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:

4.11.2020

Sadržaj

[Funkcionalna verifikacija hardvera 1](#_Toc55304756)

[1. Uvod u funkcionalnu verifikaciju hardvera 3](#_Toc55304757)

[2. Realizacija projekta – Audio Scrambler System 5](#_Toc55304758)

[2.1. AXI Interfejs 6](#_Toc55304759)

[2.2. Definisanje interfejsa 8](#_Toc55304760)

[3. UVM metodologija verifikacije projektovanih digitalnih sistema 9](#_Toc55304761)

[3.1. UVM hijerarhija 10](#_Toc55304762)

[3.2. Verifikaciono okruženje 11](#_Toc55304763)

[4. BRAM A AGENT 13](#_Toc55304764)

[5. Literatura 18](#_Toc55304765)

# 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.

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 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. Analizirani su koraci od specifikacije HDL modela do implementacije verifikacionog plana i verifikacionog okruženja.

U trećem poglavlju dat je prikaz realizacije projekta. Pojedinačni podsistemi i način na koji su oni povezani je opisan.

U četvrtom poglavlju nalazi se zaključak.

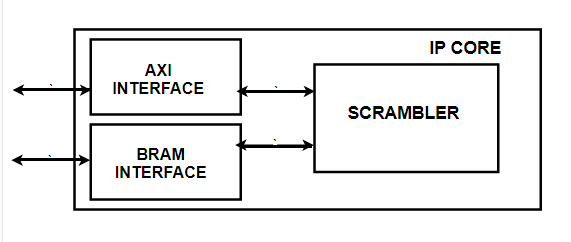
U petom poglavlju navedena je korišćena literature.

# Realizacija projekta – Audio Scrambler System

Nakon kreiranog softverskog modela virtualne platforme čija je svrha da bude zamena hardverskim prototipovima u ranim fazama razvoja sistema, projektovano je IP jezgro namenjeno implementaciji hardverske komponente sistema. Zatim je projektovano IP jezgro verifikovano pomoću UVM metodologije.

Audio scrembler sistem 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.

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 (eng. 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 32 768 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 1. Šematski prikaz implementiranog IP jezgra*

## AXI Interfejs

AXI (Advanced Extensible Interface) magistrale predstavljaju poslednju evoluciju ARM AMBA (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. AXI komunikacioni protocol ima sledeće karakteristike:

* 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 mastere i AXI slejvove. AXI masteri iniciraju transakciju, dok AXI slejvovi mogu samo da odgovaraju na započetu transakciju. Za ostvarivanje potrebnih veza između AXI mastera I AXI slejvova koriste se posebni moduli, AXI interkonekti (AXI Interconnect). U slučaju da postoji samo jesan AXI master I jedan AXI slejv, AXI interkonekt nije potreban. AXI master inicira transakciju sa željenim AXI slejvom preko odgovarajućeg AXI interkonekt modula, na koji su povezani AXI master i AXI slejv. Analizirajući ciljnu adresu transakcije, koja je deo transakcije, AXI interkonekt odlučuje koje od priključenih slejvova je namenjena transakcija I prosleđuje je na odgovarajući port.

Na slici 2. 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 masteru preko statusnog kanala upisa.



*Slika 2. Transakcija upisa podatka korišćenjem AXI-Lite interfejsa*

Na slici 3. 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 masteru, koristeći kanal pročitanih podataka. Kad AXI master prihvati podatak, transakcija čitanja je kompletirana.



*Slika3. Transakcija čitanja podatka korišćenjem AXI-Lite interfejsa*

## Definisanje interfejsa

1. Interfejs BRAM A memorije

* 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 porta 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

1. Interfejs BRAM B memorije

* b\_addr\_o – tipa STD\_LOGIC\_VECTOR (WADDR-1 downto 0) – predstavlja adresu koja se šalje iz jezgra ka BRAM B memoriji
* b\_data\_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\_wr\_o - tipa STD\_LOGIC – predstavlja jednobitni signal koji se dovodi na ulazni port za selekciju rastućih ivica clk porta na koje BRAM B memorija treba da dozvol upis

1. Komandni interfejs

* start - tipa STD\_LOGIC – predstavlja jednobitni komandni signal kojim procesor kontroliše početak rada

1. Statusni interfejs

* ready - tipa STD\_LOGIC – predstavlja jednobitni signal koji daje indikaciju kad je jezgro spremno za novu transakciju

# 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. Koncept UVM fabrike ( engl. UVM factory ) je jedan od aspekata objektno-orjentisanog programiranja. 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.

## 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 ).

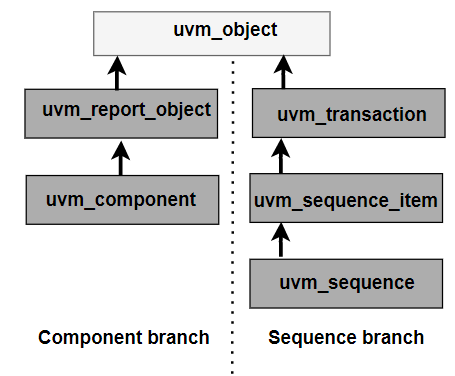
Na slici 2. prikazana je hijerarhija testbenča koji je namenjen verifikaciji Audio Scrambler System dizajna. Kao što je već rečeno na najvišem vrhu hijerarhije nalazi se test, njegovim pokretanjem instacira se verifikaciono okruženje koje obuhvata komponente kao što su Bram A, Bram B i AXI-Lite agenti, 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 agenta ( 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 dizajnu koji se verifikuje.



*Slika 4. 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 5. 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.

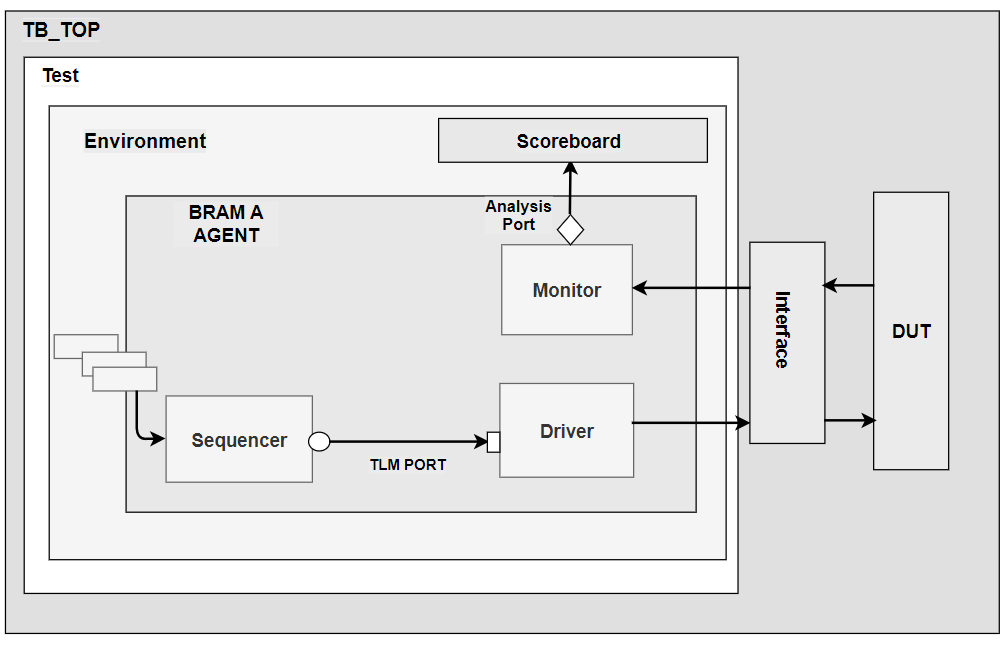
Top-level okruženje korišćeno za verifikaciju Scrambler IP jezgra na najvišem nivou hijerarhije sadrži test klasu. Test klasa sadrži environment klasu koja instancira AXI-Lite, BRAM A i BRAM B agente, koji zatim instanciraju svoje podkomponente. Environment klasa instancira scoreboard komponentu. Ova komponenta je povezana sa monitorima sva tri agenta 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 agenta. 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 agenta, na primer odabir režima rada, kontrolna polja za dozvolu prikupljanja podataka o pokrivenosti. Ova podešavanja vrše se u posebnom objektu koji se prosleđuje agentu koristeći uvm\_config\_db. Konfiguracioni objekat se kreira i podešava na višem nivou hijerarhije, zatim se preko baze prosleđuje agentu. Preuzimanje iz baze vrši se u build fazi.



*Slika 6. Struktura UVM testbenča namenjen verifikaciji Audio Scrambler System dizajna*

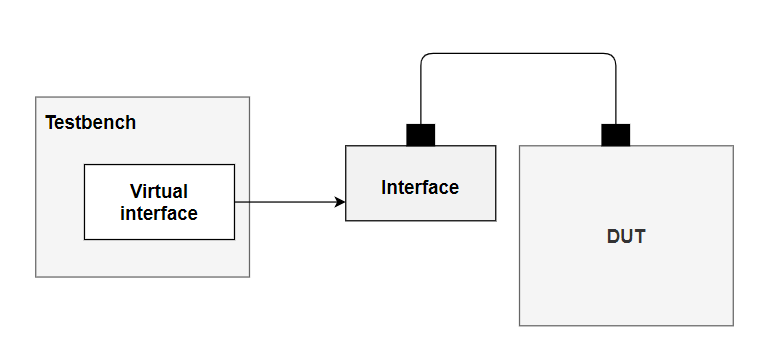
# BRAM A AGENT

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 upis podataka. U BRAM memoriju se smeštaju odbirci nad kojima je potrebno izvršiti operaciju skremblovanja. Zbog dubine BRAM memorije koja iznosi 36 864 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 32 764 memorijske lokacije za jedan blok od 8192 obirka.



*Slika 7. 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. Portovi 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 agenta.

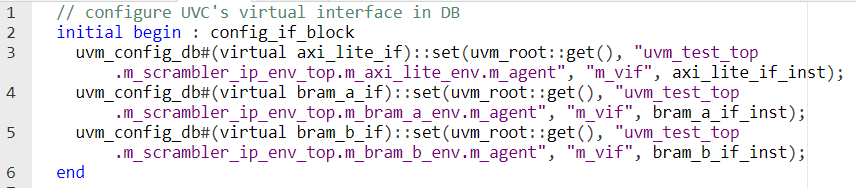


*Slika 8. Pojednostavljen prikaz komunikacije dizajna i testbenča*

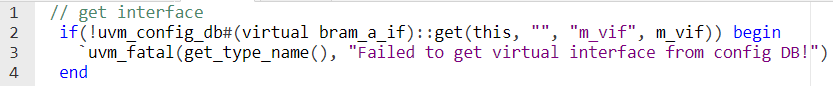
## Standardne UVM faze

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 agenta 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.

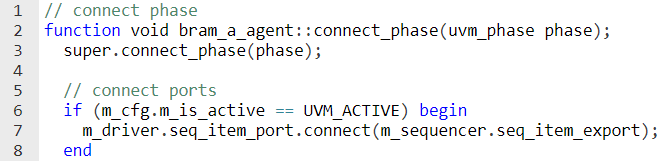


U *bram\_a\_agent.sv* preuzet je pokazivač na virtualni interfejs.

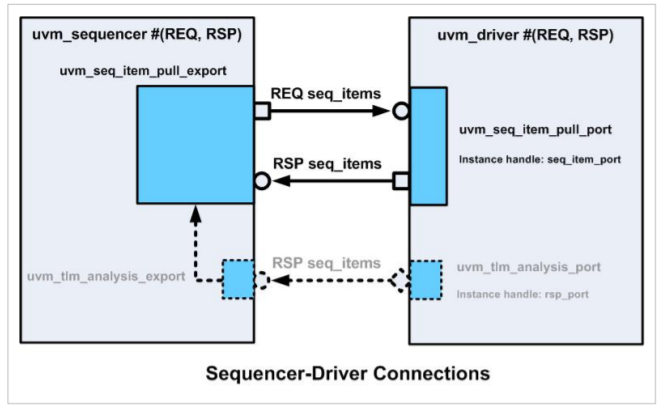


Pored navedene tri osnovne komponete, BRAM A agent takođe sadrži konfiguracionu klasu koja u build fazi na osnovu prosleđenog konfiguracionog objekta kontroliše topologiju agenta, drugim rečima kontroliše način na koji su komponente unutar agenta instancirane, zatim konfiguracioni objekat definiše pokazivač na virtualni interfejs koji monitor i drajver koriste i određuje da li je agent konfigurisan kao pasivan ili aktivan. U slučaju BRAM A agenta 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.



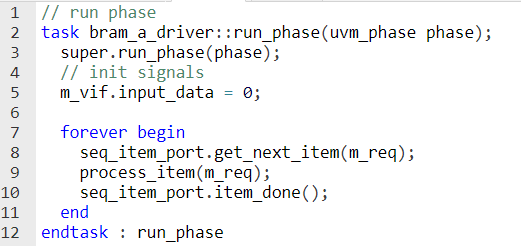
Povezivanje se vrši na osnovu TLM konekcija *analysis\_port* od monitora i *analysis\_port* od agenta, zatim pošto je agent aktivan povezuju se sekvencer i drajver tako što se povezuje *seq\_item\_pull*\_export sa *seq\_item\_pull\_port*. Takođe se dodeljuje virtualni interfejs monitoru i drajveru preko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



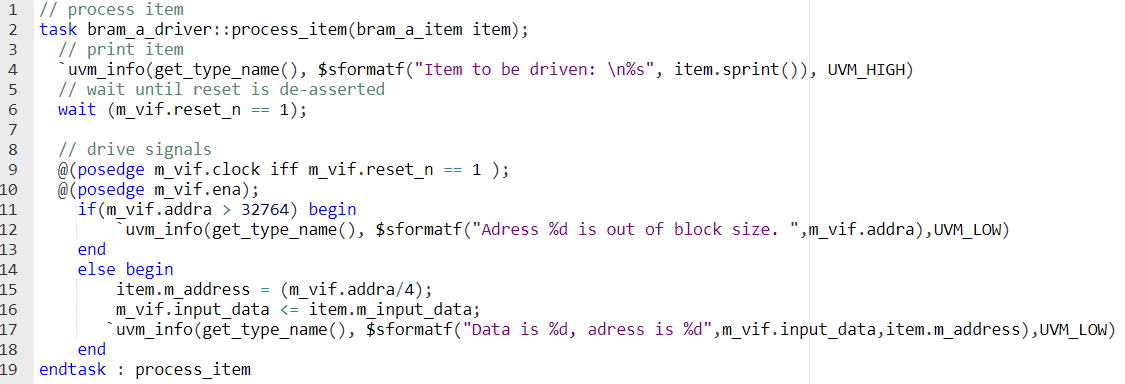
*Slika 9. Na slici je prikazan bidirekcioni TLM mehanizam komunikacije između drajver i sekvencer komponente*

## BRAM A Driver i BRAM A sekvencer komponente

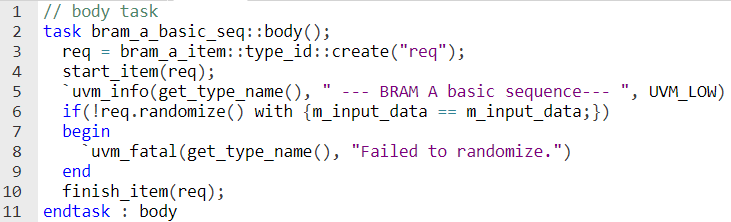
Run faza unutar bram\_a\_driver klase sadrži metode koje su zadužene za generisanje stimulusa na BRAM interfejsu DUT-a. Na izdvojenom kodnom segmentu unutar forever begin petlje uočava se get\_next\_item() blokirajuća metoda koja čeka da transakcija postane dostupna i vraća pokazivač na taj objekat.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



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 agenta 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.



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.

**

# 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