Elektrotehnički fakultet Univerziteta u Beogradu

METOD VERIFIKACIJE ČIPA SA KORISNIČKI DEFINISANIM KOMUNIKACIONIM PROTOKOLOM

Master rad

Kandidat:

Paunović Tamara 3266/2012

Mentor:

doc.dr. Čiča Zoran

Beograd, Septembar 2014.

SADRŽAJ

SPISAK SLIKA

SPISAK SKRAĆENICA

ASIC *Application Specific Integrated Circuit*

DPV *Dizajn pod verifikacijom*

DUT *Device Under Test*

DW *Data Width*

ENV *Environmen*t

OVM *Open Verification Mrthodology*

RHS *Right Hand Side*

RTL *Resistor Transistor Logic*

SoC *System on Chip*

Scbd *Scoreboard*

TB *Testbench*

TCM *Transaction Consuming Method*

UVM *Universal Verification Methodology*

1.UVOD

Sistem na čipu *(System-on-Chips - SoC)* je po definiciji, elektronski sistem ugrađen u jedinstveni čip. Takođe, po definiciji, svaki SoC sadrži najmanje jedan mikroprocesor. Da bi ispunili postavljene zahteve neki od SoC-ova imaju ugrađena dva ili tri mikroprocesora, a postoje i SoC-ovi koji koriste i više od deset procesora.

Verifikacija je proces provere rada ispravnosti uređaja. Jedan od ključnih problema koje treba savladati se odnosi na dokazivanje korektnosti rada SoC-a, što podarzumeva sledeće aktivnosti: simulacija, verifikacija i testiranje.

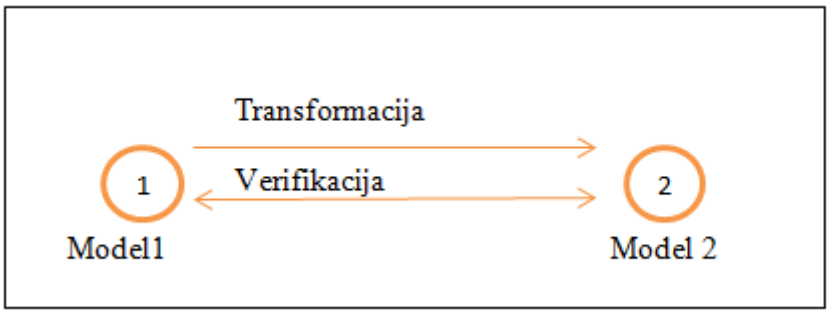
Verifikacija je multinivovska aktivnost kojom se teži ostvariti 100%-na korektnosti funkcionalnosti i zadovoljavanju vremenskih ograničenja SoC-a. Cilj verifikacije (kao jedan od najtežih i najizazovnijih aspekata u projektovanju) mora biti usmeren ka postizanju nula defekata, tj. proces verifikacije treba da teži da otkrije što više grešaka u dizajnu i time obezbedi korektan rad čipa. Pod funkcionalnošću čipa podrazumevamo sve modove rada čipa, protokole koje podržava, i kao takav, ima konkretnu primenu u industriji. Pod vremenskim ograničenjem čipa podrazumevamo broj instrukcija procesora koji on može zadavati u određenom vremenskom periodu. Prilikom procesa verifikacije od alata su najčešci *ModelSim* i *SimVision*. Cilj ove teze je dokazivanje korektnosti rada čipa kao i pokazivanje vremenskih oblika signala, čije je ponašanje saglasno sa uslovima zadatim u specifikaciji.

Teza je organizovana u pet poglavlja. Drugo poglavlje opisuje proces verifikacije, pojam testbenča kao i verifikacione komponente, detaljno opisane kroz kodove radi boljeg razumevanja njihove uloge. Transakcija je vid komunikacije između dve komponente, a sekvenca kao poseban vid transakcije, stimuliše DUT *(Device Under Test)* tokom testa, objašnjena je kroz primer. Treće poglavlje opisuje specifikaciju čipa. Četvrto poglavlje, predstavlja verifikacioni proces ASIC *(Aplication Specific Integrated Circuit)* čipa. Proces verifikacije opisan je kroz kodove potrebne za implementaciju testbenča kao i pokretanje simulacije. Svi dati programski kodovi, detaljno su objašnjeni. Peto poglavlje sadrži zaključke teze. Kodovi su pisani u programskom jeziku System Verilog, a simulacija rađena u simulatoru *SimVision*. Celokupna merenja vršena su u firmi HDL Design House.

2.VERIFIKACIJA I VERIFIKACIONE KOMPONENTE

2.1 VERIFIKACIJA

Verifikacija je proces kojim se pokazuje da su ciljevi specifikacije ostvareni u implementaciji dizajna. Prilikom kreiranja modernih čipova, oko 70% vremena se troši na verifikaciju. Stoga se može zaključiti da je verifikacija kritičan deo posla. Generalni proces verifikacije se može prikazati na slici 2.1.1:

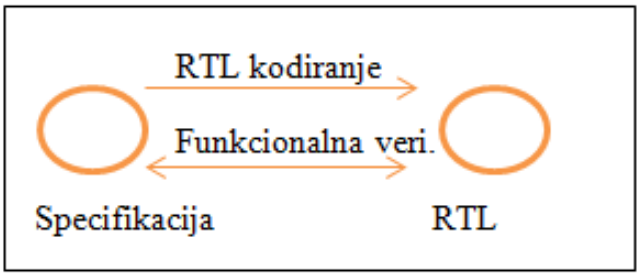


Slika 2.1.1 Uopšteni prikaz verifikacije

Model 1 je početni model na koji se primenjuje neka transformacija. To npr. može biti funkcionalna specifikacija dizajna, a transformacija *RTL (Resistor Transistor Logic)* kodovanje specifikacije. Primenom transformacije se dobija model 2, u navedenom primeru to bi bio RTL model dizajna. RTL je klasa digitalnih kola izrađenih korišćenjem otpornika i bipolarnih tranzistora. Verifikacijom treba da se nađu razlike između ova dva modela. Bitno je da se zna šta se tačno verifikuje. Najčešće se verifikuje naša interpretacija specifikacije naspram dizajna. Verifikator piše kod za verifikacuju na osnovu ličnog razumevanja specifikacije. Ako je interpretacija verifikatora pogrešna, tada verifikacija ne može ukazati na greške.

Postoje dve verifikacije formalna i funkcionalana. Formalna verifikacija dokazuje da su matematički modeli logički ekvivalentni. Neki od postojećih matematičkih modela koriste se za opisivanje željenih modela u verifikaciji. Upoređuje netliste ili otkriva greške u softveru za sintezu hardvera. Funkcionalna verifikacija kontroliše ciljeve dizajna. U svakom projektu postoji funkcionalni završni rezultati koje treba postići. Bez nje ne možemo biti sigurni da li je prelazak sa specifikacije na RTL dizajn urađen korektno. Ona dokazuje samo prisustvo grešaka ali ne i njihovo odsustvo [3].

Na slici 2.1.1 prikazana je funkcionalna verifikacija. Iz specifikacije se kodiranjem dobija RTL. Funcionalna verifikacija predstavlja ″vezu″ između RTL-a i specifikacije. Funkcionalnom verifikacijom se proverava da li je korektno odrađen prelazak sa specifikacije na RTL.



Slika 2.1.2 Grafički prikaz funkcionalne verifikacije

Testiranje i verifikacija se mogu pomešati. Testiranjem se proverava da li je dizajn napravljen korektno na čipu, dok funkcionalna verifikacija garantuje da su ciljevi dizajna ostvareni prilikom njegovog razvoja. Verifikacija traje koliko i razvoj čipa. Nakon toga se za svaki pojedinačni čip radi testiranje da se proveri da li je dizajn korektno implementiran na čipu.

2.2 TESTBENČ

Testbenč je kod za simulaciju koji se sprovodi nad dizajnom pod verifikacijom. Svaki testbenč proizvodi neki stimulus za dizajn i to može biti urađeno na razne načine. Pod stimulusom podrazumevamo pobudu na koju treba da odreaguje DUT. U zavisnosti od pristupa verifikovanju, testbenčovi se mogu klasifikovati na:

- Direktan testbenč
- Samoproveravajući testbenč
- Slučajni testbenč
- Slučajni testbenč sa ograničenjima

Direktni testbenč ima predefinisan stimulus. Taj stimulus se propušta kroz dizajn pod verifikacijom i pravi se izveštaj, koji zatim pregleda osoba koja je zadužena za verifikaciju i proverava da li su dobijeni očekivani rezultati. Za razliku od direktnog testbenča, samoproveravajući, očekivane rezultate sadrži u unapred određenom fajlu. Testbenč u toku simulacije automatski proverava da li su rezultati jednaki očekivanim. Krajnji izveštaj sadrži gotove rezultat verifikacije. Samoproveravajući testbenč je automatizovana varijanta direktnog testbenča. Ova dva načina testiranja dobri su u početnom stadijumu verifikovanja [2].

Slučajni testbenč sadrži generator koji proizvodi slučajni stimulus. Stimulus u ovom slučaju može biti bilo koja validna ulazna sekvenca. Ovaj način testiranja neophodan je da se uoče greške u samom sistemu, npr. da li do nekog od blokova, dolazi transakcija tokom simulacije. Slučajni testbenč sa ograničenjima je suštinski isti kao i slučajni testbenč ali sa vremenskim ograničenjima u stimulusu.

2.3 PRISTUP IP VERIFIKACIJI

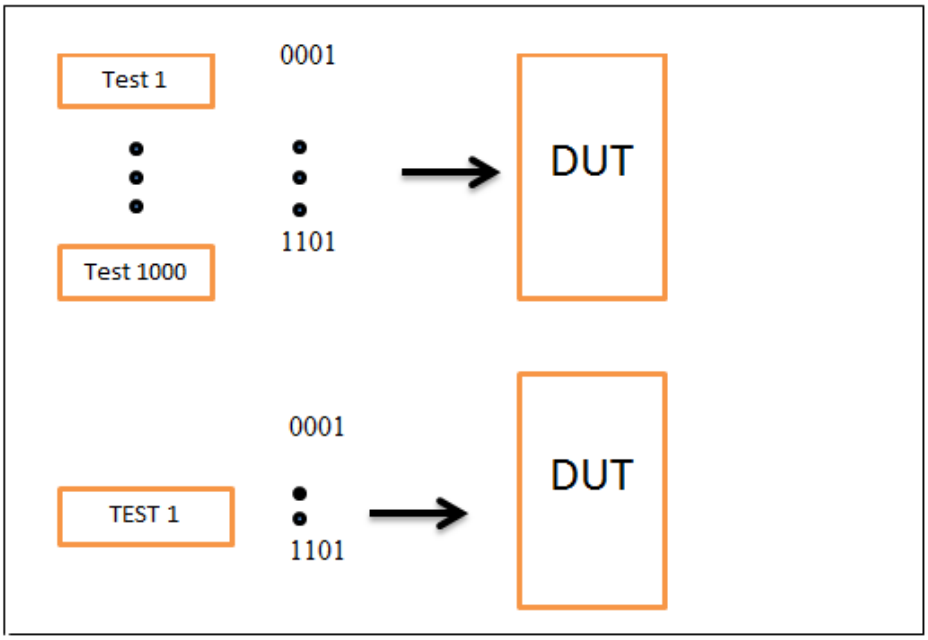
IP (*Intellectual Property*) verifikacija je verifikacija kola koga čine najmanje 500 gejtova, a koristi se, za gradnju većih i kompleksnijih aplikacija na čipu.

Digitalni sistemi vremenom postaju sve složeniji i samim tim javlja se potreba za razvijanjem složenih mehanizama za verifikaciju. U početku je korišćen dirigovani metod verifikacije, ali su se vremenom razvile pseudo-slučajne metode verifikacije, koje se danas koriste:

Dirigovani metod je način verifikacije u kom se pravi poseban test za svako ponašanje modula koje želimo da testiramo i tačno se definišu pobude kojim se pobuđuje DUT. Ovako se stvara velika količina koda koja ipak ne garantuje pouzdanost i preciznost dobijenih rezultata. Naprotiv, može se desiti da neko realno ponašanje modula ostane neverifikovano jer je autor testova zaboravio da napiše odgovarajući test ili čak je zaboravio da predvidi neki scenario. Zbog toga se ovakav pristup vremenom potisnuo u drugi plan, koristi se vrlo retko i samo kada je neophodno i to u kombinaciji sa pseudo-slučajnim metodom.

Pseudo-slučajni metod je način verifikacije koji danas svi koriste jer je mnogo pouzdaniji od dirigovanog metoda. Pseudo-slučajno generisanje pobuda znači da pobuda može u svakom trenutku da ima bilo koju vrednost iz opsega vrednosti koji je definisan i ograničen tipom podatka kog je sama pobuda. Korišćenjem ovog metoda bi u idealnom slučaju trebalo da postoji samo jedan test za verifikaciju celokupnog modula sa svim njegovim funkcionalnostima. To jeste moguće zato što se u ovakvom pristupu DUT pobuđuje pseudo-slučajnim pobudama čime se obezbeđuje da će se desiti skoro sva moguća realna ponašanja i da će ona biti testirana. Kada koristimo pseudo-slučajni metod uvek možemo da napravimo i neki dirigovani test koji će da testira neko specifično ponašanje koje želimo da proverimo, a plašimo se da se neće dogoditi odgovarajuća kombinacija pobuda koja bi obezbedila da se dogodi željeni scenario. Na slici 2.3.1 je prikazana glavna razlika u pristupu između dirigovane i pseudo-slučajne verifikacije.

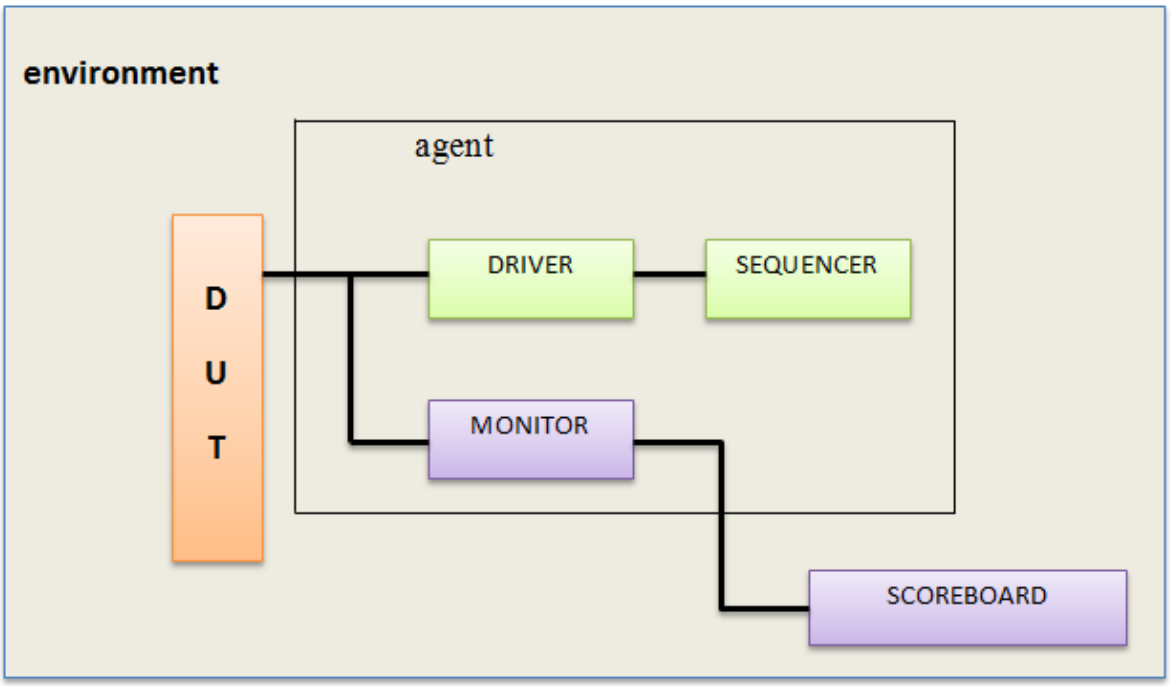
Trenutno dva najaktuelnija programska jezika za verifikaciju su SPECMAN eRM i sve popularnija UVM (*Universal Verification Methodology*). SPECMAN eRM je razvijena u kompaniji *Verisity Design* 2001. godine [4]. Nakon toga 2008. godine nastala je OVM (*Open Verification Methodology)* po ugledu na SPECMAN eRM ali je umesto *e* jezika [6] korišćen *SystemVerilog* [6]. Kasnije se kao derivat OVM-a razvija danas sve popularnija UVM . Koncepti su isti ili skoro isti kao i u eRM-u , ali prednost UVM-a se ogleda u tome što za *SystemVerilog* na tržištu postoji više alata. U master tezi koristiće se *System Verilog.*



Slika 2.3.1 Na gornjoj slici prikazana je dirigovana na donjoj pseudo slucajna verifikacija u idealnom slučaju

2.4 VERIFIKACIONE KOMPONENTE

Okruženje (*environment*) je ″kontejner komponenta″ za grupisanje podblokova (uvm komponenti) koji čine: sekvencer, drajver, monitor, skorbord (slika 2.4.1). UVM okruženje ne sadrži nikakvu dodatnu funkcionalnost, koristi se za povezivanje i kreiranje uvm komponenti. Environment klasa može da se koristi kao podokruženje drugog okruženja. Drajver i sekvencer su aktivne komponente, dok monitor i skorbord spadaju u pasivne komponente. Drajver preko virtuelnog interfejsa stimuliše DUT, dok monitor preko virtuelnog interfejsa prikuplja podatke, i šalje u skorbord gde se porede sa očekivanim vrednostima. Sekvencer je komponenta koja generiše transakciju i putem porta šalje transakciju drajveru. Agent je deo okruženja koga čine monitor, drajver i sekvencer.



Slika 2.4.1 Šematski prikaz okruženja sa verifikacionim komponentama

2.4.1 DRAJVER

Drajver je aktivna komponenta koja uzima sekvence pobuda od sekvencera kojima treba da se pobudi DUT. Drajver komunicira sa sekvencom preko sekvencera putem porta, koji se u System Verilog-u oznacava kao *uvm_seq_item_pull_port* .

Komanda *get_next_item* se poziva iz drajvera i vraća pokazivač na sekvencu koja će biti korišćena kao pobude DUT-a, komandom *item_done* kompletira se poziv sekvence [5].

2.4.2 MONITOR

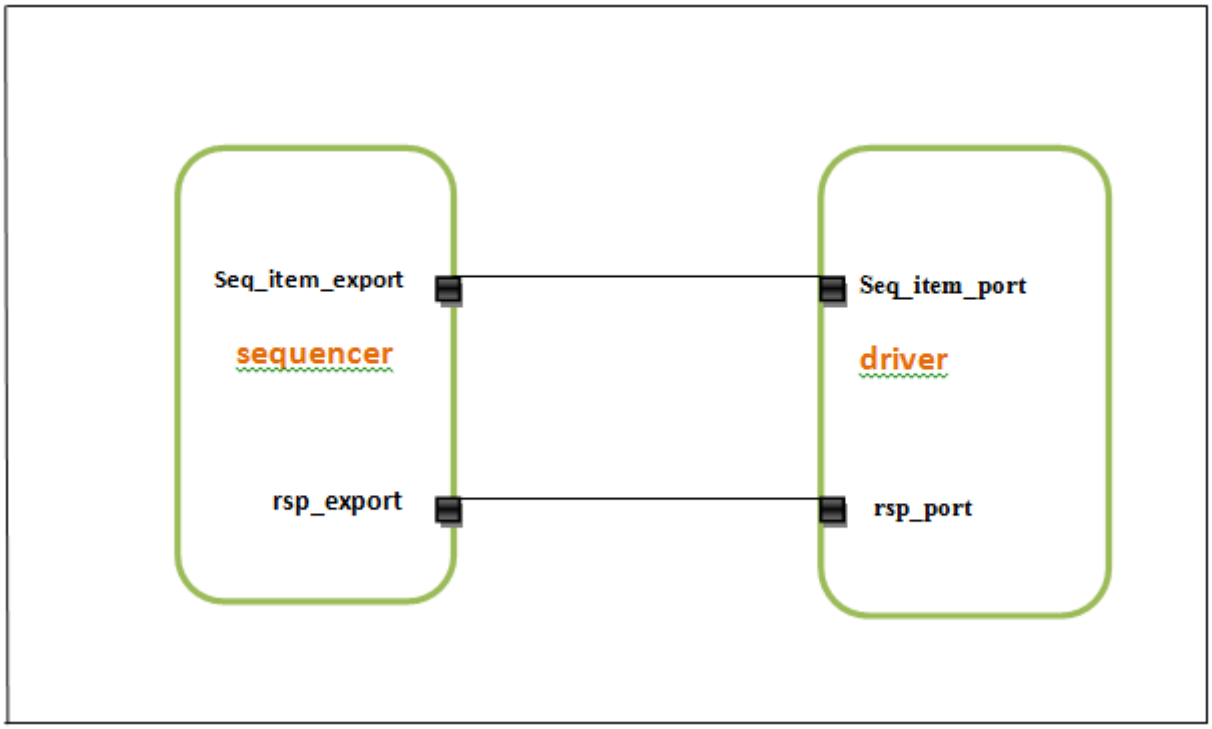
Monitor je pasivna komponenta. Monitor skuplja vrednosti sa portova DUT-a, pravi od njih transakcije i onda te vrednosti koristi za pravljenje raznih provera. Monitor se u agentu realizuje kao zasebna klasa. Transakcije koje se naprave u monitoru najčešće se šalju u skorbord gde se vrše razne provere podataka.

Monitor pokrivenosti - komponenta koja sadrži jednu ili više grupa pokrivenosti koje se koriste za prikupljanje informacije o funkcionalnoj pokrivenosti *(functional coverage)* u vezi sa onim što se dogodilo u testbenču tokom testa [5].

2.4.3 SEKVENCER

Sekvenca je specijalni tip transakcije koji dolazi do drajvera putem porta i stimuliše DUT tokom testa. Kada se sekvencer startuje, korisnički definisana sekvenca, tj. sekvenca koju je napravio verifikator po pravilima datim u specifikaciji čipa, biće pokrenuta. Ukoliko se ″premoste″ korisnički definisane sekvence tokom transakcije, onda se generišu slučajno definisane sekvence. To su sekvence čija polja uzimaju slučajno izgenerisane vrednosti. Sekvencer poseduje *sequence_item_export* preko koga se vrši povezivanje sekvencera i drajvera. U System Verilogu sekvencer se realizuje kao klasa. UVM Drajver ima 2 TLM (*Transaction Level Model*) porta:

- *seq_item_port*, port preko koga driver dobija transakcije od sekvencera, takođe drajver moze da koristi ovaj port da pošalje odgovor *(response*) da je primio sekvencu.
- *rsp_port*: Drajver šalje *response* sekvenceru.



Slika 2.4.2 Graficki prikaz portova sekvencera i drajvera

2.4.4 SKORBORD

Skorbord je verifikaciona komponenta koja određuje da li DUT radi ispravno. Prediktori su komponente koje generišu očekivane vrednosti koje skorbord koristi da bi poredio sa aktuelnim vrednostima. U svakom projektu skorbord se realizuje kao klasa koja potiče iz *uvm_scoreboard* klase, koja je prazna, ugrađena u bazu komponenata u uvm biblioteci. Najčešće je definisan kao lista i povezan je na dva ili više monitora. Uglavnom je implementiran tako da od jednog monitora

uzima podatke i smešta ih u listu i zatim vrednosti iz liste poredi sa očekivanim vrednostima, odnosno vrednostima koje se nalaze u drugom monitoru [5].

2.5 UVM FAZE

UVM komponente izvršavaju svoje ponašanje prema unapred definisanim fazama. Svaka faza definiše svoj sopstveni virtuelni metod.

- *virtual function void build()* ova faza se koristi za konstrukciju i konfiguraciju portova, komponenata i eksporta (vrsta porta koja prihvata transakcije).
- *virtual function void connect()* u ovoj fazi se vrši povezivanje portova i eksportova komponenti iz TB (*Testbench*).
- *virtual function void end_of_elaboration()* faza koja se koristi za podešavanje komponenata ako je potrebno.
- *virtual function void start_of_simulation()* početak simulacije.
- *virtual task run()* faza u kojoj se izvršava glavni deo programa sa svim petljama.
- *virtual function void extract()* u ovoj fazi se prikupljaju sve potrebne informacije
- *virtual function void check()* u ovoj fazi vrši se provera informacija dobijenih npr. iz skorborda ili čitanje pojedinih registara.
- *virtual function void report()* Faza u kojoj se generišu izveštaji o uspehu ili neuspehu simulacije.

Jedino build faza se izvršava odozgo na gore (*top down manner*), dok sve ostale suprotno (*bottom up manner*). Run faza je jedina faza koja je TCM (*Time Consuming Method*), što znači da poznaje vremensku zavisnost, u toj fazi se generišu transakcije čiji se vremenski oblici menjaju tokom vremena. U run fazi se pišu događaji (*events*) koji efikasno opisuju komunikaciju između dva istovremena aktivna procesa [5].

2.6 TRANSAKCIJE

Sekvenca je specijalni oblik transakcije koja stimuliše DUT tokom testa. Kao što je već ranije napomenuto drajver dobija sekvence putem sekvencera i preko virtuelnog interfejsa te transakcije šalje na DUT. Virtuelni interfejs je zamišljena žica koja povezuje drajver i DUT, ali preko te žice monitor (jedan ili više njih) prihvata dobijene transakcije i šalje u skorbord da se porede sa očekivanim vrednostima. Svaka transakcija u UVM-u počinje sa :

*class my_transaction **extends** uvm_sequence_item;*

'uvm_object_utils (my_transaction);

Rezervisana reč ***extends*** upućuje na to da je klasa *my_transaction* proširenje klase *uvm_sequence_item*. Druga linija koda registruje našu transakciju u fabrici (*factory*). Fabrika je apstraktan pojam u UVM-u koja generiše transakcije po tačno definisanim pravilima od strane programera. Nakon definicije klase dolazi deklaracija promenljivih. To čine polja *(fields*) koja su delovi transakcije. Ova polja se uglavnom randomizuju, tj.generišu se slučajne vrednosti, kao vrednosti datog polja. Naredne linije koda predstavljaju primer polja naše transakcije. Sva polja su randomizovana, što podrazumeva rezervisanu reč *rand* ispred tipa promenljive [5].

rand bit cmd;

rand int addr;

rand int data;

Prvo polje (*cmd*) je definisano kao bit, što znači da može uzimati vrednosti 0 ili 1, druga dva polja (*addr i data*) su tipa int (*integer*), 16-bitni broj koji obuhvata opseg -32 768 do 32 767. Ponekad je potrebno da se randomizovane vrednosti ograniče na određeni opseg vrednosti, to se radi ograničenjima (*constraints*). U narednim linijama koda dat je primer ograničenja naših polja.

constraints c_addr {addr>0; addr<256;}

constraints c_data {data>0; data<256;}

Ovim ograničenjima su promenljive *addr* i *data* ograničene u opsegu od 1 do 255.

function new (string name=" ");

super.new(name);

endfunction:new;

...

endclass my_transaction;

Funkcijom *new* vršimo alokaciju objekata, tj. dodeljujemo mesto u memoriji svim definisanim objektima. Funkcija *new* predstavlja konstruktor koji se poziva uvek kada se objekat kreira, data funkcija ne vraća nikakvu vrednost. Svaku kreiranu klasu moramo završiti rezervisanom rečju ***endclass*** i ime klase.

Kada kreiramo sekvencu, to se radi u okviru klase :

class my_sequence extends uvm_sequence (# my_transaction);

'uvm_object_utils (my_sequence);

function new (string name=" ");

```
    super.new(name);
endfunction:new;
task body;
   forever begin
     my_transaction tx;
     t x= my_transaction::type_id::create("tx");
    start_item("tx");
   assert (tx.randomize());
   finish_item(tx);
end
endtask:body;
```

Klasa *my_sequence* je proširenje *uvm_sequence* klase kojoj se prosleđuje već kreirani tip transakcije *my_transaction*. Ovu sekvencu registrujemo u fabrici, a zatim funkcijom *new* vršimo alokaciju objekta. U najbitnijem segmentu gore navedenog koda *task body* opisuje se ponašanje sekvence. Celokupan task smešten je u beskonačnu petlju. Objekat *tx* je instanca klase *my_transaction*. Komandom *start_item("tx");* drajver je spreman da primi transakciju, dok *assert (tx.randomize());* započinje randomizacije svih polja u transakciji *my_transaction* koji su definisani sa *rand*. Komandom *finish_item(tx);* drajver je primio sve očekivane transakcije.

Drajver, kao što je ranije napomenuto se takođe realizuje kao klasa, koja je proširenje klase *uvm_driver*. Klasa *my_driver* prihvata već kreiranu transakciju *my_transaction*. Naredni kod prikazuje primer jednog drajvera u system verilog kodu [5].

```
class my_driver extends uvm_driver (#my_transaction);
'uvm_object_utils (my_driver);
virtual dut_if   dut_vi;
function new (string name=" ");
    super.new(name);
endfunction:new;
function void build_phase(uvm_phase phase);
...
```

```
function run_phase (uvm_phase phase);
    forever begin
   my_transaction tx;
  @(posedge dut_vi.clock);
    seq_item_port.get_next_item(tx);
    dut_vi.cmd=tx.cmd;
   dut_vi.addr=tx.addr;
  dut_vi.data=tx.data;
   seq_item_port.item_done();
end
endtask run_phase;
```

Run faza je najbitnija za klasu drajvera. Tu se dešava celokupna transakcija. U ovoj fazi na početku je postavljen *event* gde se čeka pozitivna ivica takt signala *@(posedge dut_vi.clock)*, a komandom *seq_item_port.get_next_item(tx);* drajver je spreman da prihvati transakciju sekvencera. Polja *cmd, addr i data* koji su *random* generisani u transakciji povezani su na žice virtuelnog interfejsa koja te vrednosti šalje na DUT. Komandom *seq_item_port.item_done();* drajver završava preuzimanje transakcije. Ovim je završena run faza.

2.7 SEKVENCE I TESTOVI

Kao što je već ranije rečeno da bi smo kreirali sekvencu neophodno je da napravimo klasu, koja uglavnom nosi ime sekvence i predstavlja proširenje *uvm_sequence* klase. Zatim komandom *'uvm_object_utils (my_sequence);* tu sekvencu registrujemo u fabrici, a pozivom konstruktora *new* vršimo kreiranje objekata. U *task body* segmentu koda (iz potpoglavlja 2.7) vrši se generisanje svih vrednosti polja u sekvenci definisanih sa *rand* . Ako posmatramo deo koda:

```
task body;
forever begin
    my_transaction tx;
    tx= my_transaction::type_id::create(″tx″);
    start_item(″tx″);
    assert (tx.randomize() with{ tx.cmd=READ; a=tx.addr; });
```

```
    finish_item(tx);
end
endtask :body;
```

U liniji koda *assert (tx.randomize() with{ tx.cmd=READ; a=tx.addr; });* vrši se randomizacija svih polja *tx* transakcije ali polja *addr* i *cmd* bivaju premoštena, i dobijaju vrednosti zadate u samoj sekvenci, READ i *tx.addr*, tj ona se ne generišu slučajno i to se postiže naredbom *with*.

U okviru jedne sekvence može se pozvati i druga sekvenca i može se pristupiti njenim poljima. Ako posmatremo samo *task body* neke definisane sekvence, tada:

```
class seq_of_commands extends uvm_sequence (#my_transaction);
'uvm_object_utils (seq_of_commands);
function new (string name=″ ″);
     super.new(name);
endfunction:new;
```

task body

```
 repeat (n);
begin
    read_modify_write_seq seq;
    seq= read_modify_write_seq::type_id::create(″seq″);
    seq.start(m_sequencer,this);
end
endtask:body
```

U task body definisana je nova sekvenca *read_modify_write_seq*, čiji je objekat *seq*, a kreiran je sa

seq= read_modify_write_seq::type_id::create(″seq″);

Prilikom startovanja sekvence naveli smo *m_sequencer*. To je sekvencer koji salje *read_modify_write_seq* na drajver, dok *this* predstavlja referencu na sekvencer *seq_of_commands* sekvence [5].

Sekvence pokrećemo iz testa. Test se u UVM-u realizuje kao klasa, koja je proširenje *uvm_test* klase. Naredni deo koda prikazuje primer jednog testa u kojem se poziva sekvenca koja je definisana prethodno u tekstu [5].

```
class test1 extends uvm_test;

'uvm_component_utils (test1);

my_env my_env_h;

my_agent my_gent_h;

...

task run_phase (uvm_phase phase);

read_modify_write_seq seq;

seq= read_modify_write_seq::type_id::create(″seq″);

phase.rais_objection(this);

seq.start(m_env_h, m_agent_h, my_sequencer_h);

phase.drop_objection(this);

endtask

endclass
```

Sekvenca koja se startuje iz testa prosleđuje se na sekvencer koji se nalazi u okruženju *(m_env_h),* u agentu *(m_agent_h),* i sekvenceru *(my_sequencer_h),* ovde ne postoji referenca *this* jer sekvenca koja se startuje iz testa *read_modify_write_seq* nema roditeljsku sekvencu. Funkcije *rise_objection* i *drop_objection* (ove funkcije biće objašnjene u potpoglavlju 2.11) pozivamo isključivo u testovima pre i posle puštanja sekvence zato što jedino kroz test biramo koji stimulus treba da se izvrši.

Prilikom pozivanja sekvence u nekom testu, *assert (seq.randomize)* generišu se slučajne vrednosti svih polja sekvence *seq* definisahih sa *rand*, međutim u testu može biti ugašena randomizacija, i datim poljima moguće je dodeliti konkretne vrednosti. To se postiže:

```
seq.how_many.constraint_mode(0);

assert(seq.randomize) with {seq.n=10;}
```

Vrednost polja *n* u sekvenci *seq* čija se vrednost slučajno generisala u određenom opsegu, gašenjem ograničenja komandom *seq.how_many.constraint_mode(0),* dodelili smo mu konkretnu vrednost 10.

2.8 MONITOR I POKRIVENOST

2.8.1 POKRIVENOST KODA

Kodna pokrivenost *(code coverage)* je tehnika koja omogućava da se identifikuje koji delovi koda su izvršeni tokom simulacije u dizajnu koji se verifikuje. Ovo je veoma važno zato što tokom verifikacije testbenč može da prijavi pozitivne rezultate zato što se neki delovi koda uopšte nisu izvršili. Tada se može poverovati da je sa dizajnom sve u redu, iako može biti potpuno suprotno. Ovo je veoma opasno. Zato su napravljeni alati pokrivenosti koda.

Izvorni kod se prvo instrumentizuje, što znači da se beleže ključna mesta u kodu da bi se omogućilo sakupljanje rezultata. Instrumentizacija koda je bitna samo za dizajn koji se verifikuje, dok je nebitna za sve ostalo što se ne verifikuje kao što su testbenčevi na primer. Veliki procenat koda testbenča se izvršava samo ako se desi neka greška u dizajnu. Potom se simulira instrumentacioni kod, a bitni podaci se sakupljaju u bazu podataka. Na osnovu ove baze podataka moguće je napraviti izveštaj o raznim statistikama izvršavanja koda. Neke od tih statistika mogu biti pokrivenost izraza ili puteva unutar koda (objašnjeno kasnije u tekstu). Kompletnost izvršavanja koda ne znači i njegovu korektnost. Recimo, kompajler može optimizacijom ukloniti neke linije koda. To može dovesti do pogrešnog tumačenja rezultata. Na primer, možemo dobiti da je izvršeno 100% linija koda, ali samo zato što je 20% linija uklonjeno tokom optimizacije. Kada se radi simulacija sa pokrivenošću koda potrebno je isključiti optimizaciju. Linijska pokrivenost je najjednostavnija. Ona pokazuje koje linije koda su se izvršile, a koje nisu. Pokrivenost puteva pokazuje koji putevi su se ostvarili unutar koda od svih mogućih. Pokrivenost izraza govori o tome koji izrazi unutar koda su bili istiniti [4].

2.8.2 MONITOR I MONITOR POKRIVENOSTI

Zadatak monitora je da prepozna transakciju i pokupi tu transakciju preko virtuelnog interfejsa, gde će se u skorbordu te transakcije porediti sa očekivanim vrednostima. Svaki monitor ima svoj *analyisis port* (*ap*) preko koga šalje dobijene transakcije. Subscriber ima svoj *analysis export*, port preko koga skuplja dobijene transakcije monitora. Subscriber je UVM komponenta koja može biti skorbord, kolektor pokrivenosti ili nešto drugo. Monitor kao i druge UVM komponente definišemo kao klasu koja proširena iz *uvm_monitor*. Monitor kao formiranu klasu, registrujemo u uvm fabrici [5]. Posmatrajmo deo koda:

```
class m_monitor extends uvm_monitor;

‘uvm_component_utils (m_monitor)

uvm_analysis_port (#my_transaction) aport;

virtual dut_if dut_vi;

function new…
```

```
function void built_phase (uvm_phase phase);

aport= new (″aport″ this)

…
```

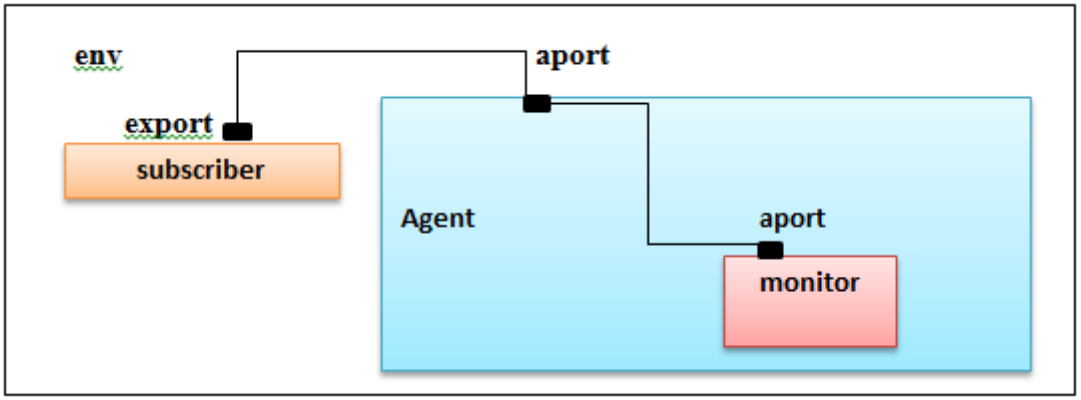
U klasi *m_monitor* definiše *analysis port* koji je označen kao *aport* koji prosleđuje transakcije tipa *my_transaction*. Virtuelni interfejs kao deo monitor klase je neophodan, jer preko njega monitor vidi transakcije koje se prosleđuju na DUT. Definiše se konstruktor *new* i naravno vrši se alokacija objekta *aport* komandom *new*. Posmatrajmo nastavak koda i run fazu.

```
...

task run_phase (uvm_phase phase);

forever begin

   my_transaction tx;

  @(posedge dut_vi.clock);

  tx=my_transaction::type_id::create(“tx”);

  tx.cmd=dut_vi.cmd;

  tx.addr=dut_vi.addr;

  tx.data=dut_vi.data;

  aport.write(tx);
```

Polja transakcije *tx* (*cmd, addr i data*) sa virtuelnog interfejsa se šalju na *aport* monitora funkcijom *aport.write (tx)* i dobijena transakcija šalje se na subscriber, gde se vrši upoređivanje podataka. To se radi u skorbordu. Agent, je deo okruženja kojeg čine monitor, drajver i sekvencer, takođe poseduje port. Ako posmatramo:

```
class my_agent uvm_agent;

…

uvm_analysis_port (#my_transaction) aport;

function void built_phase (uvm_phase phase);

   aport= new (″aport″ this)

   my_monitor_h = my_monitor::type_id::create(″m_monitor_h ″);

   my_driver_h = my_driver::type_id::create(″m_driverr_h ″);
```

```
    my_sequencer_h = my_sequencer::type_id::create(″m_sequencer_h ″);
endfunction:build_phase
function void connect_phase (uvm_phase phase);
…
my_monitor_h_h.aport.connect(aport);
```



Slika 2.8.1 Šematski prikaz povezivanja portova monitora i agenta sa exportom subscribera

U klasi *my_agent* definisan je objekat *aport.* Takođe kreirane su komponente monitor, drajver i sekvencer.U liniji koda *my_monitor_h_h.aport.connect(aport);* vrši se konekcija između aporta monitora i aporta agenta, to je potrebno zbog prosleđivanja transakcije do subscribera.

U okruženju definišemo agent i subsciber, koji na primer može biti skorbord. Okruženje je takođe realizovano kao klasa.

```
class my_env extends uvm_env;
my_agent my_agent_h;
my_subscriber my_subscriber_h;
...
function build_phase(uvm_phase phase);
    my_agent_h = my_agent::type_id::create(″m_agent_h ″);
    my_subscriber_h= my_subscriber::type_id::create(″m_subscriber_h ″);
```

```
endfunction:build_phase;

function void connect_phase (uvm_phase phase);

my_agent_h.aport.connect(my_subscriber_h.analysis_export);

endfunction:connect_phase
```

U bild *(build)* fazi kreirali smo agent i subscriber (koji može biti monitor pokrivenosti ili skorbord), zatim se u konekt *(connect)* fazi vrši spajanje aporta agenta sa analysis exportom subscribera. Ovim je završena konekt faza. Subscriber koji će u našem slučaju biti kolektor pokrivenosti, realizovaće se kao klasa:

```
class my_subscriber  extends uvm_subscriber #(my_transaction);

 ‘uvm_component_utils (m_subscriber)
```

Unutar subscriber-a definešemo promenljive čije vrednosti ubacujemo unutar grupe pokrivenosti.

```
bit cmd;

int addr;

int data;
```

Formiramo cover grupu:

```
covergroup cover_bus;

coverpoint cmd;

coverpoint addr { bins  a[16] = {[0:255];} }

coverpoint data { bins  d[16] = {[0:255];} }

endgroup cover_bus;

function void write (my_transaction t);

…

cmd = t.cmd;
addr = t.addr;        } coverage registers
data = t.data;

covergroup cover_bus.sample ();
```

endfunction;

U registru pokrivenosti upisuje se vrednosti *cmd, addr, data* prilikom jedne transakcije koja se odabira (sempluje) i biva memorisana u bazi podataka pokrivenosti. Na taj način se u grupama pokrivenosti pravi i prikazuje pokrivenost neke promenljive tj, koliko se puta pojavila svaka od mogućih vrednosti date promenljive tokom simulacije. Pokrivenost se izražava procentualno. Prilikom izrade nekog projekta uvek se teži pokrivenosti od 100%. Dešava se da pokrivenost bude manja od 90%, u tom slučaju posao verifikatora je da ispravi greške u kodu i da poveća pokrivenost neke od grupa tako što će na primer staviti da se na drugom mestu u kodu vrši semplovanje neke grupe pokrivenosti ili povećati verovatnoću pojavljivanja neke od vrednosti promenljivih.

2.9 UVM IZVEŠTAJI

Da bismo bili sigurni šta se dešava unutar sistema, kao i pri procesu debagovanja nekog problema koji se javi u procesu verifikacije, potrebno je štampati poruke i vrednosti promenljivih koje se generišu u određenom delu koda. Najčešće se prave funkcije u kojima se štampaju izveštaji.

function void write (my_transaction t);

`uvm_info (″mg″ , transaction received , UVM_NONE);

Severity originator/ID string verbosity

Ozbiljnost *(severity)* predstavlja tip poruke koji želimo da prikažemo na ekranu, može biti *`uvm_info, `uvm_error, `uvm_fatal `uvm_worning*. Info tip poruke uglavnom daje iskaze o vrednostima promenljive, obaveštava nas da li je određeni deo koda prošao kroz neku od petlji, da li je započeo neku od uvm faza kao i to da li je kod prošao kroz granu od interesa. Ovi izveštaji mogu se koristiti kao jedan od metoda debagovanja. Error je tip poruke koja javlja grešku, na primer ako se porede dve vrednosti, i trenutna vrednost ne bude ista kao očekivana u procesu verifikacije onda se javlja poruka o grešci. Ovaj tip poruke je najčešće u skorbordu gde se vrednosti transakcije porede. Upozorenje na neko dešavanje je tip poruke koji daje informaciju. Na primer ako tokom simulacije neke vrednosti promenljivih budu pregažene. Originator /ID predstavlja ime firme koja izvodi projekat, ili inicijali inženjera koji je kreirao program ali može biti i šifra poruke koja je određena u verifikacionom planu, tj.dokumentu koji piše verifikator, gde na primer svaka poruke greške ima svoj opis. String je poruka koju kreira inženjer koji piše program. Verboziti kontroliše proritet poruke. Postoje više vrsta verbozitija: [1].

- UVM_NONE - verboziti koji ne štampa u log fajlu (fajl u kome se kreiraju izveštaji tokom simulacije) poruke koje imaju označen taj verboziti.

- UVM_LOW - verboziti koja prikazuje sve LOW poruke u simulaciji, tj.sve poruke čiji je verboziti označen sa LOW.
- UVM_MEDIUM - verboziti koji štampa sve medium poruke kao i LOW poruke (ako postoje) u simulaciji
- UVM_HIGH - verboziti koji prikazuje sve HIGH ali i LOW i MEDIUM poruke u simulaciji.
- UVM_FULL – verboziti koji će se uvek prikazati u log fajlu, uvek će štampati poruku sa ovim verbozitijem.

Ovi verboziti oblici koriste se pri filtriranju poruka. Ukoliko ne želimo da se sve poruke prikažu tokom simulacije i pri pokretanju testova, jer ih može biti par hiljada, onda podesimo verboziti na neki koji ima manji prioritet kao na primer uvm_low, i onda se neće prikazati MEDIUM i HIGH poruke i na taj način smanjujemo vreme simulacije i olakšavamo debagovanje.

2.10 MODELI TRANSAKCIJE

Pod strukturom podrazumevamo testbenč, prostor i komponente koje se koriste pri verifikovanju DUT-a. Pod ponašanjem podrazumevamo sekvencu koja će stimulisati DUT tokom testa. Transakcije su glavno komunikaciono sredstvo u određenom prostoru. Moguće je dodati, modifikovati ili oduzeti stimuluse tokom transakcije nezavisno od testbenča. Postoji mogućnost da više sekvenci pokrenemo u paraleli kao i da pozovemo roditeljsku sekvencu u nekoj od sekvenci. Posmatrajmo kod koji prikazuje jednu transakciju [4]

```
class bus_item extends sequence_item;
‘uvm_object_utils (item_bus)
rand int delay;
rand logic[31:0] address;
rand op_code_enum op_code;
rand logic[31:0] data[];
string slave_name;
bit response;
function new (string name=″ bus_item″);
super.new(name);
endfunction:new;
```

sva polja transakcije se randomizuju

do_copy();

do_compare(); metodi standardnih operacija

convert2string();

do_print();

do_record();

do_unpack();

endclass: bus_item;

Pored standardnih načina definisanja transakcije, kao i registrovanje transakcije u fabrici,pozivamo i konstruktor *new* koji vrši alokaciju objekata u datoj klasi. Navedeno je i nekoliko metoda koje se sprovode u transakcijama koje ćemo detaljno objasniti u narednom primeru.

Metod *do copy()* je odgovoran za kopiranje sadržaja jedne transakcije u drugu. To je virtuelni metod zaslužan za kopiranje jedne transakcije u drugu. Transakcije koju kopiramo je RHS *(rand hand side*). Zatim se u kodu vrši kastovanje sekvence, zatim se poziva funkcija *super.do_copy()* gde se svaki sadržaj bazične transakcije kopira u transakciju trenutne klase.

```
class bus_item extends uvm_sequence_item;
 `uvm_objest_utils (bus_item);
function void do_copy (uvm_objects  rhs)
bus_item  _rhs;
if (!  $cast (_rhs, rhs)) begin
uvm_report_error (“do_copy” “Cast failed”);
return;
end
super.do_copy(rhs);
delay = rhs_.delay;
address = rhs_.address;
data = rhs_.data;
op_code= rhs_.op_code;
```

sadržaj sekvence koji se kopira u tekuću klasu bus_item

```
response = rhs_.response;

slave_name = rhs_.slave_name;

endfunction:do_copy

endclass:bus_item
```

Funkcija *do_compare()* poredi dve transakcije. Posmatrajmo datu funkciju u već definisanoj klasi *bus_item*

```
function bit do_compare (uvm_object  rhs, uvm_comparer comparer);

bus_item _rhs;

if (! $cast (_rhs   rhs)) begin

return 0;

end

return ( ( super.do_compare(rhs,comparer) &&

(delay == rhs_.delay) &&

(address ==rhs_.address) &&

(op_code ==rhs_.op_code) &&

(slave_name == rhs_.slave_name) &&

close_enough (data,rhs_.data);

(response ==rhs_.response) );

endfunction : do_compare;
```

porede se polja roditeljske klase sa poljima tekuće klase

Ako kastovanje nije uspelo funkcija vraća 0. Zatim sva polja od *rhs* (*right hand side*) poredi sa poljima tekuće klase. Funkcija *do_compare()* vraća vrednost 1 ako su sva polja koji se porede u dve transakcije jednaka.
Funkcija *convert2string()* koristi se za debagovanje ili štampanje poruka koje se odnose na sadržaj transakcije.

```
function string convert2string;

string s;

s= super.convert2string();
```

$sformat(s, %s\n delay, \t%0d\n addr, \t%0h\n op_code, \t%s\n slave_name, %s\n", s, delay, addr, op_code.name(), slave_name);

foreach (data[i]) begin

$sformat (s, %s data[%0d], \t%0h\n" s, i, data[i]);

end;

$sformat (s, %s response, \t%0h\n" s, response);

endfunction: convert2string();

Ova funkcija daje mogućnost da se štampa deo stringa ili string u celosti što omogućava lakše debagovanje i prikaz vrednosti pojedinih delova transakcije unutar koda.

Funkcija *do_record()* omogućava memorisanje polja transakcije u bazi podataka. Funkcije *do_pack()* i *do_unpack()* ne koriste se tako često. Veoma su korisne kada u jednoj transakciji želimo da preuzmemo vrednosti polja druge transakcije, onda se kreira funkcija *do_pack().* Kopiraju se vrednosti željene transakcije u transakciju od interesa. Funkcijom *do_unpack()* čitaju se vrednosti željene transakcije u trenutnoj transakciji.

2.11 PISANJE I UPRAVLJANJE TESTOVIMA

Okruženje u sebi sadrži monitor, drajver, sekvencer kao i skorbord i (ili) monitor pokrivenosti. Posao okruženja je da specificira podrazumevane sekvence i da ih pokrene. Posao testa je da stimuliše testbenč kao i to da premosti, izmeni ili prihvati sekvence koje su definisane u testbenču, ili da promeni tip pokrivenosti koji je od interesa.
Prvo ćemo se osvrnuti na definisanje okruženja u UVM-u.

class my_env extends uvm_env;

int nslaves;

test_seq seq_t;

my_slave slv_b[];

function void build_phase(uvm_phase phase);

if(!uvm_config_db#(int)::get(this,"","nslaves",nslaves);

nslaves=2;

slav_b=new(nslaves);

for (int i=0; i<=nslaves; i++) begin

```
$sformat (name, slave[%d]", i );

slv_h[i]=my_slave::type_id::create (name, this);

end

endfunction

task run_phase(uvm_phase phase)

seq_b=my_seq::type_id::create("my_seq");

seq_b.start(slv_h[0].sequencer);

endtask;

endclass
```

U bild fazi konfigurišemo sve ono što će se nalaziti u testbenču, kao na primer broj slejv agenata (agent koji ima ulogu čitanja podataka), u našem primeru to je 2, : *nslave [0]* i *nslave [1]*. Takođe u bild fazi definišemo sekvencu, slejv agent a u ran fazi pozivamo tu sekvencu koja se prosleđuje iz sekvencera.
Pogledajmo primer testa u UVM-u:

```
class base_test extends uvm_test;

'uvm_component_utils (base_test);

m_env e;

test_seq tseq_h;

function void build_phase (uvm_phase phase);

e=m_env::type_id :create("e" this);

endfunction

function void end_of_elaboration_phase (uvm_phase phase);

tseq_h =test_seq::type_id::create("tseq_h");

endfunction

endclass
```

Definisan je base_test koji je proširen iz *uvm_test*, i test je registrovan u fabrici. Zatim okruženje *e* iz koga će se pokrenuti sekvenca *tseq_h* koja se poziva u definisanom testu.

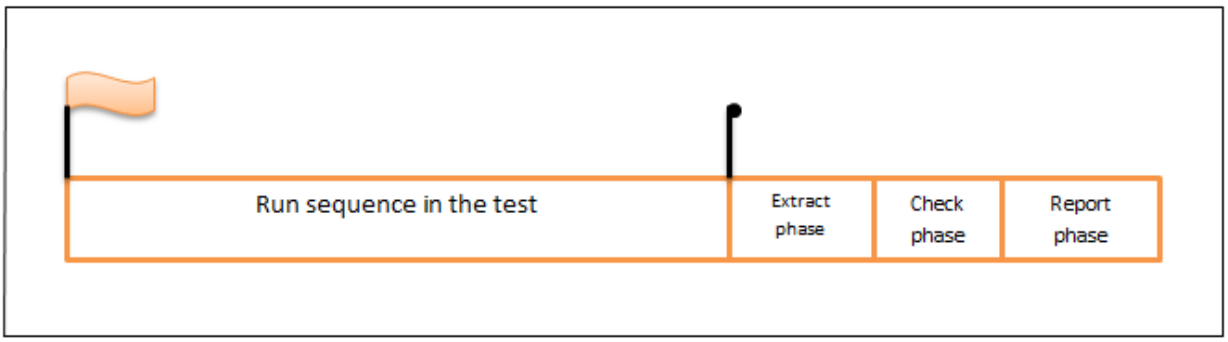
Kada pokrećemo test da se izvršava, moramo biti sigurni da će biti obezbeđen dovoljno veliki vremenski period u ran fazi da bi se izvršile sve sekvence pozvane u testu. Prilikom

startovanja run faze ″podiže″ se objekat da saopšti sistemu da su sekvence krenule da se izvršavaju, i da treba da se sačeka sve dok se potrebne definisane stvari u sekvencame ne izvrše. Bilo koja komponenta ili sekvenca može podići objekat. Kada su sve potrebne operacije gotove, ”spuštamo objekat” da signaliziramo sistemu da je sve uspešno završeno i da je kraj ran faze. Moramo ″podići objekat″ na početku faze, pre nego što bilo koje vreme protekne, u suprotnom, sistem će misliti da ne želimo ništa da uradimo i završiće ran fazu momentalno. Uvek kada se objekat “podigne” to utiče hijerarhijski, sve do testa. Podizanje objekta neke komponente automatski uslovljava komponentu koja je hijerarhijski više, da podigne objekat. Ukoliko sekvenca izvrši podizanje objekta, to automatski utiče i na njen sekvencer, pa i on izvrši podizanje objekta, takođe to uzrokuje da i agent izvrši *raise objection*. Ukoliko na primer, monitor izvrši podizanje objekata, to će uticati na agent, pa će on izvršiti podizanje još jednog objekta. Svi ovi objekti moraju biti spušteni pre nego što se faza završi. Proces spuštanja objekta (*drop object*) je hijerarhijski. Kada sekvenca spusti objekat, onda to uradi sekvencer, pa zatim agent.
UVM poseduje vreme isteka. Sistem može da čeka određeni vremenski period pre nego što se objekat spusti. Kada protekne vreme isteka (*drain timer*), tada se mora spustiti objekat sa višeg hijerarhijskog nivoa, na primer ako monitor spusti svoj objekat, tada za vreme isteka mora to učiniti i agent. Ovaj metod spuštanja objekta se ne preporučuje [5].

Preporuka je da se vrši drop i rise objection u toku pozivanja sekvence u samom testu.

```
class my_test extends uvm_test;

...

task run;

phase.rise_objection(this,“start sequence“);

vseq_h.start(null);

phase.drop_objection(this,“start sequence“);

endtask

endclass
```

Prethodno naveden kod omogućava testu da bude završen onda kada budu završene i sve sekvence u njemu. Podizanje objekta *phase.rise_objection(this,“start sequence“),* vrši se u trenutku kada se sekvenca poziva, a spuštanje objekta *phase.drop_objection(this,“start sequence“),* onda kada svi delovi transakcije budu završeni. Ovim se završava i test. Na slici 2.11.1 je grafički je prikazan period pokretanja sekvence koji počinje podizanjem objekta (podignuta zastavica) a završava se spuštanjem objekta. Nakon toga usleđuju faze ekstrakcije, provere i izveštavanja koje su opisane u potpoglavlju 2.5



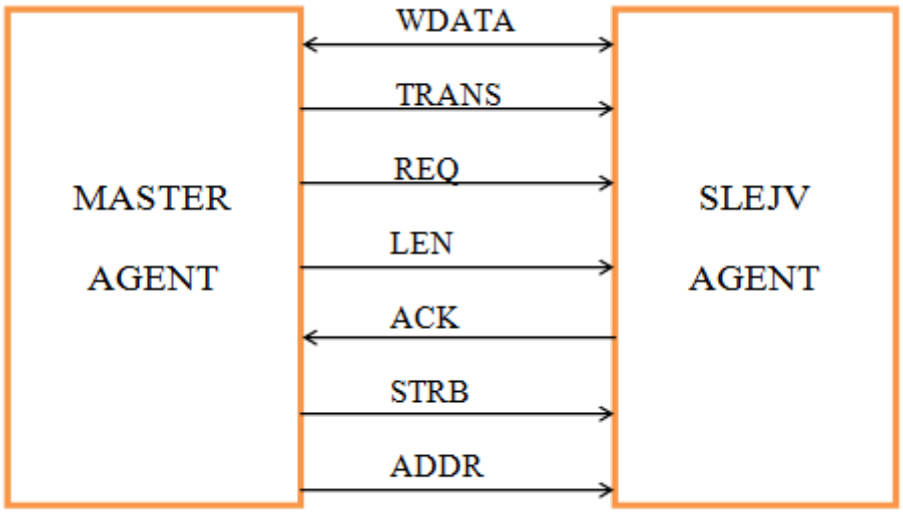
Slika 2.11.1 Grafički prikaz podizanja objekta na početku pokretanja sekvence kao i spuštanja na kraju run faze

3.SPECIFIKACIJA ČIPA

3.1 SIGNALI

Čip koji će se verifikovati je ASIC *(Application Specific Integrated Circuit)* čip, što zapravo znači da je dizajniran za specifičnu aplikaciju, umesto opšte namene čipa, kakav je mikroprocesor. ASIC čipovi ne spadaju u programabilne čipove. ASIC čipovi, kao što samo njihovo ime kaže, predstavljaju integrisane čipove specifične namene. Ovi čipovi se dizajniraju da obavljaju određenu funkciju ili skup funkcija i ne mogu se reprogramirati ili koristiti u druge svrhe (tj. obavljati druge funkcije) kao programabilni čipovi. Prednost ASIC čipova u odnosu na programabilne čipove je što su optimizovani za funkciju koju obavljaju pa samim tim efikasnije troše resurse, nema neiskorišćenih resursa kao kod programabilnih čipova. Usled većih troškova proizvodnje ASIC čipovi se koriste samo za one funkcije za koje se zna da će biti implementirane u velikom broju uređaja tj. proizvodiće se veliki broj takvih ASIC čipova čime se prevazilazi problem troškova proizvodnje.

U našem slučaju, čip se sastoji od jednog master i jednog slejv agenta. Uloga *master* agenta je da vrši funkciju upisa (upis u registar), uloga *slave* agenta je da vrši funkciju čitanja (čitanje iz registra), odnosno da master šalje podatke slejvu, dok ih slejv čita.



Slika 3.1.1 Master i slejv agent povezani žicama

Master i slejv komuniciraju preko magistrale. Signali preko kojih komuniciraju master i slave su: TRANS, ADDR, STRB, WDATA, RESET, ACK, CLK, LEN. Svaki signal ide preko zasebne magistrale.

- TRANS - signal koji ide iz mastera,označava tip transakcije i ima širinu četiri bita.Vrednosti signala su:
  - 1- ″write transaction″, master šalje podatke slejvu.
  - 2- ″read transaction″, slejv čita podatke koje mu je master poslao.
  - 3- ″idle″ nedozvoljeno stanje
- ADDR - signal koji dolazi iz mastera, predstavlja adresu u koju se upisuju podaci koji se generišu. Širina signala je 4 bajta (32 bita)
- STRB - signal koji dolazi iz mastera, osmobitne velicine.
- WDATA - bidirekcioni signal, ide od mastera ka slejvu i obrnuto, veličina podatka je 64 bita.
- RESET - signal koji dolazi sa mastera, dok god je reset aktivan, tj.ima vrednost 1, prenos podataka nije moguć, kada se reset spusti na 0, signal je neaktivan, master počinje da šalje podatke.
- ACK - signal koji dolazi sa slejva, kada se obori ACK na nula slejv počinje čitanje podataka. Trajanje ACK signala je jedan takt.
- CLK - Takt signal, učestanosti 33 MHz.
- LEN - Četvorobitni signal, koji dolazi sa mastera, predstavlja dužinu podataka koji se prenose, ako je npr LEN=4, u jednoj transakciji prenosi se 5 WDATA podatka. Vrednost LEN se kreće od 0 do 15.
- REQ - signal jednog bita koji dolazi sa mastera i pada na 0, kada od slejva stigne pozitivna potvrda ACK, tada se započinje transakcija.
- STRB – Signal širine osam bita. Može imati najviše četiri jedinice. Šalje se od mastera ka slejvu. Broj jedinica određuje koliko će se upisati bajtova sa polja adrese u polje WDATA, počevši od bita najmanje težine. Vrednost 0 nije dozvoljena.

3.2 PROTOKOL

Protokol je pravilo prenosa podataka između dva odvojena entiteta. U datom čipu korišćen je korisnički definisan protokol (*propriatery protocol*), tj. protokol definisan od strane dizajnera čipa.

STRB signal je definisan u određenom opsegu, protokol se bazira na tome da zavisno od broja jedinica u STRB signalu, sa toliko bajtova popuni polje WDATA, vrednostima adrese signala počevši od bita najmanje težine.

1 1 0 1 0 0 1 0	1 1 1 1 0 0 0 0	**1 0 0 0 1 1 0 1**	**0 1 0 0 1 1 1 0**

Polje adrese od 4 bajta

1 1 0 0 0 0 0 0

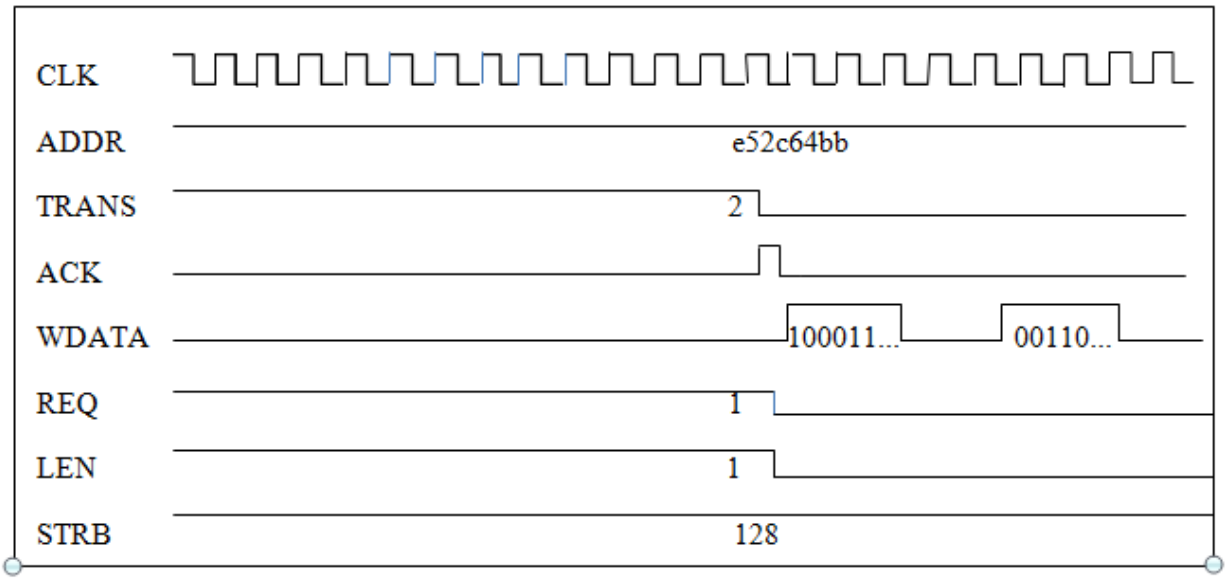
Polje strb signala od 1 bajta

11001110	11000101	01010101	00110101	00011010	11100010	**10001101**	**01001110**

Polje wdata podatka od 8 bajtova

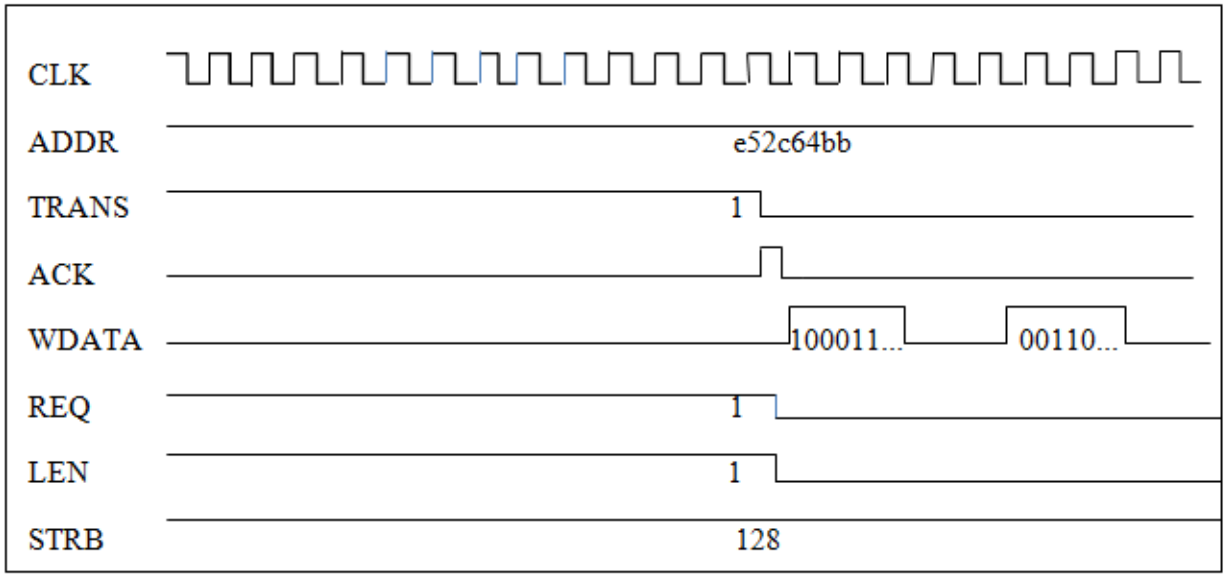
Kao što se vidi na primeru, pošto STRB ima dve jedinice, iz polja adrese uzimaju se dva bajta najmanje težine i upisuju se u WDATA na poziciji bajta najmanje težine, preostale pozicije u WDATA podatku su slučajno generisane vrednosti.

Master agent vrši funkciju upisa podataka. Kada tip transakcije bude TRANS=1, tada se vrši pisanje podataka. Kada se RESET signal postane neaktivan, njegova vrednost je tada 0, u tom trenutku master počinje pisanje podataka, šalje se LEN+1 podatak veličine 64 bita. Svaki od LEN+1 wdata podataka upisan je na adresi ADDR, koja se generiše pri svakoj novoj transakciji . Kada je vrednost TRANS=2, tada slejv vrši čitanje podataka. Iz slejv agenta dolazi signal potvrde prijema ACK, kada vrednost ACK signala padne na 0 započinje proces čitanja LEN+1 podataka. Vrednost trajanja ACK potvrde mora biti 1 takt. Ukoliko sistem izgeneriše više od 50 pozitivnih ivica takta, a nije primljena ACK potvrda, započinje se nova transakcija sa novih LEN+1 podataka. Na narednoj slici prikazana je jedna od transakcija. U toku neke od transakcija ADDR, TRANS, LEN signali ne smeju menjati vrednosti.



Slika 3.1.2 Prikaz signala sa interfejsa pri transakciji čitanja

Na slici 3.1.2 prikazan je talasni oblik pri transakciji čitanja podataka. Pri jednoj transakciji adresa zadržava istu vrednost. TRANS signal je u ovom slučaju 2, što znači da je u pitanju funkcija čitanja. ACK signal traje tačno jedan takt. Broj WDATA podataka je u prikazanom primeru 2, zato što je vrednost LEN signala jednaka jedinici. U trenutku kada REQ signal padne na nulu, tada započinje čitanje podataka. STRB signal je u binarnom zapisu 10000000.



Slika 3.1.3 Prikaz signala sa interfejsa pri transakciji pisanja

Na slici 3.1.3 prikazan je talasni oblik pri transakciji pisanja. Slika se od slike 3.1.2 razlikuje samo u TRANS signalu, koji je u ovom slučaju jednak jedinici. Vrednost TRANS=1, ukazuje na to da je u pitanju transakcija upisa podataka. WDATA podaci koji se upisuju, moraju biti jednaki onima koji pročita slejv agent. Ukoliko to nije ostvareno postoji mogućnost da je greška u samom dizajnu čipa. Tada verifikator ukazuje na potencijalne greške.

4.VERIFIKACIJA ČIPA SA KORISNIČKIM PROTOKOLOM

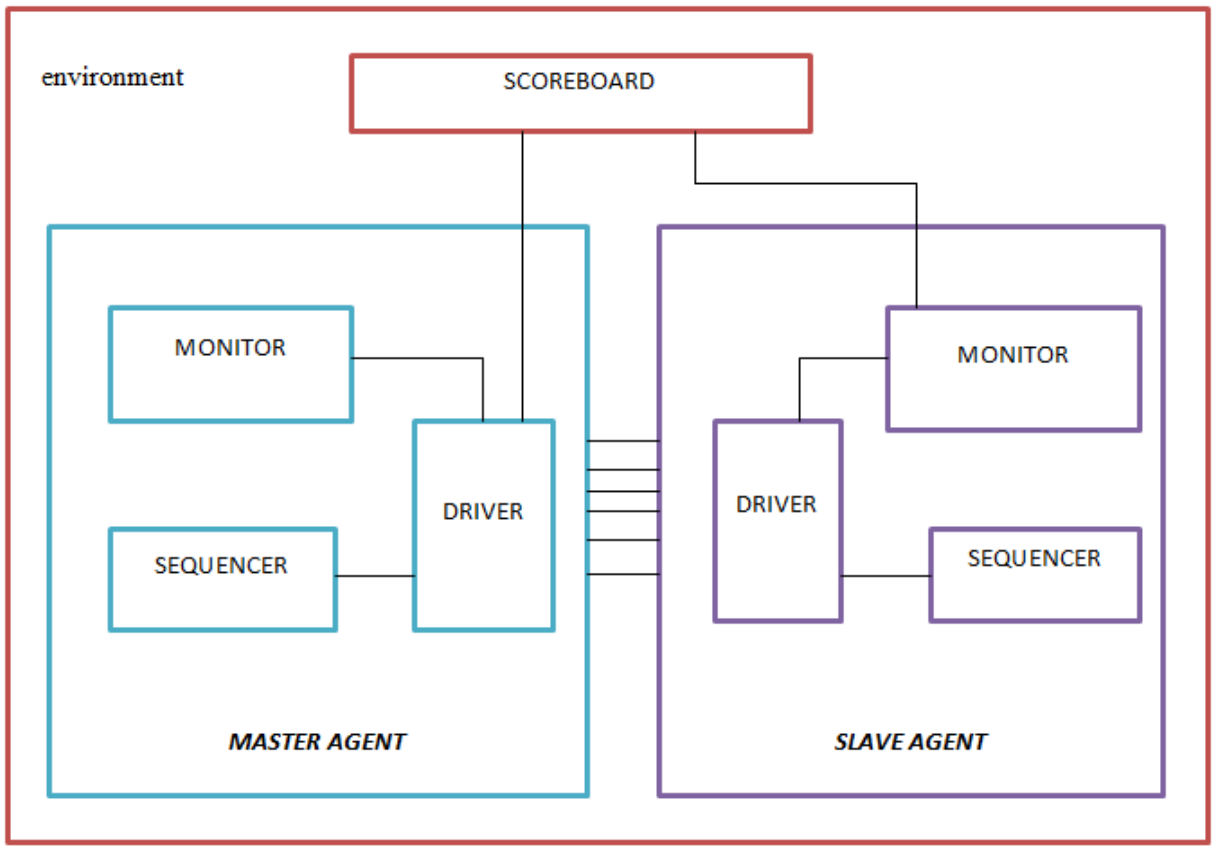
4.1 VERIFIKACIJA ČIPA

Celokupan kod koji čini proces verifikacije smešten je u folderu *example_uvc*. Kodovi su grupisani u foldere koji čine posebne celine u okviru *example_uvc*. U example_uvc nalaze se folderi *verification, makefile i agent*. U *verification* folderu smešteni su fajlovi *u2u_virtual_sequencer.sv, u2u_scoreboard.sv, u2u_test_base.sv, u2u_test_base.pkg* kao i folder *sim* gde se smeštaju log fajlovi nakon simulacije. Makefile folder, gde je istoimeni fajl koji služi za povlačenje fajlova pri kompajliranju (određuje redosled kompajliranja fajlova). Agent sadrži četiri foldera. Prvi je *seq* u kome je fajl *example_base_seq.sv*. Drugi je *tb* sa fajlom *example_top_tb.sv*. Treći je *env* folder koga čine fajlovi *env_config* (odredjuje koje komponente su prisutne u testbenču), *env_pkg.sv* i *env.sv*. Četvrti folder čine fajlovi sa verifikacionim komponentama čiji će kod, kao i njegova funkcija biti detaljno opisani u ovom poglavlju. Folder čine: *example_agent_config.sv example_agent_pkg.sv example_coverage_monitor.sv example_defines.sv example_driver.sv example_if.sv example_monitor.sv example_seq_item.sv example_sequencer.sv*. Kroz opis značenja i funkcije kodova biće prilozene slike koje su rezultati pokrenute simulacije.

Simulacija će se vršiti u simulatoru *SimVision*. *SimVision* je grafičko okruženje koje služi za otklanjanje grešaka (*debugging*). Može se koristiti za otklanjanje grešaka za analogne, digitalne ili mešovite signale napisane u *System Verilogu, Verilogu, C* ili *VHDL-u* [7]. Simulator omogućava da se simulira i posmatra ponašanje testbenča kao i RTL-a .RTL je dizajn, koji modeluje digitalni sklop u smislu protoka digitalnih signala između registara (hardvera) kao i logičke operacije koje se obavljaju nad tim signalima. Simulator koji će se koristiti poseduje *Design Browser*, koji prikazuje sve signale koje su definisani od strane verifikatora u kodu kao i signale definisane od strane dizajnera čipa. Signali se prikazuju u vidu opadajućeg menija. Konzola *(Console)* je prozor gde se posle izvršene simulacije ispisuju izveštaji koji su od značaja za otklanjanje grešaka ili ako zelimo da vidimo vrednost oderđene promenljive u nekom trenutku, onda se u kod ubacuje neki od verbozitija (poglavlje 2.9) koji vrši ispis promenljive od interesa. Talasni oblik *(wave)* prikazuje promene signala tokom cele simulacije. Za otklanjanje grešaka moguće je izdvojiti signale od interesa i analizirati njihovo ponašanje u određenim vremenskim trenucima, ili posmatrati promenu signala u određenom vremenskom periodu. Vremenska skala na grafičkom prikazu talasnih oblika je data u nano sekundama *(ns)*.

Master i slejv agent biće realizovani na jednom testbenču. Svako od agenata imaće svoj monitor, drajver, sekvencer, a skorbord će biti zajednički. Na skorbord će biti povezani preko *aporta* monitor slejva i drajver mastera. Na slici 3.1.3 prikazana je šema testbenča koji će se implementirati. Na jednom testbenču nalaze se dva agenta master i slejv. Svaki od agenata poseduje sopstveni drajver, sekvencer i monitor. Crnim linijama označene su žice koje povezuju komponente i preko kojih propagiraju signali. Sekvencer je komponenta koja generiše sekvencu, randomizuje sve potrabna polja i prosleđuje je preko žice drajveru. Sekvencer ima istu ulogu i u master i u slejv agentu. Drajver master agenta, kao aktivna komponenta, preko virtuelnog interfejsa ″uzima″ sekvence koje prosleđuje sekvencer i vrši obradu WDATA podatka (videti poglavlje 3.2). Drajver

slejva šalje randomizovane WDATA podatke na žicu ali oni se ne šalju u skorbord. Monitor master agenta, kao pasivna komponenta na svaku pozitivnu ivicu takta, skuplja signale sa virtuelnog interfejsa dok monitor slejv agenta pri transakciji pisanja podataka uzima podatke koje je drajver mastera poslao na žicu i kao takve ih šalje u skorbord. Skorbord je jedan za ceo testbenč. U skorbord pristžu podaci koji dolaze iz drajvera mastera i monitora slejva i tu se porede. Master i slejv agent komuniciraju preko interfejsa kojeg čini skup žica, gde u svakoj od žica ide po jedan signal (WDATA, TRANS, STRB, LEN, ADDR, ACK i REQ).
Cilj ovog merenja je da se pokaže da čip radi korektno, tj. da podaci upisani u registar jednaki su podacima pročitanim iz registra.



Slika 4.1.1 Šematski prikaz testbenča

4.1.1 REALIZACIJA INTERFEJSA

Interfejs je mehanizam za interkonekciju sa nekim operativnim sistemom ili drugim specijalizovanim softverom. Interfejs povezuje TB i DUT, tj. predstavlja skup žica koje povezuju dve hardverske jedinice. Preko interfejsa dolaze svi signali preko kojih komuniciraju master i slejv agent. U interfejsu su napisani i tvrdnje *(assertion)*. To su delovi koda koji postavljaju određene uslove koji se moraju ispoštovati jer su strogo definisani u specifikaciji. U našem slučaju to je na

primer uslov da ACK signal mora trajati jedan takt ili da TRANS, LEN i ADDR ne smeju menjati vrednosti tokom jedne transakcije. Naredni kod predstavlja implementaciju interfejsa.

```
interface example_if#(int DW=64) (input CLK,
                        input RESETn);

   logic [`DW - 1:0] WDATA;
   logic REQ;
   logic ACK;
   logic [31:0] ADDR;
   logic [3:0] LEN;
   logic [3:0] TRANS;
   bit     [7:0] STRB;

// EXAMPLE interface signals orientation from the master side
   modport MASTER (output WDATA,
                   output REQ,
                   output ADDR,
                   output STRB,
                   output TRANS,
                   output LEN,
                   input  ACK);


// EXAMPLE interface signals orientation from the slave side
   modport SLAVE  (input WDATA,
                   input REQ,
                   output ACK,
                   input TRANS,
                   input STRB,
                   input LEN,
                   input ADDR);

assert property (@(negedge CLK) disable iff (!REQ) REQ==1|=>##[1:50] (!REQ))
       else $error("t-paunovic","no ACK,CLK is longer than  50");
assert property (@(negedge CLK) disable iff (!ACK) ACK==1|=>##[1:1] (!ACK))
       else $error("t-paunovic","ACK longer then 1 CLK");
assert property(@(negedge CLK) disable iff (!REQ) REQ==1|=>$stable(TRANS))
       else $error("t-paunovic","Trans is idle!");
assert property(@(negedge CLK) disable iff (!REQ) REQ==1|=>$stable(LEN))
       else $error("t-paunovic","len is uncorrect!");
assert property(@(negedge CLK) disable iff (!REQ) REQ==1|=>$stable(ADDR))
       else $error("t-paunovic","addr is uncorrect!");


covergroup cov_if @(posedge CLK);
REQ_VALUE:coverpoint REQ {
bins req_one={1};
```

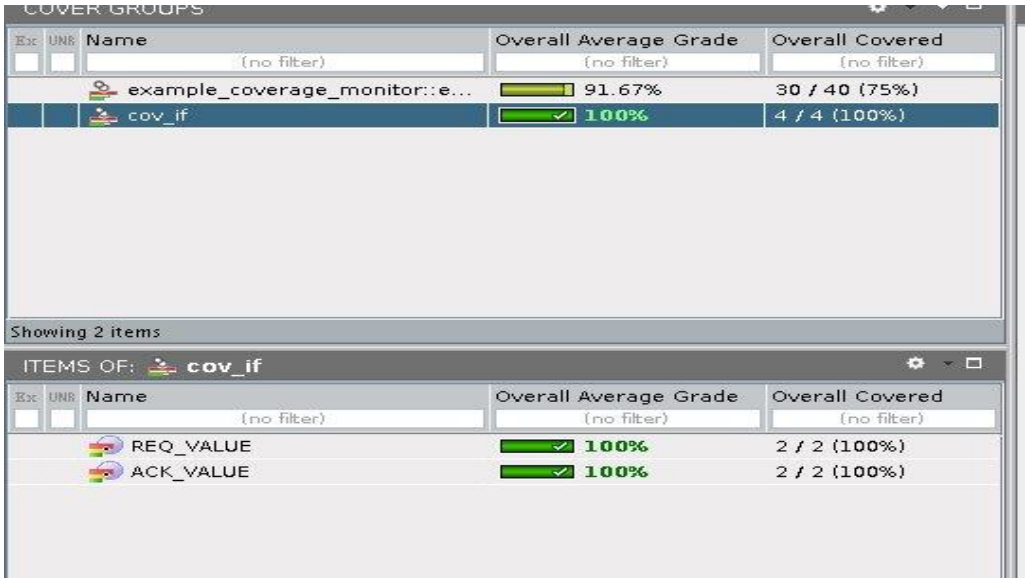
```
bins req_zero={0};
}
ACK_VALUE:coverpoint ACK{
bins low={0};
bins high={1};
}
 endgroup
cov_if cov=new();
endinterface:example_if
```

Signali CLK i RESETn dolaze iz testbenča, oni su signali ulaza *(input)* u sistem. Vrednost DW=64, predstavlja *data width* tj,veličinu polja WDATA. Prilikom definisanja signala postavljeno je da signali WDATA, REQ, TRANS, ADDR i LEN budu definisani kao *logic* tip promenljive što znači da mogu obuhvatati skup vrednosti *0, 1 , x* ili *z*. STRB signal je definisan kao bit što znači da može biti samo 0 ili 1. Svaki od ovih signala predstavlja vektor odgovarajuće dužine ili širine. Tako na primer ADDR signal je 32-bitni pa se u interfejsu predstavlja kao niz od 32 elemenata [31:0], LEN i TRANS su četvorobitni [3:0], STRB je osmobitni [7:0].

Modport definiše interfejs, tačnije signale koji dolaze sa interfejsa. Signali koji su ulaz u master agent biće izlaz *(output)* iz slejv agenta. Signali koji su izlaz iz master agenta (WDATA, REQ,TRANS,ACK,LEN) biće ulaz slejv agenta.

U interfejsu su definisani i *concurrent assertion* koji su pisani na osnovu negativne ivice takt signala. Ovim assertion izrazima obezbeđeno je da može proteći maksimum 50 taktova do prijema ACK potvrde. Ukoliko bude više, javlja se poruka o grešci: "no ACK,CLK is longer than 50". Drugim assertionom je obezbeđeno da ACK potvrda traje jedan takt, dok preostala tri assertiona uslovljavaju da signali TRANS, REQ i ADDR ne menjaju vrednosti za vreme trajanje jedne transakcije. Ukoliko se dogodi suprotno, assertion će se upaliti i javiće se poruka o grešci: "Trans is idle!" , "len is uncorrect!" ili "addr is uncorrect!".

U interfejsu su takođe napisane dve grupe pokrivenosti, naime, signali REQ i ACK mogu imati vrednosti 0 ili 1, tako da je REQ signal opisan sa *req_one=1* i *req_zero=0*. Ukoliko tokom simulacije budu ostvarene vrednosti i 0 i 1 tada će pokrivenost biti 100%. Isto važi i za ACK signal koji može imati vrednosti 0 ili 1, pokrivenost će biti 100% ako se obe vrednosti ostvare tokom simulacije. Na slici 4.1.1 je prikaz grupe pokrivenosti iz interfejsa koja je dobijena kao rezultat na kraju simulacije. Kada se na kraju simulacije u simulatoru *SimVision* otvori konzola postoji u meniju alatka koja se naziva *cover groups*. Klikom na tu alatku otvara se prozor sa definisanim grupama pokrivenosti. Pored imena svake grupe pokrivenosti prikazuje se i procentualni prikaz koliko je vrednosti pogođeno od mogućih. Vrednosti su definisane u samoj grupi pokrivenosti i zavise od promenljive koja se posmatra. Na primer, slici 4.1.1 prikazana je grupa pokrivenosti *cov_if* koja je definisana u interfejsu. U datoj grupi su dve promenljive koje se posmatraju ACK i REQ i one mogu imati vrednosti 0 ili 1. Na slici se jasno vidi da su ACK_VALUE i REQ_VALUE 100% pokrivene što znači da je svaki od signala REQ i ACK tokom simulacije imao vrednost i 1 i 0.



Slika 4.1.2 Grupa pokrivenosti sa interfejsa

4.1.2 REALIZACIJA DRAJVERA

Drajver kao aktivna UVM komponenta, realizovana je u sklopu testbenča.U okviru jednog testbenča napravljena su dva agenta master i slejv od kojih svako ima svoj drajver. U okviru *example_defines* fajla koji se nalazi u folderu *example_agent*, definisani su agenti : MASTER i SLEJV.

Example defines

```
`ifndef DW
  `define DW=64
`endif

typedef enum {MASTER,SLAVE} example_agent_ms_kind_t;
```

U gore navedenom fajlu definisana je promenljiva DW koja je jednaka dužini podataka WDATA, koja je 64 bita (8 bajta). To je pogodno, da ako se promeni dužina podatka WDATA, samo se promeni vrednost promenljive DW i svuda u kodu gde se pominje promenljiva DW, računaće se nova, izmenjena vrednost.

U okviru jedne klase *example_driver*, definisani su i drajver master agenta i drajver slejv agenta.

```
class example_driver extends uvm_driver #(example_seq_item, example_seq_item);
```

```
// UVM  Factory Registration Macro
//
`uvm_component_utils(example_driver)

// Virtual Interface
virtual example_if EXAMPLE;

uvm_analysis_port #(example_seq_item) ap;

//pointer to the config.
example_agent_config m_cfg;

//------------------------------------------
// Data Members
//------------------------------------------
byte ind;
int pr;
int br;
extern function new(string name = "example_driver", uvm_component parent = null);
extern task run_phase(uvm_phase phase);
extern function void build_phase(uvm_phase phase);

endclass: example_driver

function example_driver::new(string name = "example_driver", uvm_component parent = null);
super.new(name, parent);
endfunction

task example_driver::run_phase(uvm_phase phase);

// Master driver run_phase
if(m_cfg.master == MASTER) begin : master_run
   example_seq_item req;
   example_seq_item rsp;
   example_seq_item cloned_item;
   EXAMPLE.WDATA <= 0;
   EXAMPLE.REQ <= 0;
   EXAMPLE.TRANS <= 1;
   EXAMPLE.LEN <= 4'h3;
   EXAMPLE.STRB <=8'd0;
   @(posedge EXAMPLE.RESETn);
   @(posedge EXAMPLE.CLK);
   forever
   begin
      seq_item_port.get_next_item(req);

      @(posedge EXAMPLE.CLK);
```

```
        EXAMPLE.ADDR= req.address;
        EXAMPLE.LEN=req.len;
        EXAMPLE.STRB=req.strb;
        $display("len==========", EXAMPLE.LEN);
        EXAMPLE.TRANS=req.trans;

if (req.trans!=0) begin
    EXAMPLE.REQ<=1;
    fork
    begin
    @(posedge EXAMPLE.ACK);
    ind=0;
    end
    begin
    for(int i=0;i<50;i++)
    begin
    @(posedge EXAMPLE.CLK);
    end
    ind=1;
    end
join_any
disable fork;

 if(ind==0) begin

   fork
   begin
     if(req.trans==1)  begin

     for (int i=0;i<EXAMPLE.LEN+1;i++) begin
     @(posedge EXAMPLE.CLK);
     br=0;
     for (int l=0;l<8;l++) begin
     if (req.strb[l]==1) br++;

     end
        for (int j=0; j<8*br;j++)
            req.data[i][j]=req.address[j];
    EXAMPLE.WDATA<=req.data[i];
    $cast(cloned_item, req.clone());
    ap.write(cloned_item);

   `uvm_info(get_type_name(), $sformatf(" wdata is=%h",EXAMPLE.WDATA), UVM_LOW);
   `uvm_info(get_type_name(), $sformatf(" Adresa je=%h",EXAMPLE.ADDR), UVM_LOW);
   `uvm_info(get_type_name(), $sformatf(" STRB je=%b",EXAMPLE.STRB), UVM_LOW);
        end
        end
```

```
   else   begin
   for(int i=1; i<EXAMPLE.LEN+1; i++)
   @(posedge EXAMPLE.CLK);
   end
   end

   begin
   @(posedge EXAMPLE.CLK);
   EXAMPLE.REQ<=0;
   end
   join
   end
else
EXAMPLE.REQ<=0;

end
seq_item_port.item_done();
    end
end : master_run

// Slave driver run_phase
if(m_cfg.master == SLAVE) begin : slave_run
    example_seq_item req;
    example_seq_item rsp;
    EXAMPLE.ACK <= 0;
    @(posedge EXAMPLE.RESETn);
    @(posedge EXAMPLE.CLK);
    forever
    begin

    @(posedge EXAMPLE.REQ);
    pr=$urandom_range(8,90);
    $display("PR=%d", pr);
    if (pr<50)   begin
    for(int i=0;i<pr;i++) @(posedge EXAMPLE.CLK);
    EXAMPLE.ACK <= 1;
    @(posedge EXAMPLE.CLK);
    EXAMPLE.ACK <= 0;
    if (EXAMPLE.TRANS==2) begin
    for (int j=0;j<EXAMPLE.LEN+1;j++)  begin
    if(j!=0) @(posedge EXAMPLE.CLK);
    EXAMPLE.WDATA<=10-j;
   `uvm_info(get_type_name(), $sformatf(" wdata  of slave is=%h",EXAMPLE.WDATA),
UVM_LOW);
      end
    end
    end
 end
```

```
end : slave_run
endtask: run_phase

function void example_driver::build_phase(uvm_phase phase);
   if (!uvm_config_db #(example_agent_config)::get(this, "", "example_agent_config", m_cfg))
      `uvm_fatal("CONFIG_LOAD", "Cannot get() configuration example_agent_config from
uvm_config_db. Have you set() it?")
   ap = new("ap", this);
endfunction: build_phase
```

Drajver je napravljen kao klasa *example_driver*. Pored promenljivih koje su definisane u drajveru, definisan je i pokazivač na virtuelni interfejs *virtual example_if EXAMPLE*, jer drajver prihvata transakcije koje mu prosleđuje sekvencer. U ran fazi, ukoliko je *m_cfg.master == MASTER*, tada se kreira drajver koji je na master strani. Potrebno je istaći da uslov koji postoji u specifikaciji koji specificira da pre primljene ACK potvrde mora proteći maksimum 50 pozitivnih ivica takta, u suprotnom vrši se ponovna transmisija, i čeka se ACK potvrda. Ovaj uslov je realizovan kroz fork petlju. Uveden indikator *ind* koji dobija vrednost 1 ukoliko protekne više od 50 taktova, ukoliko ne, indikator dobija vrednost 0. Fork petlja je realizovana u dve grane koje se paralelno izvršavaju. Petlja prekida izvršavanje ukoliko se u jednoj od grana ispuni uslov. U sličaju, u prvoj grani, čeka se pozitivna ivica ACK potvrde *@(posedge EXAMPLE.ACK)*, u drugoj grani broje se pozitivne ivice takta. Ako se prva grana izvrši tada je indikator ind=0. Druga grana, u koji se broje pozitivne ivice signala takta, obustavlja se komandom *disable fork*, i počinje transakcija pisanja iz master agenta. Ukoliko se prva izvrši druga grana, znači da je prošlo 50 taktova, indikator dobija vrednost 1, i ulazi u granu *else* od uslova *if(ind==0).* U toj grani, kao što se vidi iz priloženog koda, čeka se pozitivna ivica takta i ne vrši se nikakva transakcija. Ukoliko je uslov *if(ind==0)* ispunjen, i ukoliko je ispunjen uslov *if(req.trans==1)* onda se započinje transakcija pisanja. Kada prođe uslov *if(req.trans==1),* postavlja se se petlja koja se izvrši i=LEN+1 puta, zato što u jednoj transakciji treba da se upiše LEN+1 podataka, kao što je zadato specifikacijom. U okviru ove petlje formira se podatak WDATA. Naime, prvo se izvrši brojanje jedinica u STRB signalu, promenljiva *br* čuva vrednost broja jedinica signala STRB u jednoj transakciji. Naredna petlja ide od j=0 do j= 8*br, zato što toliko bitova se upisuje u podatak *data[i][j]* koji su jednaki bitima *address[j]*. Tako generisani podaci, šalju se na virtuelni interfejs preko *ap* porta. To se realizuje komandom *ap.write(cloned_item).* Da bi u konzoli prikazali da je dati protokol ostvaren, vršimo ispis podataka *example.wdata , example.address, i example .strb.*

Ako je u pitanju slejv agent drajver je relizovan na sledeći način: uslovom if (EXAMPLE.TRANS==2) obezbeđuje se uslov da je transakcija jednaka 2, odnosno da je u pitanju transakcija čitanja podataka. Petlja se izvrši LEN+1 puta, tačno onoliko puta koliko treba da se pročita podataka koji je poslao master agent. Na virtuelni interfejs šalje se EXAMPLE.WDATA<=10-j, gde j predstavlja broj od 0 do LEN+1.

Komandom *endtask: run_phase* završena je ran faza. Naredna slika predstavlja prikaz konzole dobijene pokretanjem simulatora. Na slici 4.1.3 se vide izgenerisane vrednosti WDATA, STRB i ADDR signala prilikom jedne transakcije upisa podataka iz mastera u slejv koji šalje

drajver na virtuelni interfejs. Pošto je STRB signal sa jednom jedinicom, iz podatka ADDR će se prepisati 8*1 najnižih bitova a to je u prikazanom slučaju **bb.** Kao što se vidi sa slike 4.1.2 biti najmanje težine WDATA podatka su **bb**, koji su prepisani iz adrese.

```
_driver [example_driver]  wdata is=f3e497db0be0e0bb
_driver [example_driver]  Adresa je=e52c64bb
_driver [example_driver]  STRB je=10000000
_monitor [example_monitor] PODACI IZ SLAVE AGENTA=8d49d556bad31ebb

_driver [example_driver]  wdata is=8d49d556bad31ebb
_driver [example_driver]  Adresa je=e52c64bb
_driver [example_driver]  STRB je=10000000
_monitor [example_monitor] PODACI IZ SLAVE AGENTA=5ba79645b955a0bb

_driver [example_driver]  wdata is=5ba79645b955a0bb
_driver [example_driver]  Adresa je=e52c64bb
_driver [example_driver]  STRB je=10000000
_monitor [example_monitor] PODACI IZ SLAVE AGENTA=b9d92d29153caebb

_driver [example_driver]  wdata is=b9d92d29153caebb
_driver [example_driver]  Adresa je=e52c64bb
_driver [example_driver]  STRB je=10000000
_monitor [example_monitor] PODACI IZ SLAVE AGENTA=3b357fd3b89ee5bb
```

Slika 4.1.3 Prikaz podataka iz drajvera

4.1.3 REALIZACIJA MONITORA

Monitor je pasivna komponenta koja je instancirana u okviru testbenča. U fajlu *example_monitor* napravljeni su i monitor mastera i monitor slejva. Monitor slejv agenta povezan je na *ap* port agenta, a *ap* port agenta na skorbord, gde će slati transakcije koje je pročitao. Monitor je realizovan kao klasa *example_monitor*. Naredni kod prikazuje instanciran monitor.

```
class example_monitor extends uvm_component;

// UVM Factory Registration Macro

`uvm_component_utils(example_monitor)

// Virtual Interface

virtual example_if EXAMPLE;

//pointer to the config.

example_agent_config m_cfg;

//------------------------------------------

// Data Members

//------------------------------------------

//------------------------------------------

// Component Members

//------------------------------------------

uvm_analysis_port #(example_seq_item) ap;

endinterface: example_ifextern function new(string name = "example_monitor", uvm_component parent = null);

extern function void build_phase(uvm_phase phase);

extern task run_phase(uvm_phase phase);

extern function void report_phase(uvm_phase phase);

endclass: example_monitor


function example_monitor::new(string name = "example_monitor", uvm_component parent = null);
```

```
 super.new(name, parent);
endfunction
function void example_monitor::build_phase(uvm_phase phase);
 ap = new("ap", this);
endfunction: build_phase
task example_monitor::run_phase(uvm_phase phase);
    example_seq_item item;
    example_seq_item cloned_item;
    item = example_seq_item::type_id::create("item");
    @(posedge EXAMPLE.RESETn);
    @(posedge EXAMPLE.CLK);
    forever begin
        // Detect the protocol event on the TBAI virtual interface
        if (m_cfg.master == MASTER) begin
        if (EXAMPLE.TRANS==1)begin
        @(posedge EXAMPLE.REQ);
         for (int i=0;i<item.len+1;i++) begin
         item.trans=EXAMPLE.TRANS;
         item.address=EXAMPLE.ADDR;
         item.strb=EXAMPLE.STRB;
         item.len=EXAMPLE.LEN;
         item.data[i] = EXAMPLE.WDATA;
         $cast(cloned_item, item.clone());
         $display($time," upisujem ap1");
         ap.write(cloned_item);
         end
```

```
        end
        end
  else @(posedge EXAMPLE.ACK)
  @(negedge EXAMPLE.CLK);
   begin
        item.trans=EXAMPLE.TRANS;
        item.address=EXAMPLE.ADDR;
        item.len=EXAMPLE.LEN;

 IDLE: assert (item.trans!=3)
 else `uvm_error("t-paunovic","Master not allowed,trans is 3");

 if (EXAMPLE.ACK==1)
 REQ_ACK:assert (EXAMPLE.REQ==1)
 else `uvm_error("t-paunovic","REQ is not 1");

   // Clone and publish the cloned item to the subscribers

 if (m_cfg.master == SLAVE)
 if(EXAMPLE.TRANS==1)  begin
 for (int i=0;i<item.len+1;i++)   begin
 @(negedge EXAMPLE.CLK)
 item.data[i]=EXAMPLE.WDATA;
 `uvm_info(get_type_name(), $sformatf("PODACI IZ SLAVE AGENTA=%h",item.data[i]),
UVM_LOW);
        end
```

```
$cast(cloned_item, item.clone());
$display($time," upisujem ap2");
ap.write(cloned_item);
$display("ap napunjen");
end
end
end
endtask: run_phase
function void example_monitor::report_phase(uvm_phase phase);
// Might be a good place to do some reporting on no of analysis transactions sent et
endfunction: report_phase
```

Monitor se definiše rezervisanom reči ***class*** i imenom koje je u ovom slučaju *example_monitor*. Potrebno je ovako definisanu UVM komponentu registrovati u fabrici, što postžemo komandom *`uvm_component_utils(example_monitor),* zatim definišemo virtuelni interfejs kao i analysis port *ap,* koji povezuje agent i monitor komponentu. Celokupna klasa sadrži tri faze: *build phase*, *run phase*, *report phase*. U build fazi vrši se alokacija memorije koja predstavlja objekat *ap* porta koji je označen kao *ap = new("ap", this).*

U run fazi master i slejv agent skupljaju transakcije sa virtuelnog interfejsa. Ako su ispunjeni uslovi *if (m_cfg.master == MASTER)* i *if (EXAMPLE.TRANS==1)* monitor master agenta skuplja transakciju sa interfejsa pri TRANS=1, što je po specifikaciji transakcija pisanja podataka. Definisan je i objekat *item*, koji je pokazivač na klasu *sequence_item,* gde se smeštaju verdnosti signala trenutne transakcije kao što su WDATA, LEN, TARANS, ADDR, STRB. Tim poljima pristupamo sa dodavanjem imena objekta u našem slučaju *item.ime_signala*. Linijom koda *item.data[i] = EXAMPLE.WDATA* u polje item prosleđujemo podatke primljene sa interfejsa. Tako upisujemo i podatke ADDR, LEN, TRANS, STRB. Funkcijom *ap.write(cloned_item)* tako pročitani podaci šalju se na *ap* port. Ukoliko nije ostvaren uslov *if (EXAMPLE.TRANS==1)* čeka se pozitivna ivica takta *@(posedge EXAMPLE.ACK).*

Uslovom *if (m_cfg.master == SLAVE)* kao i *(EXAMPLE.TRANS==1)* obezbeđeno je da slejv agent skuplja podatke sa interfejs žice pri transakciji *″read″* tj.čitanja. U *item.data[i]=EXAMPLE.WDATA* ″pakuju″ se podaci koji dolaze sa virtuelnog interfejsa i funkcijom *ap.write(cloned_item)* dobijeni podaci se šalju preko *ap* porta u skorbord.

Naredna funkcija napisana je u okviru *example_agent_config* fajla. U okviru ovog fajla istaćićemo samo konekt *(connect)* fazu.

```
function void example_agent::connect_phase(uvm_phase phase);
```

```
    m_monitor.EXAMPLE = m_cfg.EXAMPLE;
    // m_monitor.example_index = m_cfg.example_index;
    if(m_cfg.master == MASTER) ap = m_driver.ap;
    else ap = m_monitor.ap;
    // Only connect the driver and the sequencer if active
    if(m_cfg.active == UVM_ACTIVE) begin
       m_driver.seq_item_port.connect(m_sequencer.seq_item_export);
       m_driver.EXAMPLE = m_cfg.EXAMPLE;
       m_driver.m_cfg = m_cfg;
       m_monitor.m_cfg = m_cfg;
    end
    if(m_cfg.has_functional_coverage) begin
       m_monitor.ap.connect(m_fcov_monitor.analysis_export);
       m_fcov_monitor.m_monitor = m_monitor;
    end
endfunction: connect_phase
```

U konekt fazi priložene funkcije spaja se drajver mastera i monitor slejv agenta sa *ap* portom agenta koji se posle povezuje na *ap* port skorborda, gde će se porediti podaci koje je poslao master i koje je slejv pročitao. Deo koda koji povezuje date portove je:

```
    if(m_cfg.master == MASTER) ap = m_driver.ap;
    else ap = m_monitor.ap;
```

Zatim se vrši povezivanje sekvencera i drajvera, export sekvencera se povezuje na port drajvera *m_driver.seq_item_port.connect(m_sequencer.seq_item_export),* monitor i drajver su povezani na virtuelni interfejs, odakle se vrši slanje i preuzimanje podataka. Monitor pokrivenosti je povezan sa monitorom *m_monitor.ap.connect(m_fcov_monitor.analysis_export).*

U monitoru su napisana dva assertiona, prvi ne dozvoljava da transakcija bude 3, što je po specifikaciji IDLE mod rada, u suprotnom prijaviće se greška "t-paunovic","Master not allowed,trans is 3". Drugi, uslovljava da kada signal ACK bude 1, REQ signal mora biti 1, jer je nemoguće u suprotnom primiti ACK potvrdu. Ako ovaj uslov nije zadovoljen javlja se poruka o grešci "t-paunovic","REQ is not 1".

4.1.4 REALIZACIJA MONITORA POKRIVENOSTI

U monitoru pokrivenosti kao pasivnoj UVM komponenti instanciranoj u okviru master i slejv agenta definisane su i napravljene grupe pokrivenosti. To su grupe koje pravi verifikator, da bi proverio pokrivenost promenljive od interesa. Pošto svi definisani signali mogu imati više vrednosti u okviru jedne transakcije kao i random vrednosti, od interesa je proveriti koliko se puta i da li se uopšte javljaju očekivane vrednosti. Zbog toga se kreiraju grupe pokrivenosti. Naredni kod predstavlja implementiran monitor pokrivenosti koji je povezan sa monitorom od koga ″uzima″ vrednosti signala tokom transakcije.

```
class example_coverage_monitor extends uvm_subscriber #(example_seq_item);
// UVM Factory Registration Macro

`uvm_component_utils(example_coverage_monitor)

// monitor handler

example_monitor   m_monitor;

//----------------------------------------

// Cover Group(s)

//----------------------------------------

 covergroup example_cov_data;

 TRANS_VALUE:coverpoint analysis_txn.trans{

      bins one={1};

      bins two={2};

}

LEN_VALUE:coverpoint analysis_txn.len{

            bins one={[0:3]};

            bins two={[4:7]};

            bins three={[8:10]};

            bins four={[11:13]};

            bins five={[14:15]};

}

ADDR_VALUE:coverpoint analysis_txn.address{

bins address_one={[32'h0:32'h509be39f]};

bins address_two={[32'h509be3af:32'hAAAAAAAA]};

bins address_three={[32'hAAAAAAAB:32'hFFFFFFFF]};

}

CMP: cross ADDR_VALUE,LEN_VALUE,TRANS_VALUE;
```

```
endgroup
//------------------------------------------
// Component Members
//------------------------------------------
example_seq_item analysis_txn;
//------------------------------------------
// Methods
//------------------------------------------
// Standard UVM Methods:
extern function new(string name = "example_coverage_monitor", uvm_component parent = null);
extern function void write(T t);
extern function void report_phase(uvm_phase phase);
endclass: example_coverage_monitor

function example_coverage_monitor::new(string name = "example_coverage_monitor", uvm_component parent = null);
   super.new(name, parent);
   example_cov_data = new();
endfunction
function void example_coverage_monitor::write(example_seq_item t);
   analysis_txn = t;
   example_cov_data.sample();
endfunction:write
function void example_coverage_monitor::report_phase(uvm_phase phase);
// Might be a good place to do some reporting on no of analysis transactions sent etc
endfunction: report_phase
```

Kao i svaka komponenta UVM-a, bilo da je aktivna ili pasivana, monitor pokrivenosti će se realizovati kao klasa *example_coverage_monitor* koja je proširenje klase *uvm_subscriber* i kojoj se prosleđuju transakcije formirane u *example_seq_item-u.* Posle definicije klase, komponentu monitor pokrivenosti moramo registrovati u fabrici *`uvm_component_utils(example_coverage_monitor).*

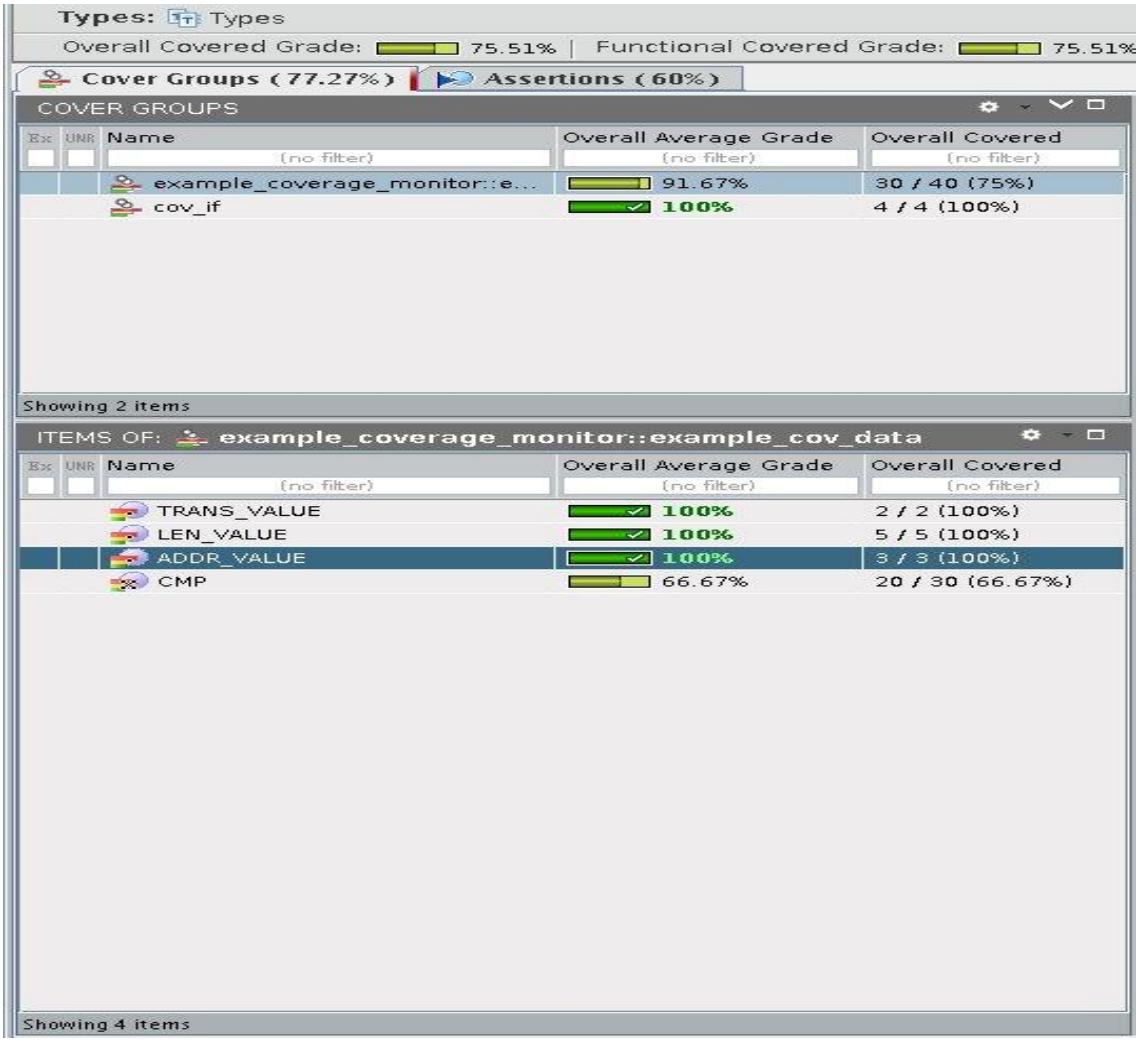
Grupa pokrivenosti koju kreiramo zove se *example_cov_data*, proverićemo pokrivenost TRANS, ADDR i LEN signala. Svaki od navedenih signala predstavlja coverpoint. Svaki *coverpoint* je pokazivač na polje signala koji čuva trenutnu vrednost signala u toku transakcije. *Coverpointi* su: TRANS_VALUE, LEN_VALUE i ADDR_VALUE. U okviru svakog *coverpointa* je rezervisana reč *bins* koja obuhvata skup mogućih vrednosti koje ta promenljiva može imati. Tako da su i za LEN i za TRANS i za ADDR napravljeni bins skupovi sa vrednostima koji dati signal može imati. Tako na primer, *coverpoint* TRANS_VALUE može imati samo dve vrednost u toku transakcije 1 ili 2, 3 je nedozvoljena. Stoga, kreirana su dva binsa koji uzimaju vrednosti 1 ili 2. Isto je urađeno za naredna dva signala. Komandom *example_cov_data = new()* vrši se alokacija memorije i smeštaju se rezultati grupe pokrivenosti. Linija koda *example_cov_data.sample()* uzima odbirke datih signala dobijenih u transakciji.

Ako postoji više signala čije se vrednosti prate za analizu pokrivenosti, uglavnom, ali ne i obavezno može se uraditi i presek dobijenih vrednosti različitih signala komandom *cross.* U datom primeru to je postignuto sa CMP: cross ADDR_VALUE, LEN_VALUE, TRANS_VALUE. Ovim se ispituju sve mogiće kombinacije vrednosti signala ADDR, LEN i TRANS.

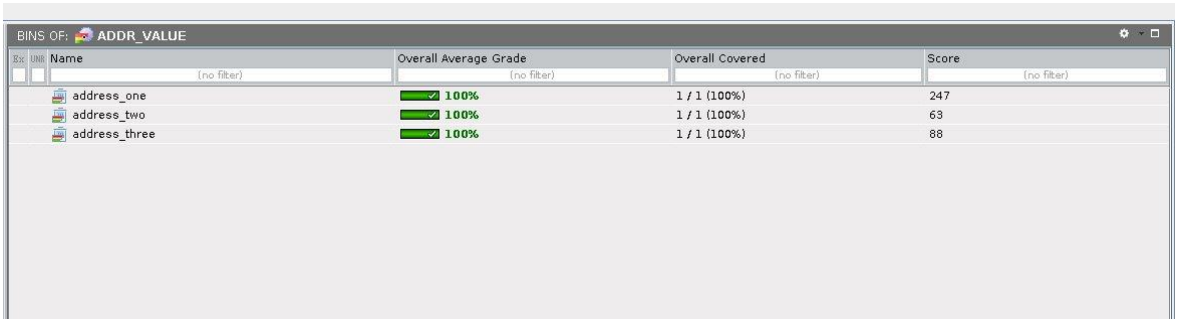
U report fazi mogu se dodavati uvm izveštaji koji su od značaja za analizu pokrivenosti .

Cilj svake analize pokrivenosti je da pokrivenost bude 100% odnosno da u toku simulacije budu prisutne sve moguće vrednosti za signal od interesa. Što je pokrivenost veća, to je verifikator analizirao bolje dati čip. Na narednim slikama dati su rezultati analize pokrivenosti čipa kojeg verifikujemo, posle jedne simulacije. Kao što se vidi pokrivenost kreirane grupe pokrivenosti je 100%, što znači da se za svaki bin koji predstavlja skup vrednosti za određeni signal u toku simulacije je ostvarena bar jedna vrednost.

Na slici 4.1.4 prikazana je grupa pokrivenosti dobijena otvaranjem konzole po završetku simulacije. Simulator *SimVision* poseduje dugme *cover groups* čijim se klikom otvara prozor sa grafičkim prikazom grupa pokrivenosti. Prva kolona je ime *coverpointa*. U našem slučaju to su: ADDR_VALUE, LEN_VALUE, TRANS_VALUE. Druga kolona je procentualni prikaz rezultata analize pokrivenosti. Za sva tri *coverpointa* je to 100%. Za coverpoint CMP, koji predstavlja kombinacije svih mogućih vrednosti ADDR, LEN i TRANS signala postignuta pokrivenost je 66,67%. To znači da nisu ostvarene sve kombinacije. Zadnja kolona je odnos mogućih verdnost i postignutih vrednosti tokom simulacije. Tako na primer 5/5 znači da je ostvareno svih pet vrednosti LEN signala, tokom simulacije od mogućih pet.



Slika 4.1.4 Prikaz cover grupe iz coverage monitora posle završene simulacije



Slika 4.1.5 Grafički prikaz binsa ADDR_VALUE coverpointa

Sa slike 4.1.5 jasno se vidi da su svi binsi *coverpointa* ADDR_VALUE jednaki 100%, sto znači da je u svakom od binsa definisanih sa:

bins address_one={[32'h0:32'h509be39f]};

bins address_two={[32'h509be3af:32'hAAAAAAAA]};

bins address_three={[32'hAAAAAAAB:32'hFFFFFFFF]};

u toku simulacije ostvarila bar jedna vrednost. Isto važi i za binse *coverpointa* LEN_VALUE i TRANS_VALUE.

4.1.5 REALIZACIJA SKORBORDA

Skorbord spada u pasivne UVM komponente koje su neophodne pri svakom procesu verifikacije. Ponekad se u skorbordima mogu pisati i grupe pokrivenosti, mada se to uglavnom kreira u već pomenutom monitoru pokrivenosti. Skorbord je jedna od najsloženijih UVM komponenti zato se u skorbordu kreiraju funkcije koje porede podatke, predviđaju podatke, izbacuju poruke o greškama (ukoliko do njih dođe), čuvaju vrednosti podataka i daju informaciju o trenutnim vrednostima signala u transakciji. Skorbord se kao i sve do sada navedene komponente implementira kao klasa koja je proširenje klase *uvm_scoreboard.* Naredni kod predstavlja klasu skorborda *u2u_scoreboard,* koji je deo testbenča čipa koji verifikujemo.

```
`uvm_analysis_imp_decl(_example_master)
`uvm_analysis_imp_decl(_example_slave)

class u2u_scoreboard extends uvm_scoreboard;
 `uvm_component_utils(u2u_scoreboard)

 example_seq_item data_to_check[$];

 uvm_analysis_imp_example_master #(example_seq_item,u2u_scoreboard) example_add;
 uvm_analysis_imp_example_slave #(example_seq_item,u2u_scoreboard) example_match;

 function new (string name = "", uvm_component parent = null);
 super.new(name, parent);
 example_add  = new("example_add", this);
 example_match = new("example_match", this);
 endfunction : new

 // implement example_add analysis port from reference model
 virtual function void write_example_master(example_seq_item packet);
 data_to_check.push_back(packet);
 $display("puni red!!!");
 endfunction : write_example_master


   virtual function void write_example_slave(example_seq_item packet);
      example_seq_item temp_packet;
      $display("usao u write");
```

```
      temp_packet = data_to_check.pop_back();

 for(int i=0;i<packet.len+1;i++) begin
    U2U_SCB_CHECK_1: assert(temp_packet.data[i] == packet.data[i]) else begin
    `uvm_error("U2U_SCB_CHECK_1:","Test Failed - Slave side did not receive the expected packet")
 end

 `uvm_info(get_type_name(), $sformatf(" packet data=%h",packet.data[i]), UVM_LOW);
 `uvm_info(get_type_name(), $sformatf(" temp packet data=%h",temp_packet.data[i]), UVM_LOW);

  end

U2U_SCB_CHECK_2:assert(temp_packet.len == packet.len) else begin
 `uvm_error("U2U_SCB_CHECK_2:","Test Failed - Slave side did not receive the expected packet")
end

U2U_SCB_CHECK_4:assert(temp_packet.trans == packet.trans) else begin
`uvm_error("U2U_SCB_CHECK_4:","Test Failed - Slave side did not receive the expected packet")
end

U2U_SCB_CHECK_5:assert(temp_packet.address == packet.address) else begin
`uvm_error("U2U_SCB_CHECK_5:","Test Failed - Slave side did not receive the expected packet")
end

endfunction : write_example_slave

 function void report_phase(uvm_phase phase);
 if(data_to_check.size() ==0) begin
 `uvm_info("U2U_SCB_REPORT:","Test Passed - Scoreboard empty after testing", UVM_LOW)
 end

U2U_SCB_CHECK_2: assert (data_to_check.size()!=0)  begin
 `uvm_error("U2U_SCB_CHECK_2:","Test Failed - Scoreboard is not empty after testing")
 `uvm_info(get_type_name(), $sformatf(" velicina reda je=%d",data_to_check.size()), UVM_LOW);
 end
endfunction: report_phase

endclass : u2u_scoreboard
```

Posle definicije klase i registrovanja u fabrici, definišemo red *data_to_check*, to je niz koji nema fiksan broj članova a tip podatka je *example_seq_item*. Da bi drajver mastera i monitor slejva uspešno slali svoje transakcije potrebno je definisati portove.

```
uvm_analysis_imp_example_master #(example_seq_item,u2u_scoreboard) example_add
uvm_analysis_imp_example_slave #(example_seq_item,u2u_scoreboard) example_match
```

Potrebno je istaći funkciju *write_example_master* kojoj prosleđujemo *seq_item* koji je tipa *packet.* U već definisani *data_to_check,* pakuju se transakcije koje stižu do porta skorborda iz drajvera mastera *data_to_check.push_back (packet).* U funkciju *write_example_slave,* u promenljivu *temp_packet* čitaju se podaci koji su smešteni u red *temp_packet = data_to_check.pop_back().*

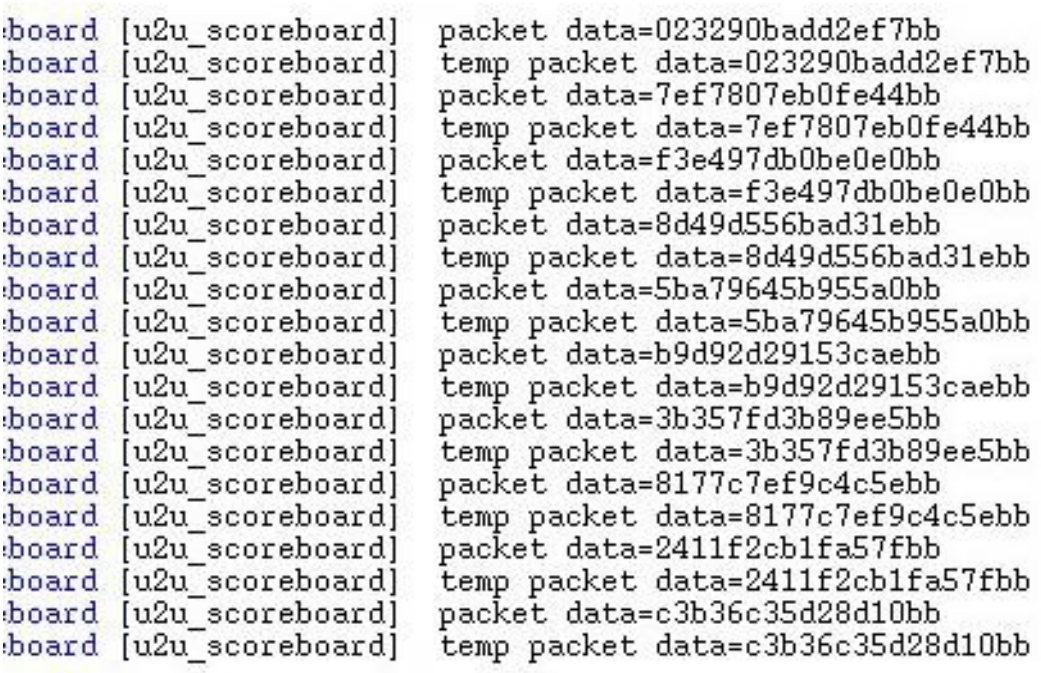
Sada se u skorbordu pišu čekeri koji porede pristigle podatke iz mastera i slejva. Formira se petlja koja se vrti LEN+1 puta jer se očekuje toliko pristiglih podataka u toku transakcije. Prvo se porede dobijeni podaci pristigli iz mastera i slejva uslovom*: temp_packet.data[i] == packet.data[i]*, ukoliko su oni jednaki na kraju simulacije u konzoli će se prikazati njihove vrednosti, ukoliko nisu izaći će poruka o grešci: ″U2U_SCB_CHECK_1:","Test Failed - Slave side did not receive the expected packet". Vrši se identična provera i za vrednosti LEN, ADDR i TRANS. Ukoliko neka od primljenih vrednosti nije ista (ona koju pošalje master i slejv) tada se prikazuje poruka: *"Test Failed - Slave side did not receive the expected packet."*

Svaki skorbord na kraju poređenja podataka treba da ima prazan red u kojem su bile smeštene trenutne vrednosti transakcije. To se postiže uslovom *if(data_to_check.size() ==0)* ukoliko je ovaj uslov ispunjen, tj. ukoliko je veličina reda 0 javlja se poruka "U2U_SCB_REPORT:",*"Test Passed - Scoreboard empty after testing"*. Odnosno ako je zadovoljen uslov: *data_to_check.size()!=0*, javlja se poruka "U2U_SCB_CHECK_2:", "*Test Failed - Scoreboard is not empty after testing"*.

U praksi se može desiti da veličina reda ne bude nula nakon testiranja, to se dešava zbog kašnjenja u sistemu, pa se ponekad zanemari ako veličina reda nije mnogo veća od nule. Međutim ako je veličina reda posle testiranja velika, tj. ako ostane izvestan broj podataka koji nisu upoređeni sa očekivanim, znači da je više podataka predviđeno i greška se mora otkloniti.

Sa slike 4.1.6 vidi se izveštaj iz konzole dobijen na kraju simulacije. Jasno se vidi da su paketi pristigli iz mastera *(packet data*) i slejva *(temp packet data)* identični, kao i to da je test prošao, tj. da je skorbord prazan posle testiranja (slika 4.1.7).

```
sao u write
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=023290badd2ef7bb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=023290badd2ef7bb
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=7ef7807eb0fe44bb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=7ef7807eb0fe44bb
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=f3e497db0be0e0bb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=f3e497db0be0e0bb
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=8d49d556bad31ebb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=8d49d556bad31ebb
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=5ba79645b955a0bb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=5ba79645b955a0bb
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=b9d92d29153caebb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=b9d92d29153caebb
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=3b357fd3b89ee5bb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=3b357fd3b89ee5bb
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=8177c7ef9c4c5ebb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=8177c7ef9c4c5ebb
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=2411f2cb1fa57fbb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=2411f2cb1fa57fbb
UVM_INFO ../env/u2u_scoreboard.sv(87) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  packet data=c3b36c35d28d10bb
UVM_INFO ../env/u2u_scoreboard.sv(88) @ 88770: uvm_test_top.m_env.m_scoreboard [u2u_scoreboard]  temp packet data=c3b36c35d28d10bb
o napunjen
```



Slika 4.1.6 Podaci pristigli u scbd iz drajvera mastera (packet data) i iz monitora slejva(temp packet data) koji su jednaki

```
UVM_INFO
/opt/eda/cadence/incisiv12.20.021/tools/uvm/uvm_lib/uvm_sv/sv/base/uvm_objection.svh(1268) @
102985: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase

UVM_INFO ../env/u2u_scoreboard.sv(106) @ 102985: uvm_test_top.m_env.m_scoreboard
[U2U_SCB_REPORT:] Test Passed - Scoreboard empty after testing
```

Slika 4.1.7 Izveštaj iz konzole da je test prošao tj. da je skorbord prazan

4.1.6 REALIZACIJA SEQUENCE_ITEM

Sequence item je fajl u kome se definisani sva polja sekvence, kao i moguće funkcije koje se sprovode nad sekvencama kao što su *do_copy, do_compare, convert2string*. U posmatranom kodu nad sekvencom koja će ići u drajver sprovešćemo samo funkciju *do_copy*. Klasa koja opisuje *sequence_item* zove se *example_sequence_item*. Naredni kod prikazuje *sequence_item*.

```
`define uvm_record_field (NAME,VALUE);

$add_attribute(recorder.tr_hendle,VALUE,NAME);

class example_seq_item extends uvm_sequence_item;

`uvm_objects_utils(example_seq_item);

//data members

rand bit [‘DW-1:0] data[16];

rand logic[1:0] trans;

rand logic[3:0] len;

rand logic[31:0]address;

rand bit [7:0] strb;

rand int delay;

constraint transfer {trans<3};

constraint strb_signal{strb inside {8’b00001111,8’b11100000,8’b11000000,8’b10000000};}

//standard UVM metods

extern function new(stringname=”example_seq_item);

extern function do_copy(uvm_objects rhs);

extern bit do_compare(uvm_objects rhs,uvm_comparer,comparer);

extern string convert2string();

void do_print(uvm_printer,printer);

endclass:example_seq_item;

function example_seq_item::new(string_name=″example_seq_item″);

  super.new(new);
```

```
endfunction
function void example_seq_item::do_copy(uvm_object rhs)
        example_seq_item rhs_;
  if (!cast(rhs_,rhs)) begin
`uvm_fatal("do-copy", "cast of rhs object failed");
end
super.do_copy(rhs);
for (int i=0; i<16; i++) begin
data[i]=rhs_.data[i];
end
data=rhs_.data;
trans=rhs_.trans;
len=rhs_.len;
address=rhs_.address;
streb=rhs_.strb;
endfunction:do_copy;
function bit example_seq_item::do_comparer(uvm_object rhs,uvm_comparer comparer);
example_seq_item rhs_;
  if (!cast(rhs_,rhs)) begin
`uvm_fatal("do-comparer", "cast of rhs object failed");
return 0;
end
return super.do_compare(rhs,comparer) && data=rhs_.data;
endfunction: do_copare;
function string example_seq_item::convert2string();
string s;
```

```
$ sformat (s,%s/n super.convert2string());
return s;
endfunction convret2string;
function void example_seq_item::do_print(uvm_printer);
if (printer.knobsprint==0) begin
$display(convert2string());
end
else begin
printer.m_sting=convert2string();
end
endfunction:do_print
```

Posle definicije klase, potrebno je tu klasu registrovati u fabrici *`uvm_objects_utils (example_seq_item)*. Zatim sledi definicija članova podataka koju će sadržati transakcija kao što su LEN, TRANS, STRB, ADDR itd. Svi podaci su definisani kao random vrednosti. WDATA je definisan kao višedimenzioni niz od kojih svaki od njih ima dužinu 64, a ima ukupno 16 takvih nizova. U toku jedne transakcije se prenosi LEN+1 podatak, a maksimalna vrednost LEN je 15. Posle definisanja podataka uveli smo 2 ograničenja, prvo je da TRANS može imati vrednost 1 ili 2, a drugi predstavlja skup mogućih vrednosti STRB signala. Pri generisanju vrednosti u svakoj transakciji uzima se vrednost isključivo iz skupa definisanog u ograničenju. U ovom fajlu definisane su 3 funkcije *do_copy*, *do_compare*, *convert2string* od kojih će od značaja biti samo *do_copy*. Funkcija *do_copy* kopira trenutne vrednosti transakcije u cilju njihove dalje obrade.

Prilikom kompajliranja radi puštanja simulacije i analize dobijenih rezultata, potrebno je imati i fajlove koji se nazivaju pkg *(package)* a sadrže grupu pojedinih fajlova koji se kompajliraju. U datom sličaju to je *example_agent_pkg*.

```
package example_agent_pkg;
import uvm_pkg::*;
`include ″uvm_macros.svh″
`include“example_defines_sv“
`include “example_agent_config.sv”
```

`include “example_seq_item”

`include “example_driver”

`include ”example_monitor”

`include ”example_coverage_monitor”

`include ’example_sequencer”

`include ”example_agent”

endpackage: example_agent_pkg

Kao što se vidi ovaj fajl u sebi sadrži spisak fajlova koje čine agent. Ovi fajlovi se čitaju, traže po imenu klase, kada se pronađu kreće se u proces kompajliranja (proces prevođenja iz izvornog u mašinski jezik). Ako je program sintaksno ispravno napisan, kompajler ne prijavljuje nikakve greške i tada počinje simulacija.
Treba istaći da je učestanost takta 33 MHz. To se postiže uvođenjem petlje

initial begin

CLK=0;

RESETn=0;

repeat(8) begin

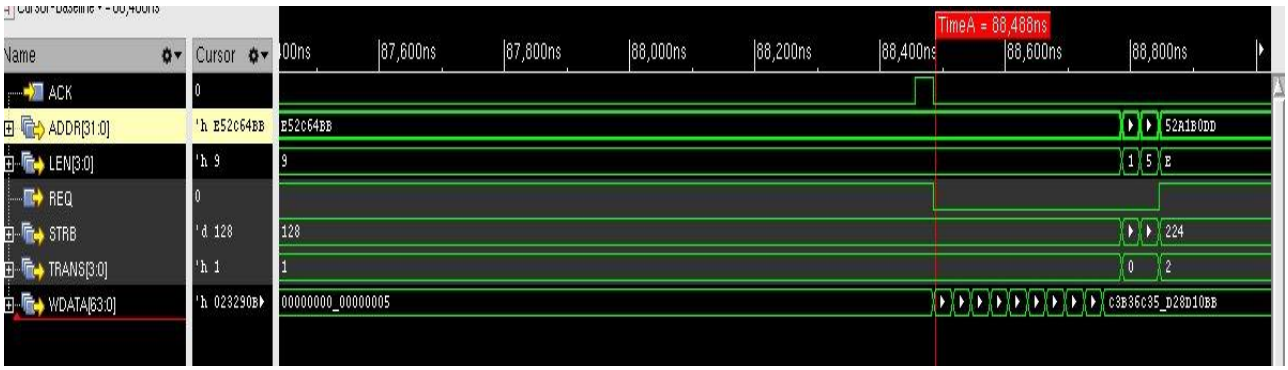
for (int i=0; i<15; i++) #1ns;

CLK= ~CLK;

end

Na svakih 15 ns menja se polaritet takta, period signala je 30ns, stoga, učestanost signala je 33MHz.

Posle kompajliranja fajlova, simulator pokreće grafičko okruženje, gde se prikazuje talasni oblik sa svim signalima definisanih u interfejsu (slika 4.1.8). Na slici su prikazani signali sa interfejsa ACK, ADDR, LEN, REQ, STRB, TRANS i WDATA. Adresa je u jednoj transakciji e52c64bb, LEN je 9, što znači da će u jednoj transakciji biti upisano 10 WDATA podataka. REQ se spušta na 0 u trenutku kada i ACK potvrda pada na 0. Tada je master primio potvrdu da može da započne transakciju pisanja podataka. TRANS signal kao što se vidi sa slike je jednak jedinici, što znači da je u pitanju transakcija pisanja. STRB signal je 128 u decimalnom zapisu, što u binarnom iznosi 10000000. Ta vrednost spada u grupu vrednosti koje može uzeti signal, a koji su definisani ograničenjem *strb_signal* u *example_seq_item* klasi. U trenutku koji je obeležen na slici 88,488ns počinje transakcija pisanja, jer je primljena ACK potvrda i REQ signal je spušten na 0. Od tog trenutka se pišu WDATA podaci i vidi se da ih ima ukupno 10 (jer je LEN 9). Vidi se vrednost zadnjeg upisanog WDATA podatka koje je u heksadecimalnom zapisu c3b36c35_d28d10bb.



Slika 4.1.8 Grafičko okruženje,prikaz simulacije

5.ZAKLJUČAK

Proces verifikacije je vrlo odgovoran segment projektovanja, koji se odnosi na dokazivanje korektnosti rada čipa.Verifikacija može ukazati na greške, ali ne može garantovati njihovo odsustvo. Zato je neophodno da se razume princip rada čipa, jer pogrešno shvatanje specifikacije može ukazati na nepostojeće greške u implementaciji čipa, tako da to može izazvati ozbiljne komplikacije pri projektovanju. Prilikom svakog procesa verifikacije neophodno je analizirati talasni oblik signala sa svim signalima od interesa. Bagovi (greške u dizajnu) mogu se videti iz posmatranja ponašanja signala na talasnom obliku, koje nije isto kao očekivano. Tada verifikator ukazuje na potencijalne grške.

Simulacija u datom primeru traje 102,985 ns. To je dovoljno za simulaciju realnog rada čipa. Kroz datu simulaciju, signali su pokazali da je ponašanje u skladu sa specifikacijom tj. da su ispoštovana sva pravila ponašanja signala data u specifikaciji. U toku simulacije nije uključen ni jedan assertion, što znači da su uslovi napisani u tvrdnjama *(assertion)* ispunjeni. Ni jedno ograničenje *(constraint)* nije premošteno, u protivnom javila bi se poruka o grešci, tj. svi signali su uzimali vrednosti iz dozvoljenog skupa. Dobijeni su očekivani rezultati merenja, podaci koje pošalje master agent preko magistrale, jednaki su podacima koje slejv agent pročita. Odnosno, podaci upisani u registar jednaki su podacima pročitanim iz registra.

Troškovi proizvodnje ASIC čipova su veliki, pa je ekonomska isplativost prisutna samo ako se proizvode u velikim serijama. Proizvođači čipova danas nude veliki broj ASIC-a, koji obavljaju kompleksne funkcije, kao što su transformacija koordinata, PWM modulacija.

Ovaj rad se može iskoristiti za upoznavanje sa procesom verifikacije. Dati promer čipa sa korisnički definisanim komunikacionim protokolom se može lako proširiti i njegovi delovi koristiti za verifikaciju čipova koji implementiraju i druge funkcije. Korisnički protokol koji je korišćen u ovoj tezi, bazira se na AMBA *(Advance Microcontroller Bus Architecture)* AHB (*Advance High Performance Bus)* Lite protokolu [8].

LITERATURA

[1] *A practical guide to adopting the Universal Verification Methodology (UVM)* [Online] Available: http://www.scribd.com/doc/131393841/A-Practical-Guide-to-Adopting-the-Universal-Verification-Methodology-UVM

[2] Janick Bergeron,*"Writing testbench usig System Verilog"*, Springer 2006

[3] Spear Chris, *"System verilog for verification"*, Springer 2006

[4] Srikanth Vijayaraghavan, *"A practical guide for System Verilog assertions"*, Springer 2006

[5] https://verificationacademy.com

[6] http://www.asic-world.com/specman/tutorial.html

[7] http://www.cadence.com/products/fv/simvision/pages/default.aspx

[8] https://web.eecs.umich.edu/~prabal/teaching/eecs373-f11/readings/ARM_IHI0033A_AMBA_AHB-Lite_SPEC.pdf