *agent*i, *scoreboard* komponenta, virtualni sekvencer. Instanciranjem *agent* komponente instanciraju se monitor, drajver i sekvencer komponente. Veoma bitna komponenta je konfiguracioni objekat koji sadrži podatke kao što su broj agenata, način konfigurisanja *agent*a ( da li je *agent* pasivan ili aktivan ), da li je potreban *master* ili *slave*. Drugim rečima, na osnovu konfiguracionog objekta na jednom mestu definišemo izuzetno bitne informacije o konfiguraciji testbenča. Ovakav način implementacije doprinosi tome da je verifikaciono okruženje veoma lako prilagoditi nekom drugom dizajna koji se verifikuje.



*Slika 6. UVM hijerarhija testbenča za verifikaciju Audio Scrambler System dizajna*

## Verifikaciono okruženje

Tehnike objektno-orjentisanog programiranja olakšavaju kreiranje testbenča. Nasleđivanje kao jedan od ključnih koncepata objektno-orjentisanog programiranja omogućava brži razvoj verifikacionih komponenti. Sve klase nasleđuju *uvm\_object klasu,* koja se nakon nasleđivanja proširuje i prilagođava potrebama verifikacionog procesa. Ovu klasu nasleđuje *uvm\_component* klasa koja sadrži mehanizam faza, mogućnost konfigurisanja i *TLM* interfejse. Komponente koje čine verifikaciono okruženje nasleđuju *uvm\_component* klasu.



Slika 7. Na slici je prikazan pojednostavljen dijagram nasleđivanja u *UVM* metodologiji

Radi bolje kontrole izvršavanja toka testbenča uvodi se pojam *UVM* faza koje predstavljaju sinhronizacioni mehanizam za verifikaciono okruženje. Za kreiranje i konfigurisanje okruženja koristi se *build* faza. *Build* faza se realizuje pozivima funkcija što znači da ona ne troši simulaciono vreme Naredna je *run* faza gde zapravo teče simulaciono vreme. Nakon toga sledi *clean up* faza koja služi za sakupljanje i analizu rezultata. Navedene faze pozivaju virtualne metode koje se nasleđene od *UVM*\_component bazne klase.

Uloga *UVM* testbenča na top-level modulu je da obuhvati i da se poveže sa dizajnom koji se verifikuje ( *DUT* ) kao i da pokrene *run\_test()* metodu pomoću koje se pokreću redom *UVM* faze.

## Verifikacioni plan scrambler\_ip jezgra

Koraci koji su deo verifikacionog plana:

* Provera funkcionalnosti *AXI-LITE* protokola. *DUT* pomoću *AXI-LITE* interfejsa dobija komande na osnovu kojih obavlja svoju funkcionalnost. Potrebno je proveriti da li se signali postavljaju u skladu sa *AXI-LITE* protokolom.
* Potrebno je proveriti da li se funkcionalnost *DUT*-a poklapa sa specifikacijom. Proveriti da li *DUT* ispravno vrši obradu adresa koje šalje *BRAM* A memorija. Provera se vrši u *scoreboard* komponenti.

## Top testbenč

Top-level okruženje korišćeno za verifikaciju *scrambler\_ip* jezgra na najvišem nivou hijerarhije sadrži test klasu. Test klasa sadrži *environment* klasu koja instancira *AXI-Lite*, *BRAM* A i *BRAM* B *agent*e, koji zatim instanciraju svoje podkomponente. *Environment* klasa instancira scoreboard komponentu. Ova komponenta je povezana sa monitorima sva tri *agent*a preko *TLM* interfejsa. Pomoću scoreboard komponente vrši se provera dobijenih i očekivanih rezultata. Pored scoreboarda environment klasa instancira virtualni sekvencer. Ova komponenta zadužena je za pokretanje sekvenci na višem nivou hijerarhije. Virtualni sekvencer sadrži pokazivač na sva tri sekvencera koji se nalaze unutar *AXI-Lite*, *BRAM* A i *BRAM* B *agent*a. Prednost korišćenja virtualnog sekvencera je mogućnost koordinacije pokretanja sekvenci kad u verifikacionom okruženju postoji više aktivnih agenata. Još jedna veoma bitna komponenta koja se nalazi unutar *environment* klase je konfiguraciona klasa. Konfiguraciona klasa sadrži polja koja služe za odabir načina konfigurisanja *agent*a, na primer odabir režima rada, kontrolna polja za dozvolu prikupljanja podataka o pokrivenosti. Ova podešavanja vrše se u posebnom objektu koji se prosleđuje *agent*u koristeći *uvm*\_config\_db. Konfiguracioni objekat se kreira i podešava na višem nivou hijerarhije, zatim se preko baze prosleđuje *agent*u. Preuzimanje iz baze vrši se u *build* fazi.

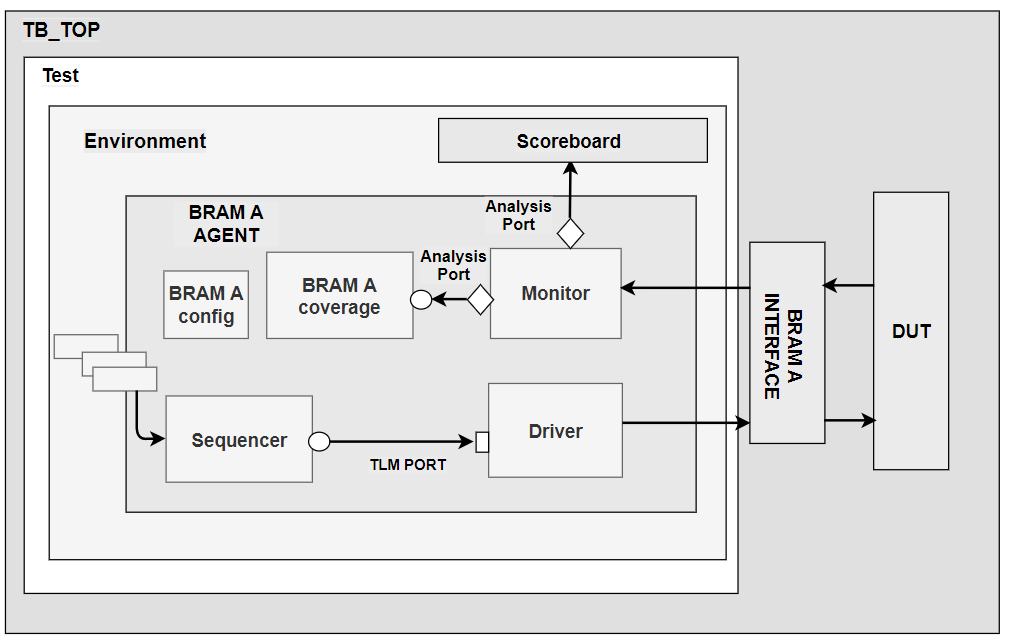


*Slika 8. Struktura UVM testbenča namenjen verifikaciji Audio Scrambler System dizajna*

# *UVM* environment klasa

## BRAM A agent

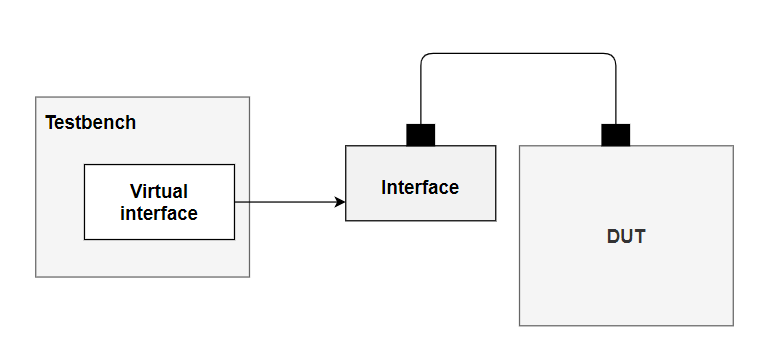
Način rada block RAM memorije simuliran je pomoću *BRAM* *A* *agent* komponente. *BRAM* memorija je konfigurisana kao dvopristupna. Pri čemu u realizaciji ovog projekta se *BRAM* *A* memorija koristi za smeštanje pročitanih podataka. U *BRAM* memoriju se smeštaju odbirci nad kojima je potrebno izvršiti operaciju skremblovanja. Zbog dubine *BRAM* memorije koja iznosi 36864 memorijske lokacije odbirci su grupisani blok od 8192 odbirka. Svaki odbirak je predstavljen sa 32 bita, organizacija memorije bajt adresabilna, iskorišćen je memorijski opseg od 32764 memorijske lokacije za jedan blok od 8192 obirka.



*Slika 9. Na slici je prikazan BRAM A agent sa svojim podkomponentama*

Klase koje čine *BRAM* A *agent* su obuhvaćene u fajlu *bram*\_a\_pkg.sv radi kompaktog korišćenja na višem nivou hijerarhije. *BRAM* *A* *agent* koristi *bram\_a\_if* interfejs koji je implementiran u fajlu *bram\_a\_if.sv*. Verifikacija je realizovana po principu *black\_box*, što znači da prilikom verifikacije nije poznata unutrašnja implementacija dizajna koji se verifikuje već su poznati samo signali na interfejsu dizajna.Za povezivanje dizajna koji je statična komponenta i testbenča koji je dinamičan koristi se virtualni interfejs.

*Port*ovi dizajna povezani su direktno na instancu interfejsa, dok testbenč koristi virtualni interfejs kao pokazivač na interfejs. Na taj način testbenč može indirektno da nadgleda i kontroliše signale na dizajnu, što je neophodno za realizaciju driver i monitor komponente unutar *agent*a.

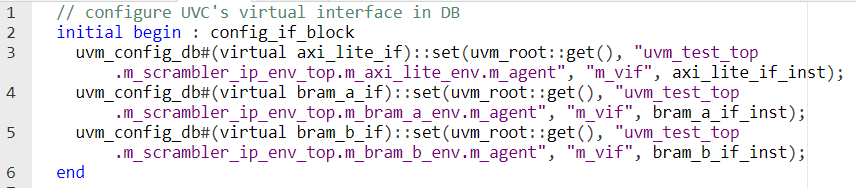


*Slika 10. Pojednostavljen prikaz komunikacije dizajna i testbenča*

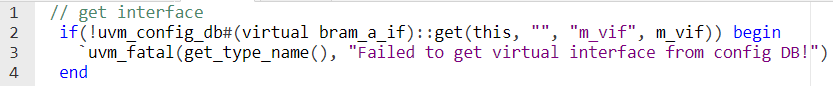
### Standardne *UVM* faze BRAM A *agent*a tokom simulacije

Da bi se mogao generisati stimulus *BRAM* A *agent* kao i ostatak testbenča mora proći kroz *UVM* faze. Prvo se prolazi kroz *build* fazu. *Build* faza funkcioniše tako što kreira i konfiguriše prvo komponente na najvišem nivou hijerarhije, zatim nivo hijerarhije ispod, sve dok ne kreira i konfiguriše komponente *agent*a što je najniži nivo hijerarhije *UVM* testbenča. *BRAM* interfejsu se pristupa preko *BRAM* A *agent* komponente. *BRAM* A *agent* je konfigurisan kao aktivna komponenta koja se sastoji od sekvencera koji šalje podatke drajveru u vidu sekvenci, drajvera koji emulira signale koji se šalju dizajnu i monitora koji nadgleda signale sa interfejsa.

U *scrambler\_ip\_top\_tb.sv* fajlu pre početka testa potrebno je dodeliti pokazivaču koji pokazuje na interfejs pokazivač virtualnog interfejsa pomoću *UVM\_config\_db::set.*

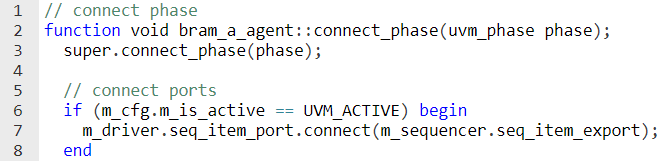


Listing 1. Dodela konfiguracije



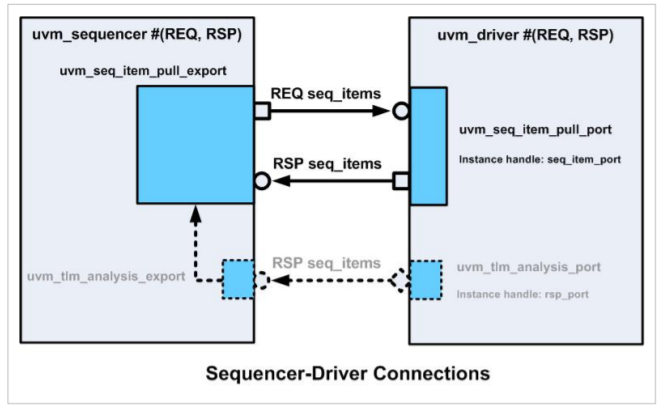
*Listing 2. Preuzimanje pokazivača na virtualni interfejs u bram\_a\_agent.sv*

Pored navedene tri osnovne komponete, *BRAM* *A* *agent* takođe sadrži konfiguracionu klasu koja u *build* fazi na osnovu prosleđenog konfiguracionog objekta kontroliše topologiju *agent*a, drugim rečima kontroliše način na koji su komponente unutar *agent*a instancirane, zatim konfiguracioni objekat definiše pokazivač na virtualni interfejs koji monitor i drajver koriste i određuje da li je *agent* konfigurisan kao pasivan ili aktivan. U slučaju *BRAM* *A* *agent*a promenljivoj unutar *uvm\_agent* klase je dodeljena vrednost *UVM*\_ACTIVE, čime je *BRAM* *A* *agent* konfigurisan kao aktivan *agent*. Nakon završene *build* faza komponente su kreirane. Sledeći korak je povezivanje komponenti. Komunikacija između *UVM* drajver i sekvencer komponente odvija se preko standardnih *TLM* ( *engl. Transaction Level Modeling* ) interfejsa. Na kodnom segmentu izdvojenom iz fajla *bram\_a\_agent.sv* prikazan je način povezivanja deklarisanih Portova koji se nalaze u u *uvm*\_*sequence* baznoj klasi koju nasleđuje *bram\_a\_sequencer* klasa i baznoj klasi *uvm*\_*driver* koju nasleđuje *bram\_a\_driver* klasa. *BRAM* *A* *agent* je konfigurisan kao aktivan *agent*, što znači da *UVM* drajver komponenta komunicira preko svog interfejsa sa *UVM* sekvencer komponentom koja implementira taj interfejs.



*Listing 3. Povezivanje drajver i sekvencer komponente u connect fazi*

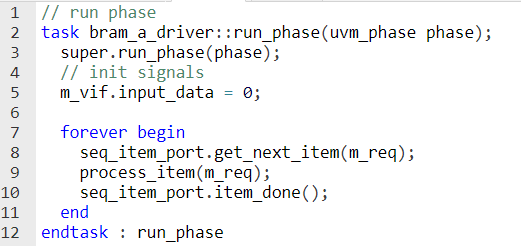
Povezivanje se vrši na osnovu *TLM* konekcija *analysis\_port* od monitora i *analysis\_port* od *agent*a, zatim pošto je *agent* aktivan povezuju se sekvencer i drajver tako što se povezuje *seq\_item\_pull*\_ex*port* sa *seq\_item\_pull\_port*. Takođe se dodeljuje virtualni interfejs monitoru i drajveru preko konfiguracionog objekta. *Port* i *eksport* klase su parametrizovane *sequence\_item* objektom, koju šalju da bi inicirali transakciju ili kao odgovor na transakciju.



*Slika 11. Na slici je prikazan bidirekcioni TLM mehanizam komunikacije između drajver i sekvencer komponente*

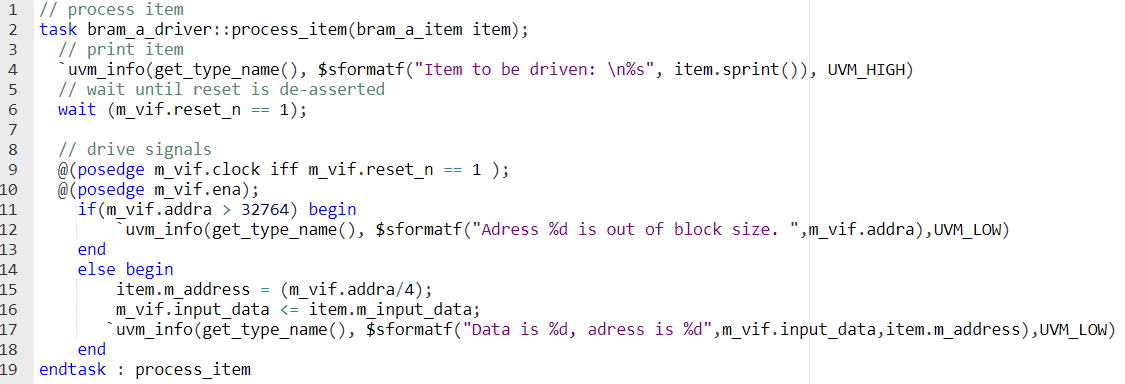
### BRAM A driver i BRAM A sekvencer

Run faza unutar *bram\_a\_driver* klase sadrži metode koje su zadužene za generisanje stimulusa na *BRAM* interfejsu *DUT*-a. Na izdvojenom kodnom segmentu unutar *forever begin* petlje uočava se *get\_next\_item()* blokirajuća metoda koja čeka da transakcija postane dostupna i vraća pokazivač na taj objekat. Zatim se poziva *process\_item* metoda. Nakon toga metoda *item\_done* signalizira da je uspešno kompletiran drajver-sekvencer *handshake*.



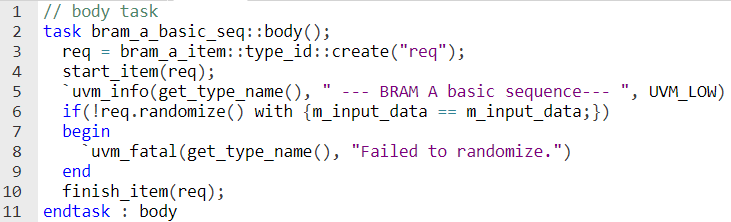
*Listing 5. Run faza bram\_a\_driver komponente*

Unutar *proces\_item* metode implementirana je logika na osnovu koje drajver treba da očekuje a zatim i šalje signale ka dizajnu preko *BRAM* interfejsa. Drajver komponenta unutar *BRAM* A *agent*a čeka da dizajn postavi na logičku jedinicu *enable* signal na *BRAM* A memoriji unutar dizajna. Kad se *enable* signal postavi na logičku jedinicu drajver šalje sekvenci adresu sa koje želi da pročita podatak. Ukoliko je adresa izvan opsega izbacuje se poruka,u suprotnom *BRAM* A *agent* simulira način rada *BRAM* memorije tako što sekvenca vraća podatak sa adrese koju je prosledio drajver.



*Listing 6. Implementacija process\_item metode unutar bram\_a\_driver komponente*

Sekvenca predstavlja glavni mehanizam za generisanje stimulusa. Sve klase unutar *UVM* metodologije nasleđuju *uvm\_object* klasu. *UVM* sekvenca je parametrizovana tipom *sequence\_item-a* koji će se generisati i koristi objekat ( *req* ) , što znači da se polja unutar sekvence mogu na jednostavan način randomizovati i kreirati različit stimulus. U kodnom isečku dat je prikaz implementacije *body()* taska unutar *bram\_a\_basic\_seq* klase. Pre pokretanja sekvence potrebno je kreirati objekat, što je urađeno unutar *body()* taska u 3. liniji koda. Nakon toga sekvenca se pokreće *start()* metodom, koja se nalazi na 4. liniji koda. Pozivom *start()* metode *req* objekat koji je napravljen u sekvenci prosleđuje se *get\_next\_item* metodi koja se nalazi unutar *bram\_a\_driver* klase.

**

*Listing 7. Kreiranje, randomizovanje i pokretanje bram\_a\_basic\_seq sekvence*

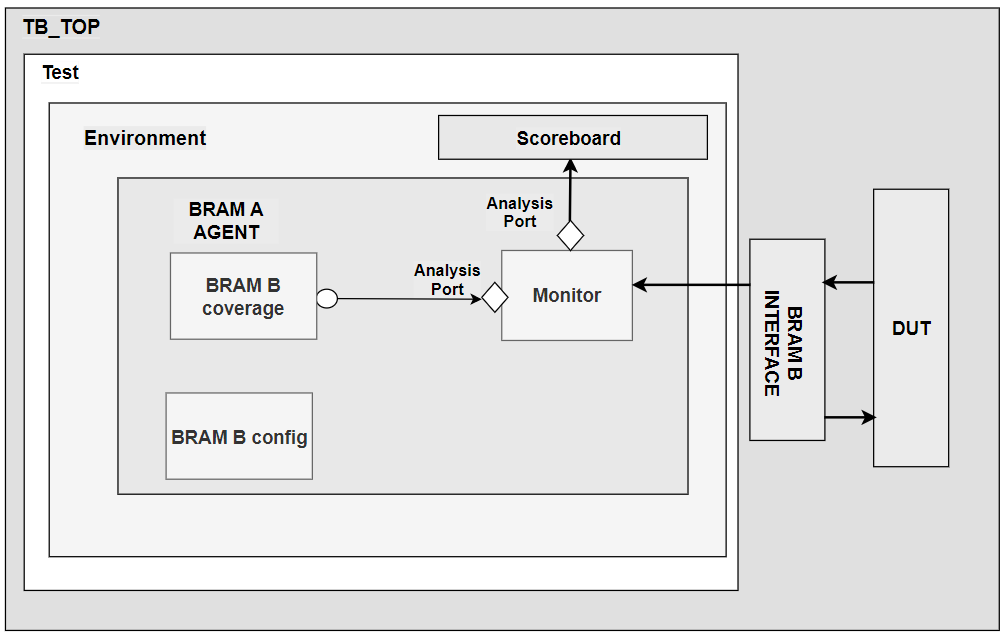
### BRAM A monitor i scrambler\_ip\_scoreboard

Komponenta koja nadgleda ponašanje *DUT*-a je *uvm\_monitor* komponenta. Na osnovu stimulusa koji *bram\_a\_driver* komponeneta šalje preko virtualnog interfejsa na *DUT*, odvojena komponenta *bram\_a\_monitor* očekuje odgovor *DUT*-a tako što nadgleda signale sa vritualnog interfejsa. Monitor komponenta je odvojena i nezavisna od drajver komponente kako ne bi uticala na stimulus koji se generiše. Kad *DUT* postavi signal *enable\_a* na logičku jedinicu, sinhrono sa taktom prosleđuju se adresa i podatak koji se u tom trenutku simulacije nalaze na portovima dizajna. Takođe se prosleđuje i sam *enable\_a* signal radi kasnije provere. Kad se sakupe potrebni podaci sa virtualnog interfejsa daljoj verifikaciji se pristupa na višem nivou apstrakcije. Korišćenjem *TLM* ( *engl. Transaction Level Modeling* ) interfejsa sakupljeni podaci se grupišu u jednu transakciju. Komunikacija između *bram\_a\_monitor* i *scrambler\_ip\_scoreboard* komponente korišćenjem *UVM* objekta *analysis\_port* kojisadrži *write()* funkciju. Pozivom *write()* funkcije šalje se transakcija *scoreboard* komponenti koja se nalazi na višem nivou hijerarhije i koja na osnovu dobijene transakcije vrši dalje provere. *Write()* funkcija je implementirana u scoreboard komponenti. *TLM* konekcija na osnovu koje *scrambler\_ip\_scoreboard* dobija transakcije ’*uvm\_analysis\_imp\_decl(\_bram\_a).* Monitor i scoreboard komponenta su povezani pomoću *connect* metode u *scrambler\_ip\_env\_top* klasi.

## BRAM B agent

Pomoću *BRAM* *B* *agent*a simuliran je način rada *BRAM* *B* memorije. *BRAM* *B* memorija. je kao i *BRAM* *A* memorija konfigurisana kao dvopristupna. *BRAM* *B* memorija koristi se za smeštanje skremblovanih podataka koje scrambler\_*ip* jezgro obradi. Kapacitet *BRAM* *B* memorije takođe iznosi 36864 memorijske lokacije gde su odbirci grupisani u blokove od 8192 odbirka, pri čemu je svaki odbirak predstavljen sa 32 bita.

Za razliku od *BRAM* *A* *agent* komponente *BRAM* *B* *agent* je konfigurisan kao pasivan *agent*. *BRAM* *B* *agent* ne generiše stimulus tako da ne sadrži *UVM* drajver, *UVM* sekvencer i *UVM* sekvence. Sastoji se od *bram\_b\_monitor*, bram\_b\_coverage i bram\_b\_config komponente kao što je prikazano na slici 12.



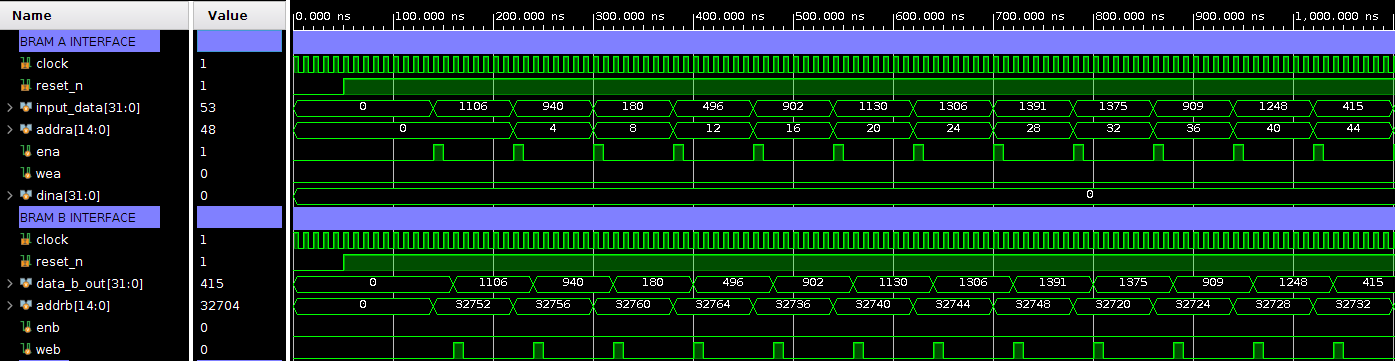
*Slika 12. Na slici je prikazan BRAM B agent sa svojim podkomponentama*

### BRAM B monitor i scrambler\_ip\_scoreboard

Simulacija *BRAM* *B* memorije se realizuje pomoću *UVM* monitor i *UVM* coverage komponente tako što se sakupljaju signali sa virtualnog interfejsa. U *bram\_b\_monitor* klasi ukoliko je signal za dozvolu upisa *write\_enable* asertovan sinhrono sa taktom sakupljaju se adresa i podatak sa *BRAM* *B* interfejsa. Takođe se sakuplja signal *write\_enable* radi dalje provere. Zatim se sakupljeni signali grupišu u transakciju koja se pozivom *write()* funkcije prosleđuje *scrambler\_ip\_scoreboard* komponenti koja se nalazi na višem nivou hijerarhije. Komunikacija između *bram\_b\_monitor* i *scrambler\_ip\_scoreboard* komponente kao i kod *BRAM* *A* *agent*a realizovana je pomoću *TLM* interfejsa. *TLM* konekcija na osnovu koje *scrambler\_ip\_scoreboard* dobija transakcije ’*uvm\_analysis\_imp\_decl(\_bram\_b).* Monitor i scoreboard komponenta su takođe povezani pomoću *connect* metode u *scrambler\_ip\_env\_top* klasi.

## Rezultati simulacije BRAM A i BRAM B memorije

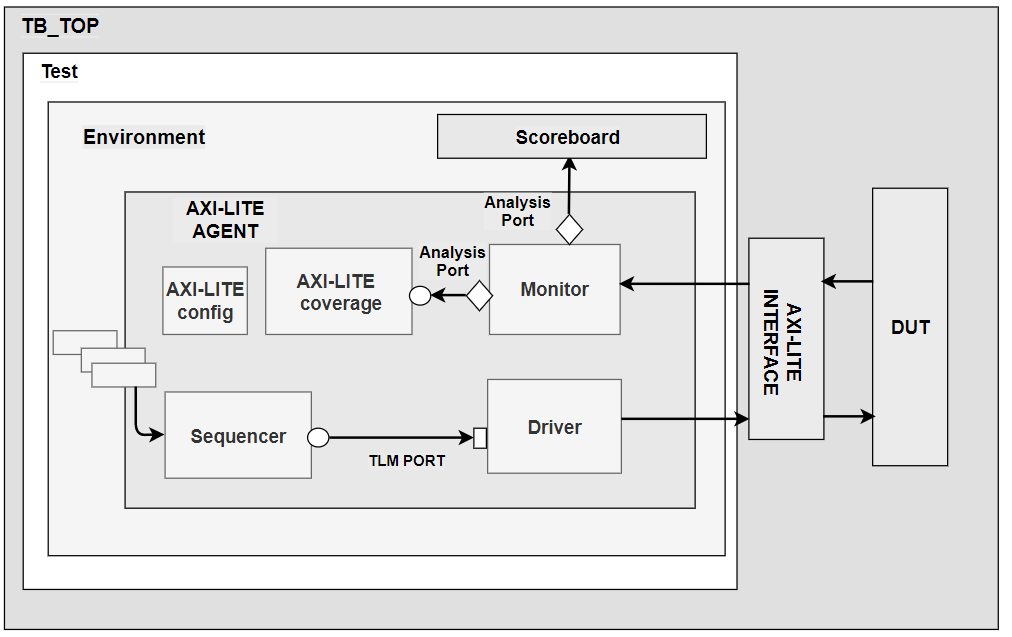
Na narednoj slici prikazan je izgled *BRAM A* i *BRAM B* interfejsa posle 1 *ms* nakon pokretanja simulacije. Signal *BRAM A* interfejsa koji daje indikaciju dozvole čitanja *ena* povezan je sa signalom *enable\_a* na dizajnu asertuje se za svaku transakciju. Takođe signal *BRAM B* interfejsa web koji služi kao indikator dozvole upisa asertuje se posle svake transakcije. Signal *web* se dovodi na signal *write\_enable* koji se nalazi na dizajnu. Radi preglednosti prikazani su rezultati simulacije za prve 44 adrese. *DUT* postavlja adrese koje želi da pročita iz *BRAM A* memorije počevši od 0 – 32764. Sa slike se vidi da su adrese na *BRAM B* interfejsu skremblovane. Podaci koji su se nalazili u *BRAM A* memoriji smešteni su u *BRAM B* memoriju u pravilnom redosledu, što znači da se na ulaznoj adresi nakon operacije skremblovanja nalazi isti podatak.

**

*Slika 5. Prikaz BRAM A i BRAM B interfejsa za prvih 1 ms trajanja simulacije*

## AXI-LITE agent

Obzirom da *scrambler\_ip* jezgro komunicira sa *Zynq* procesorom na osnovu *AXI-LITE* protokola za razmenu podataka koriste se standardizovane *AXI* magistrale. Zadatak *AXI-LITE* *agent*a je da implementira ovaj protokol. Pomoću *AXI-LITE* interfejsa moguće je pristupiti kontrolnim signalima unutar modula. Postavljanjem ovih kontrolnih signala moguće je poslati komandu *DUT-u* da počne sa obradom podataka i proveriti da li je *DUT* spreman za obradu narednog bloka podataka.



*Slika 13. Na slici je prikazan AXI-LITE agent sa svojim podkomponentama*

### AXI-LITE monitor

*AXI-LITE* monitor komponenta je zadužena za sakupljanje signala *DUT*-a. *AXI-LITE* monitor je realizovan korišćenjem standardnih *TLM* interfejsa. Monitor komponenta koristi analysis *PORT* i poziv *write* funkcije kako bi poslala transakcije ostalim komponentama. U zavisnosti od statusih signala sakuplja određene signale u zavisnosti da li se radi o *read* ili *write* transakciji. Signal s\_*AXI*\_awready daje indikaciju da se radi o *write* transakciji. Na osnovu toga monitor sakuplja pomoću virtualnog interfejsa signal koji se nalazi na adresnom Portu *DUT-a*, zatim sakuplja signal sa porta za podatke i sakuplja podatak o tipu transakcije radi dalje provere. Nakon sakupljenih signala koji omogućavaju upis u *DUT* poziva se *write()* funkcija koja scoreboard komponenti prosleđuje ove informacije u grupisane vidu transakcije. Zatim ukoliko je signal *s\_axi\_arready* asertovan radi se o *read* transakciji. Sakuplja se pročitana adresa, a ukoliko je u narednom takt signalu *s\_axi\_arvalid* asertovan sakuplja se i pročitan podatak. Takođe sakuplja se i podatak o tipu transakcije radi dalje provere. Nakon sakupljenih signala koji omogućavaju čitanje iz *DUT-a* poziva se *write()* funkcija koja scoreboard komponenti prosleđuje ove informacije takođe u vidu transakcije. U *scrambler\_IP\_top\_env* klasi pomoću *connect* metode povezani su *scrambler\_ip\_scoreboard* i *AXI\_monitor* komponente.

### AXI-LITE drajver, sequencer, sequence

Read transakcija

Na osnovu *UVM* metodologije šalju se sekvence pomoću sekvencera ka *axi\_lite\_driver* komponenti koja ima direktan pristup *AXI-LITE* interfejsu kao što se vidi na slici 13. Početni korak *axi\_lite drajver* komponente je inicijalizacija signala kojima se pristupa preko virtualnog interfejsa. Zatim kao i kod *BRAM* *A* i *BRAM* *B* drajvera poziva se *get\_next\_item* metoda i čeka se na objekat koji će biti poslat od strane sekvencera. Nakon dobijenog pokazivača na objekat poziva se metoda *process\_item* u kojoj je realizovana logika drajvera pomoću *AXI*\_LITE protokola. Na osnovu polja *rw\_op* koji se postavlja u sekvenci drajver dobija informaciju da li se vrši čitanje ili upis u *DUT*. Način rada dizajna je takav da je potrebno prvo proveriti vrednost *ready* kontrolnog signala. Zato se prvo pokreće *axi\_lite\_read\_ready\_register\_seq* sekvenca. Ova sekvenca postavlja *rw\_op* polje na logičku nulu što je ekvivalent *read* operaciji. Na osnovu vrednosti koju drajver dobija kao vrednost *rw\_op* polja unutar taska *read\_trans* postavlja se signal *s\_axi\_arvalid* koji se nalazi na *DUT*-u, pomoću virtualnog interfejsa na logičku jedinicu. Ovo polje daje indikaciju da je podatak koji se trenutno čita validan. Zatim se prosleđuje adresa postavljena u sekvenci na adresu na *DUT*-u preko virtualnog interfejsa. Nakon toga u narednom ciklusu takt signala postavlja se *s\_axi\_arready* na jedinicu kao indikacija da je uspešno upisan podatak. *AXI-LITE* protokol šalje jedan podatak po transakciji. U narednom takt ciklusu se signal *s\_axi\_arready* postavlja na logičku nulu što je indikacija da je završen proces upisa. Čime je kompletirana jedna *read* transakcija.

Write transakcija

Prema specifikaciji scrambler\_*IP* jezgro pokreće operaciju skremblovanja pomoću *start* kontrolnog signala. Iz sekvenca koja se pokreće u testu je *axi\_lite\_write\_start\_register\_value\_seq*. Ova sekvenca postavlja *rw\_op* polje na jedinicu, što je ekvivalent write operaciji i postavlja jedinicu kao vrednost podatka koji se upisuje u *DUT* preko *AXI-LITE* interfejsa. Unutar drajvera poziva se *write\_trans* task koji nakon provere *rw\_op* polja prosleđuje *DUT*-u adresu koja je u sekvenci postavljena da se poklapa sa adresom na koju je mapiran *start* registar. Zatim postavlja signal *s\_axi\_awvalid* na logičku jedinicu. Čime daje indikaciju da je trenutna adresa validna. Nakon toga *s\_axi\_awstrb* signalu se prosleđuje vrednost u kojoj su sva četiri bita setovana što znači da su sva četiti bajta u podatku validna. Zatim se prosleđuje podatak koji je postavljen u sekvenci pomoću virtualnog interfejsa na signal *s\_axi\_awdata*. Zatim se signalu *s\_axi\_bready* prosleđuje vrednost logičke jedinice koji služi kao statusni signal koji označava da je *AXI* *master* spreman da prihvati podatak o statusu. Zatim se zajedno sa signalom takta proverava *s\_axi\_awready* signal koji kad se asertuje označava da je *DUT* spreman da prihvati podatak. Zatim pošto se pomoću *AXI-LITE* protokola upisuje jedan podatak po transakciji u narednom takt signalu provera se da li je *DUT* postavio *s\_axi\_awready* signal na logičku nulu. Nakon toga vrednosti statusnih signala *s\_axi\_awvalid* i *s\_axi\_awstrb* postavljaju se na logičke nule, i uklanjaju se vrednosti sa magistrala za podatke i adrese. Zatim se proverava finalni status transakcije pomoću provere da li je *DUT* postavio *s\_axi\_bvalid* na logičku nulu, ukoliko jeste statusni signal *s\_axi\_bready* se postavlja na logičku nulu. Signal *s\_axi\_bready* ostaje na nuli do sledeće transakcije upisa. Nakon *proces\_item* metode poziva se *item\_done* metoda.



Slika 6. Signali AXI-Lite interfejsa u slučaju read i write transakcije prikazani za 1 ms trajanja simulacije

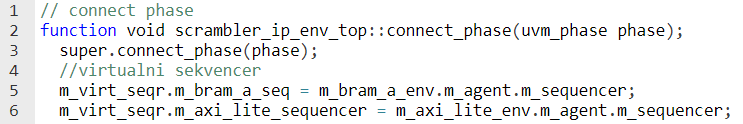
Na slici 7 su predstavljeni signali sa *DUT-a* nakon 1 *ms* trajanja simulacije. Signal *start\_axi\_o\_s* se asertuje nakon *s00\_axi\_aresetn* signala reseta koji je aktivan na niskom nivou. Signal *ready\_axi\_i* dobija vrednost logičke nule nakon asertovanja *s00\_axi\_aresetn* i *start\_axi\_o\_s*. Sve dok *ready\_axi\_i* signal ne dobije vrednost logičke jedinice *scrambler\_ip* jezgro nije spremno da prihvati novi blok podataka od 8192 podatka za obradu.



*Slika 7. Signali AXI-LITE interfejsa na DUT-u prikazani za 1 ms trajanja simulacije*

## Virtualni sekvencer

Kontrola redosleda pokretanja sekvenci vrši se pomoću virtualnog sekvencera. To je komponenta koja je instancirana u *scrambler\_ip\_env\_top* klasi. Koordinacija generisanja stimulusa na *BRAM* *A*, BRAM B i *AXI-Lite* interfejsu postiže se pomoću virtualnog sekvencera tako što on sadrži pokazivače na *bram\_a\_sequencer* i *axi\_lite\_sequencer* komponente. Obzirom da je *BRAM* *B* *agent* konfigurisan kao pasivan on ne sadrži sekvencer što znači da virtualnom sekvenceru nije moguće dodeliti pokazivač na *bram\_b\_sequencer*. Virtualni sekvencer se nalazi na nivou hijerarhije iznad sekvencera pojedinačnih agenata. U *scrambler\_ip\_env\_top.sv* fajlu u connect fazi povezani su virtualni sekvencer i sekvencer unutar *BRAM* *A* i *AXI-Lite* *agent*a kao što se vidi na kodnom isečku.



*Listing 8. Povezivanje virtualnog sekvencera sa sekvencerima unutar agenata*

Kontrola rada *DUT*-a vrši se na osnovu registara unutar *AXI-LITE* kontrolera. Četiri memorijski mapirana registra su napravljena prilikom kreiranja *AXI-LITE* interfejsa, pri čemu je čitav modul konfigurisan tako da prvi registar koji se nalazi na adresi 0x00 koristi kao *reset* registar, drugi registar koji je memorijski mapiran na adresi 0x04 koristi kao *start* registar koji služi za pokretanje rada modula, i *ready* registar je mapiran na adresi 0x08 koji služi kao indikator da je modul spreman da obradi naredni blok. Četvrti registar se ne koristi jer se podaci šalju preko *BRAM* interfejsa.

Virtualna sekvenca se pokreće pomoću virtualnog sekvencera u *run\_phase* tasku unutar *scrambler\_IP\_test\_example.sv* fajla. Virtualna sekvenca sadrži redosled sekvenci na osnovu kog se *DUT* verifikuje. Prvo se pokreće *axi\_lite\_read\_ready\_register\_seq* sekvenca *AXI-Lite* *agent*a koja je zadužena za čitanje vrednosti *ready* registra koji je memorijski mapiran na adresi 0x08. Nakon toga pokreće se *axi\_lite\_write\_start\_req\_value\_seq* sekvenca unutar *AXI\_Lite* *agent*a koja upisuje logičku jedinicu u *start* registar koji je memorijski mapiran na adresi 0x04.

Zatim se pokreće *axi\_lite\_write\_start\_req\_value\_seq* sekvenca unutar *AXI*\_Lite *agent*a koja upisuje logičku nulu u start registar. Nakon toga pokreće se blok *BRAM* a sekvenci koji za cilj ima da pošalje preko *BRAM* *A* interfejsa *DUT*-u blok od 8192 adrese na kojima se nalaze randomizovani podaci. Dalji tok preuzima *DUT* koji ukoliko ispravno radi treba da na osnovu algoritma skrembluje adrese i posalje preko *BRAM* *B* interfejsa u *BRAM* *B* memoriju.

## Scoreboard

Ispravnost rada *DUT*-a provera se pomoću scoreboard komponente. Ova komponenta je instancirana u scrambler\_*IP*\_env\_top klasi. U ovoj klasi takođe je i povezana u “*connect\_phase”* tasku sa monitorima *BRAM* *A*, *BRAM* *B* i *AXI-LITE* *agent*a. Scoreboard komponenta sadrži *TLM* konekcije i implementaciju *write()* metoda. Provera rada *DUT*-a vrši se u *write()* funkcijama unutar scoreboarda. Ove funkcije kao parametar dobijaju transakcije koje monitori šalju preko *analysis\_port-a* i na taj način imaju uvid u signale koji se nalaze na virtualnom interfejsu.

Scoreboard se sadrži tri *write()* funkcije. U *axi\_lite\_write(axi\_lite\_item m\_axi\_lite)* funkciji vrši se provera na osnovu dobijene transakcije prikupljene od strane *axi*\_lite\_monitora da li se adrese dobijene u transakciji poklapaju sa memorijski mapiranim adresama registara unutar *AXI-LITE* interfejsa. Zatim u *bram\_a\_write(bram\_a\_item m\_bram\_a\_item)* vrši se smeštanje adresa iz dobijene transakcije koju sakuplja *bram\_a\_monitor* u red za dalju proveru. Takođe u *bram\_a\_write* funkciji vrši se provera da li je enable signal asertovan kad se pristupa *BRAM* memoriji.

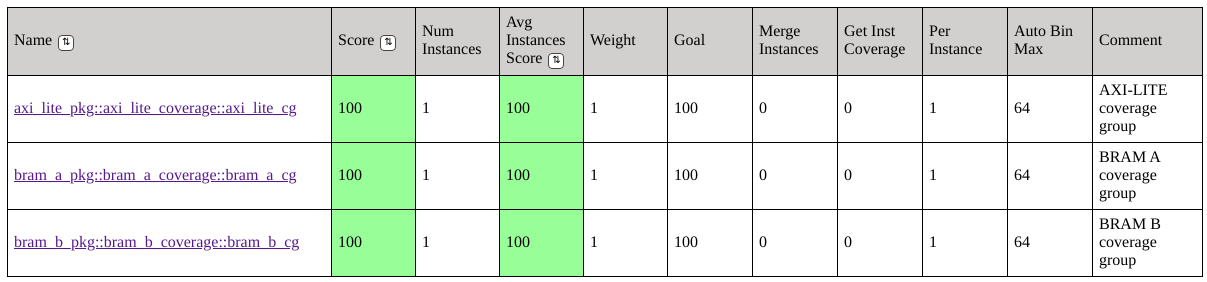
Na osnovu specifikacije *BRAM* memorije da bi se izvršio upis potrebno je da je enable signal postavljen na logičku jedinicu. U *bram\_b\_write(bram\_b\_item m\_bram\_b\_item)* funkciji adrese dobijene iz transakcije koju sakuplja *bram*\_b\_monitor smeštaju se u red, radi dalje provere. Nakon toga u *bram\_b\_write* funkciji vrši se provera da li je write enable signal postavljen na logičku jedinicu. *BRAM* *B* memorija služi kao memorija za upis skremblovanih podataka, tako da je pre upisa potrebno setovati signal za dozvolu upisa.

Zatim se u *bram\_b\_write* funkciji poziva funkcija *scrambler\_address\_checking( bram\_a\_que, bram\_b\_que )*. Ova funkcija kao parametre prima *bram\_a\_que i bram\_b\_que* redove. U ovim redovima smeštene su *BRAM* *A* i *BRAM* *B* adrese pri dolasku svake transakcije.

Funkcija *scrambler\_address\_checking* počinje sa radom nakon provere da li su pristigle adrese čitavog bloka koji se šalje jezgru (8192). Sve dok uslov ne bude ispunjen ispisuje se poruka o trenutnoj popunjenosti redova. Ukoliko je uslov ispunjen prelazi se na izvršavanje implementiranog referentnog modela. U for petlji za svaku *BRAM* *A* adresu vrši se proračun na osnovu kog se pomoću *BRAM* *A* adrese dobija vrednost *bram\_b\_address\_calc*. *Bram\_b\_address\_calc* nakon proračuna treba da sadrži adresu ekvivalentnu adresi koja bi se dobila nakon što jezgro izvrši operaciju skremblovanja. Što se i poredi pomoću *BRAM* *B* adrese smeštene u *bram\_b\_que*. Ukoliko se adrese za čitav blok podataka poklapaju dobija se poruka o uspešnom radu *DUT*-a. U suprotnom scoreboard komponenta prijavljuje poruku o grešci i ispisuje vrednosti adresa koje se ne poklapaju.

## Verifikaciona pokrivenost scrambler\_ip jezgra

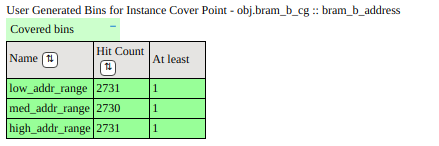
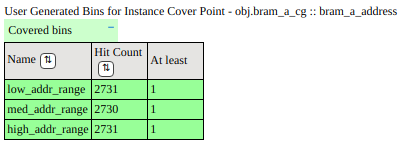
Prikupljanje pokrivenosti vrši se pomoću *Xilinx Coverage Report Generator* alata. Način prikupljanja pokrivenosti realizovan je u posebnim komponentama koje su implementirane u sva tri *agent*a. Ove komponente implementirane su u fajlovima *bram\_a\_coverage.sv*, *bram\_b\_coverage.sv, axi\_lite\_coverage.sv*. Na osnovu podataka dobijenih iz ovih komponenti dobijamo informaciju o osobinama *DUT*-a koje su verifikovane i u kojoj meri se poklapaju sa specifikacijom. Nadgledanjem *BRAM* *A* interfejsa vrši se provera pokrivenosti tako što se sakupljaju informacije o vrednosti adrese. Adresa može imati 32764 vrednosti, koje su pri prikupljanju pokrivenosti podeljene u opsege. Vrednost randomizovanih podataka koji se šalju je ograničena na opseg 0-1500. Tokom prikupljanja pokrivenosti proverava se da li su prikupljene vrednosti za podatke iz ograničenog opsega. Prikupljanje pokrivenosti za *BRAM* *B* interfejs se obavlja na isti način. Od interesa su prikupljene adrese i podaci. *AXI-LITE* interfejs je od predstavlja bitan izvor informacija za prikupljanje pokrivenosti jer sadrži kontrolne registre.



*Slika 14. Izveštaj Xilinx Coverage RePORT Generator alata o cover grupama*

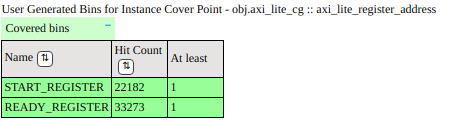
Unutar *bram\_a\_cg* i *bram\_b\_cg* grupe provereno je da li se desio pristup celokupnom memorijskom opsegu. Čitav blok koji se obrađuje u jezgru sastoji se od 8192 adrese i koje zauzimaju memorijski opseg od 32764 memorijske lokacije. U *bram\_a\_cg* i *bram\_b\_cg* opseg memorijskih lokacija je podjednako podeljen na tri dela. Prilikom prikupljana pokrivenosti proverava se da li su sakupljene adrese iz sva tri opsega.

Pokrivenost *bram\_a\_cg* i *bram\_b\_cg:*

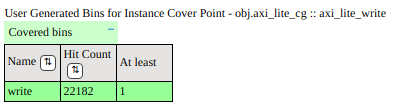
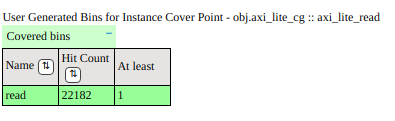


Unutar *axi\_lite\_cg* grupe provereno je:

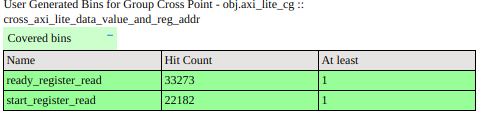
1. Da li se desio pristup registrima unutar modula.



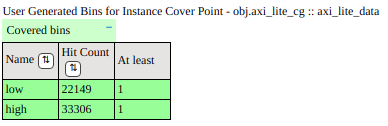
1. Provera da li se desila operacija čitanja i upisa.



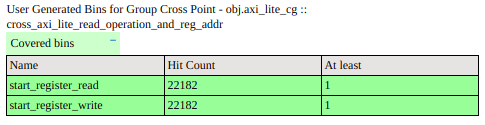
1. Provera da li se desila operacija čitanja i upisa u odgovarajuće registre. Proverena je operacija čitanja za *read-only READY* registar i operacije čitanja za *START* registar.



1. Provera da li podatak koji se upisuje ili čita iz registara ima vrednost logičke nule i logičke jedinice.



1. Provera da li se u *START* registar upisuje i čita.



# Literatura

* 1. Rastislav Struharnik, vežbe i predavanja predmeta Funkcionalna verifikacija

<https://www.elektronika.ftn.uns.ac.rs/funkcionalna-verifikacija-hardvera/specifikacija/specifikacija-predmeta/>

* 1. Rastislav Struharnik, vežbe i predavanja predmeta Projektovanje složenih digitalnih sistema

<https://www.elektronika.ftn.uns.ac.rs/projektovanje-slozenih-digitalnih-sistema/specifikacija/specifikacija-predmeta/>

* 1. [http://read.pudn.com/downloads654/ebook/2659841/*UVM*-cookbook.pdf](http://read.pudn.com/downloads654/ebook/2659841/UVM-cookbook.pdf)
  2. Rastislav Struharnik, vežbe i predavanja predmeta Diskretni sistemi

https://www.elektronika.ftn.uns.ac.rs/diskretni-sistemi/specifikacija/specifikacija-predmeta/

1. *Callgrind –* Alat se koristi za profajliranje funkcija. Generiše listu poziva funkcija korisničkog programa u vidu grafa. Pomoću grafa poziva dobija se podatak koja funkcija, počevši od *main* funkcije, ima najviše poziva iz drugih funkcija ( cena funkcija ). Pored toga, alat broji događaje koji se dešavaju direktno u jednoj funkciji. Na taj način alat sakuplja podatke na osnovu kojih može međusobno porediti funkcije. [↑](#footnote-ref-1)
2. *Valgrind* – Alat koji dinamički analizira korisnički program u izvršavanju. Između ostalog sadrži alate koji se koriste za detekciju memorijskih grešaka ( *engl. Memcheck* ), praćenje rada dinamičke memorije ( *engl. Massif* ) , profajler funkcije ( *engl. Callgrind* ). [↑](#footnote-ref-2)
3. *TLM* – *engl. Transaction Level Modelling* predstavlja industrijski standard koji pomoću transakcija ostvaruje komunikaciju između procesa. Modeli na nivou transakcija komuniciraju pozivima funkcija. Jednim pozivom funkcije postavlja se veći broj signala, što čini *TLM* simulacije brže od *RTL* simulacija. [↑](#footnote-ref-3)