Funkcionalna verifikacija hardvera

Vežba 5
Uvod u UVM metodologiju razvoja verifikacionih okruženja

Sadržaj

Ova i naredne vežbe su posvećene univerzalnoj verifikacionoj metodologiji. U ovoj vežbi je dat kratak pregled metodologije i objašnjeni osnovni koncepti. Data je funkcionalna specifikacija dizajna koji će se verifikovati i opis transakcija. Naredne vežbe su posvećene pojedinim komponentama, njihovoj ulozi i načinu korišćenja.

1 Uvod

UVM (engl. *Universal Verification Methodology*) je standardizovana metodologija za funkcionalnu verifikaciju sa pomoćnom bibiliotekom u SystemVerilog jeziku. Kreirala ju je firma Accellera u saradnji sa mnogim EDA prodavcima i klijentima sa željom da se postigne lakša ponovna upotreba testbenčeva i jednostavno kreiranje univerzalnih, visoko-kvalitetnih VIP-a (engl. *Verification Intellectual Property*). UVM je baziran na OVM-u (engl. *Open Verification Methodology*) i eRM-u (engl. *e Resuse Methodology*).

Jedna od glavnih karakteristika ove metodologije je UVC (engl. *Universal Verification Component*), odnosno univerzalne verifikacione komponente koje imaju istu strukturu (sadrže monitore, drajvere, sekvencere, ...) što omogućava lako korišćenje bilo kako nezavisne komponente ili kao deo većeg sistema.

Objektno-orijentisani dizajn, kao glavna karakteristika SystemVerilog jezika, omogućava lako kreiranje verifikacionih komponenti, dok veliki broj predefinisanih funkcija i taskova znatno ubrzava proces kreiranja testova.

UVM obezbeđuje *framework* za verifikaciju zasnovanu na pokrivenosti (engl. *Coverage Driven Verification*, skraćeno CDV) koja ne zahteva kreiranje velikog broja testova, osigurava temeljnu verifikaciju na osnovu zadatih parametara i olakšava proces pronalaženja problema. Ovo se postiže korišćenjem *self-checking* testbenčeva, automatskog generisanja testova i korišćenjem podataka o pokrivenosti.

1.1 Istorijat

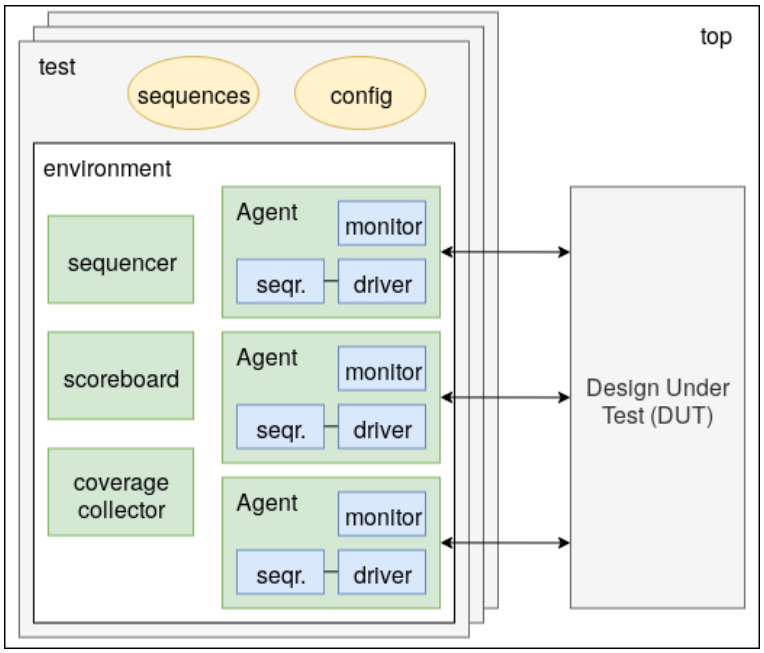
UVM 1.0 je nastao 2011 godine i, zbog velike podrške simulatora, ubrzo zamenio, do tada popularni, OVM. UVM 1.0 je preuzeo dosta iz OVM 2.1.1 verzije (u velikom delu koda samo zamenjujući prefikse "ovm_" sa "uvm_") i dodao par nadogradnji koje umnogome olakšavaju upotrebu i pružaju veću funkcionalnost. Među njima su *end-of-test* mehanizam za olakšavanje završavanja testova, *callback* mehanizam, lakša podešavanja izveštaja, ... UVM 1.0 je pratilo nekoliko manjih izdanja gde su ispravljene sitne greške i dodate nove funkcionalnosti uključujući registarski sloj, faze, sekvence, TLM interfejse, ... Trenutna verzija UVM-a je UVM 1.2 koji je izašao 2014 godine. UVM biblioteka, kao i prethodne verzije se mogu preuzeti sa http://www.accellera.org/downloads/standards/uvm

2 UVM verifikaciono okruženje

U ovom poglavlju je dat kratak pregled klasičnog UVM testbenča. Ukratko je objašnjena struktura testbenča, navedene osnovne komponente, način kreiranja testova, ali i osnovni elementi UVM biblioteke. Dat je samo kratak pregled i uloga, a sve komponente i navedene UVM-ove mogućnosti će biti detaljno razrađene u narednim vežbama.

2.1 Tipična struktura testbenča

Svaki UVM testbenč se sastoji od određenog broja verifikacionih komponenti sa jasno definisanom ulogom, koje su povezane na odgovarajući način, prate metodologiju i omogućavaju laku ponovnu upotrebu. Na blok dijagramu (slika 1) je prikazana tipična struktura testbenča.



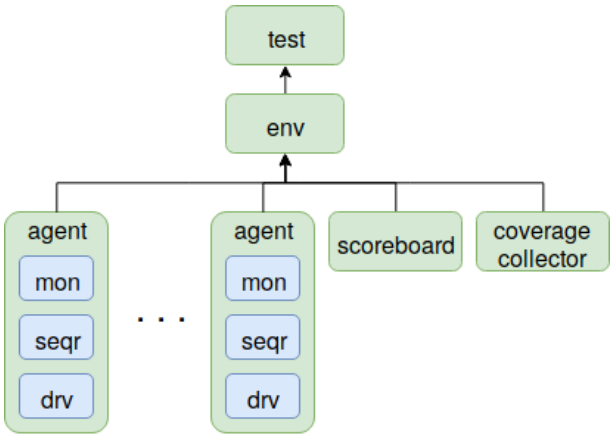
Slika 1: UVM testbenc

Prikazani delovi testbenča su:

- *data objekat* (paket, *sequence item*, transaction) - koristi se kao stimulus za dizajn koji se verifikuje. Mora sadržati polja koja su potrebna drajveru da bi uspešno inicirao transakciju, ali može sadržati i neka dodatna polja koja se koriste za kontrolu, kao i ograničenja pri randomizaciji
- sekvencer komponenta - je stimulus generator. U njemu se kontrolišu podaci koje će drajver slati DUT-u. Kontrola podataka se vrši u definisanim sekvencama koje određuju broj paketa, redosled, sadržaj i sl. Kreiranje sekvenci je glavni način generisanja stimulusa u UVM-u
- drajver komponenta - je aktivna komponenta koja emulira signale koji se šalju dizajnu. Na osnovu podataka od sekvencera i semplovanja signal na interfejsu, generiše signale na ulaze dizajna koji se verifikuje, prateći implementirani protokol
- monitor komponenta - je pasivna komponenta zadužena za nadgledanje signala, ali ne i njihovo generisanje. Može sakupljati podatke o pokrivenosti, ali i raditi proveru. Sakupljene informacije šalje ostalim komponentama na dalju obradu (npr. *scoreboard*-u na dalju kontrolu)

- agent komponenta - obuhvata drajver, sekvencer i monitor. Iako se ove podkomponente mogu koristiti nezavisno to iziskuje poznavanje imena, uloge i načina povezivanja. Agent obuhvata ove tri komponente, predstavlja apstraktniji pristup datom interfejsu i olakšava ponovno korišćenje ovih komponenti. Aktivan agent instancira sve tri komponente, dok pasivan instancira samo monitor (pasivan agent ne generiše signale, služi samo za nadgledanje; uglavnom je deo većeg okruženja, gde se generisanje podataka vrši na nekom višem nivou)
- *scoreboard* - na osnovu podataka dobijenih iz monitora vrši proveru
- konfiguracioni objekat - sadrži sve podatke potrebne za konfigurisanje testbenča npr. broj agenata, da li je potreban master ili *slave*, da li je agent aktivan ili pasivan i sl.
- okruženje - obuhvata sve prethodno navedene delove u jednu klasu radi jednostavnog korišćenja
- test

Hijerarhija testbenča je određena nizom *has-a* klasnim vezama tj. komponente na višem nivou hijerarhije sadrže komponente na nižem nivou. Npr. agent sadrži drajver, sekvencer i monitor. Ovo je prikazano na slici 2.



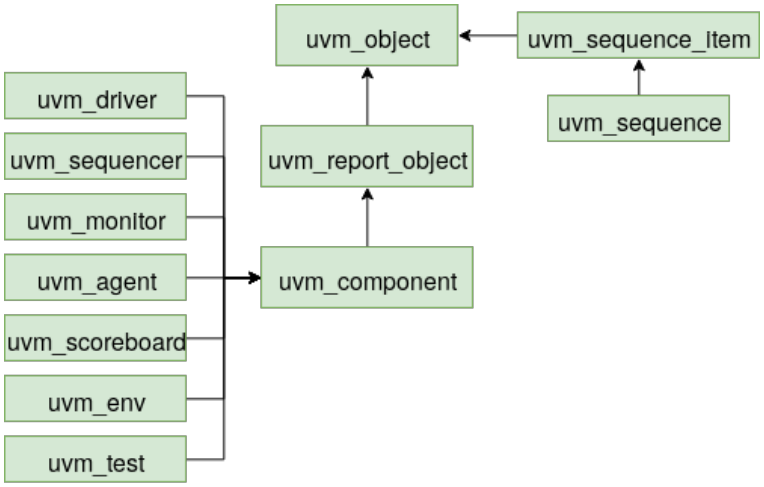
Slika 2: UVM hijerarhija

Sa ove slike se vidi i da je test na najvišem nivou hijerarhije. On je zadužen za instanciranje i konfiguraciju okruženja i startovanje sekvenci. Često postoji bazna test klasa u kojoj se vrše instanciranje okruženja, dok se u izvedenim testovima vrše specifične operacije za taj test npr. redosled sekvenci, randomizacija i sl.

2.2 UVM klasna hijerarhija

Na slici 3 je prikazana delimična UVM klasna hijerarhija. Nasleđivanje omogućava brz razvoj dobro struktuiranih verifikacionih komponenti i objekata. Bazne klase sadrže potrebna polja i metode za kontrolu, pa je, umesto pisanja svake komponente od nule, potrebno samo modifikovati određene metode za konkretne potrebe.

Kao što se vidi sa slike 3, svaki deo testbenča opisan u prethodnom poglavlju ima odgovarajuću UVM klasu i kako bi se implementirao potrebno je naslediti tu klasu. Npr. ukoliko želimo da implementiramo monitor, potrebno je napisati novu klasu koja nasleđuje *uvm_ monitor*. Slično će drajver nasleđivati *uvm_ driver*, sekvencer *uvm_ sequencer* i sl.



Slika 3: UVM klasna hijerarhija

Takođe se može uočiti da sve klase nasleđuju *uvm_ object* klasu. Jedna od bitnijih podklasa je *uvm_ component*. Klasa komponenta je izvedena iz klase objekat, međutim sadrži veoma bitne osobine zbog kojih se često posebno izdvaja. Prvenstveno sadrži UVM-ov mehanizam faza, mogućnost korišćenja konfiguracija i TLM interfejse. Sve klase koje se koriste za kreiranje komponenti verifikacionog okruženja (*uvm_ driver*, *uvm_ monitor*, *uvm_ env*, ...) nasleđuju *uvm_ component* klasu. Obično se u literaturi kao dve glavne grupe navode objekti i komponente, pri čemu se pravi razlika između klasa nasleđenih iz *uvm_ object* i klasa nasleđenih iz *uvm_ component* (iako su i ove klase naseđene iz *uvm_ object*, izdvajaju se zbog velikog broja posebnih osobina).

2.3 UVM faze

UVM-ov mehanizam faza je jedna od najbitnijih funkcionalnosti i novina u ovoj biblioteci. Kao što je već navedeno, jedna od osnovnih mogućnosti *uvm_ component* klase su upravo faze. Pošto u verifikacionom okruženju postoji veliki broj komponenti koje rade u paraleli, potrebno je obezbediti da se određene funkcionalnosti izvrše u jasno definisanim trenucima. Faze predstavljaju sinhronizacioni mehanizam za okruženje odnosno sve komponente su uvek sinhronizovane u pogledu faza.

Postoje tri osnovne grupe UVM faza:

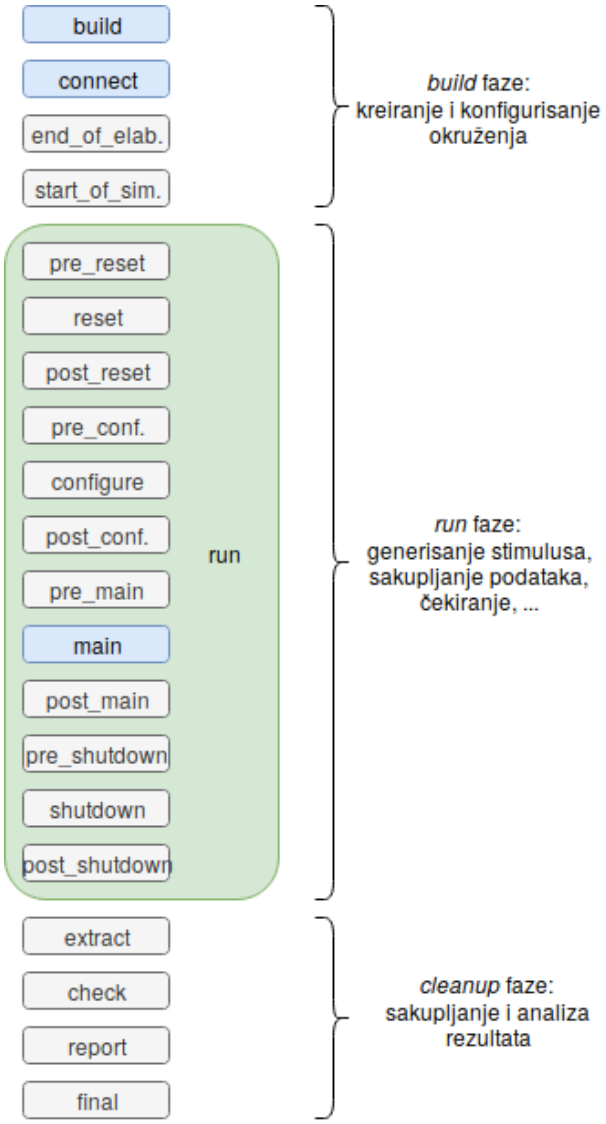
- *build* faze - za kreiranje i konfigurisanje okruženja
- *run-time* faze - gde zapravo teče simulaciono vreme
- *clean up* faze - sakupljanje i analiza rezultata

Ove faze su zapravo samo virtuelne metode u *uvm_ component* klasi, čijim se modifikacijama omogućava jasno definisan *flow* u okruženju. Korišćenjem faza se povećava mogućnost ponovne upotrebe komponenti jer je uloga svake faze, i funkcionalnosti koje se u njoj izvršavaju, jasno definisana. Na slici 4 su prikazane sve UVM faze i grupisane u tri osnovne grupe.

U tabeli 1 je dat kratak opis svih faza u redosledu izvršavanja.

Run faza ima veliki broj pod-faza koijma se može vršiti detaljnija sinhronizacija. One su date u tabeli 2.

UVM pruža velike mogućnosti pri korišćenju ovih faza. Kao što se vidi, *run-time* faze obuhvataju veliki broj pod-faza čija je preporučena uloga i redosled jasno definisan. Radi lakšeg prelaska



Slika 4: UVM faze

Faza	**Opis**
build	instanciranje podkomponenti, podešavanje konfiguracija, ...
connect	međusobno povezivanje komponenti, podešavanje interfejsa, ...
end_of_elaboration	prikaz topologije, naknadne izmene (u ovoj fazi je okruženje potpuno kreirano i povezano)
start_of_simulation	prikaz topologije, priprema za početak simulacije, podešavanja *run-time* konfiguracija, simulatora, ...
run	sama simulacija (generisanje stimulusa, provera, ...)
extract	preuzimanje preostalih podataka iz komponenti (npr. informacije iz *scoreboard*-a ili monitora)
check	preostale provere
report	prikaz rezultata ili upis rezultata u fajlove
final	završetak svih preostalih radnji (npr. zatvaranje fajlova ili terminacija simulatora)

Tabela 1: UVM faze

Faza	**Opis**
pre_reset	sve aktivnosti koje treba obaviti pre reseta
reset	specifično ponašanje tokom reseta
post_reset	sve aktivnosti koje treba obaviti nakon reseta
pre_configure	priprema DUT-a za konfiguraciju posle reseta
configure	konfigurisanje DUT-a
post_configure	čekanje da DUT odreaguje na konfiguraciju
pre_main	provera da su sve komponente spremne za rad
main	glavna faza - generisanje stimulusa, sakupljanje podataka, čekiranje, ...
post_main	finalizacija aktivnosti iz *main* faze
pre_shutdown	pomoćna faza za aktivnosti pre *shutdown* faze
shutdown	provera da je generisan stimulus korektno propagiran, provera statusnih registara, ...
post_shutdown	finalizacija aktivnosti, poslednja aktivna faza

Tabela 2: UVM *run* faze

sa OVM-a na UVM, omogućeno je korišćenje i samo *run* faze umesto svih pod-faza (pogledati sliku 4, *run* obuhvata *reset*, *main*, *post_main*, ..).

Dve najčešće korišćene faze su *build* i *run* (ili *main*), pa će kostur većine komponenti biti (primetiti da je *run/main* faza implementirana kao task zato što troši simulaciono vreme):

```
class example extends uvm_component;

  // polja i podkomponente
  // ...

  `uvm_component_utils(example) // factory registration

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // ...
  endfunction : build_phase
```

```
  task main_phase(uvm_phase phase);
    super.main_phase(phase);
    // ...
  endtask : main_phase

endclass : example
```

2.4 UVM *factory*

Factory metoda je jedna od klasičnih softverskih dizajn šablona (engl. *desing pattern*). Prilikom funkcionalne verifikacije često je potrebno uvesti promene u određenim klasama. Ovo se može desiti zbog uvođenja nove funkcionalnosti, specifičnih zahteva nekog testa i sl. Na primer, možda je potrebno dodati nova polja u transakciju ili modifikovati način generisanja signala ili neka ograničenja pri randomizaciji. UVM *factory* omogućava jednostavan način zamene postojeće klase izvedenom klasom bez menjanja ostatka okruženja. Primer: ukoliko u okruženju koristimo *old_transaction* klasu za slanje podataka, a želimo da koristimo *new_transaction* koja ima dodata određena ograničenja za randomizaciju potrebna za dati test. Ovo je moguće postići jednom linijom koda. Tada se sve instance *old_transaction* klase zamenjuju sa *new_transaction*, bez menjanja ostatka koda. Odnosno u celom okruženju se koristi *new_transaction* umesto *old_transaction*. Ovaj mehanizam se naziva *factory override* i biće obrađen u narednim vežbama.

Kako bi ovo bilo moguće, potrebno je pratiti određena pravila prilikom implementacije klasa, objašnjena u nastavku. Konvencija je uvek pisati kod na ovaj način radi lakše ponovne upotrebe.

2.4.1 Registracija

UVM objekti i komponente treba da sadrže kod za *factory* registraciju koji sadrži *wrapper* klasu oko date klase (mora biti *typedef*-ovana na *type_id*), statičnu funkciju da se dobije ovaj *type_id*) i funkciju za dobijanje imena tipa. Međutim, pošto se pri svakoj registraciji moraju obaviti ovi koraci, koji prate dati šablon, razvijeni su određeni *factory registration* makroi. Konvencija je koristiti ove makroe umesto ručnog pisanja potrebnog koda kako bi se izbegle eventualne greške i povećala čitljivost koda. Postoje četiri makroa:

- ` uvm_component_utils(<name>) - za komponente
- ` uvm_component_param_utils(<name>) - za parametrizovane komponente
- ` uvm_object_utils(<name>) - za objekte
- ` uvm_object_param_utils(<name>) - za parametrizovane objekte

Ukoliko je potrebno i korišćenje *field* makroa, umesto ovih koriste se *begin..end* makroi koji kreiraju blok unutar kojeg se navode *field* makroi. Npr.

```
`uvm_object_utils_begin(<tip>)
  `uvm_field_* macro invocations here
`uvm_object_utils_end
```

Sami *field* makroi (makroi za polja u klasi) ne služe za registraciju već za implementaciju pomoćnih metoda (*copy*, *compare*, *print*, *pack*, ...). Njihova upotreba je opciona i biće demonstrirana u poglavlju o transakcijama.

Spisak svih *utility* makroa se može pronaći na https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/macros/uvm_object_defines-svh.html

2.4.2 Podrazumevane vrednosti konstruktora

Pošto su konstruktori u *uvm_object* i *uvm_component* klasi zapravo virtuelne funkcije, potrebno je pratiti njihov prototip. Kako bi se omogućila ispravna konstrukcija, konstruktor treba da sadrži podrazumevane vrednosti za sve argumente.

Postoji razlika između komponenti i objekata:

```
// komponenta
function new(string name = "example ", uvm_component parent = null);
  super.new(name, parent);
endfunction

// objekat
function new(string name = "example");
  super.new(name);
endfunction
```

2.4.3 Kreiranje komponenti i objekata

Kako bi *factory* mogao da ispravno kreira objekte i komponente, potrebno je koristiti *create* metodu umesto direktnog pozivanja *new*. Npr.

```
class top_component extends uvm_component;

  sub_component comp; // some uvm_component
  sub_object obj; // some uvm_object

  `uvm_component_utils (top_component)

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

  function void build_phase(uvm_phase phase);
    super.build_phase();
    comp = sub_component::type_id::create("comp", this);
    obj = sub_object::type_id::create("obj");
    // umesto comp = new; obj = new;
  endfunction : build_phase

endclass : top_component
```

Prateći prethodne konvencije, skeleton svake objekat klase će biti nalik:

```
class example_class extends uvm_object;

  `uvm_object_utils (example_class)

  function new(string name = "example");
    super.new(name);
    // ...
  endfunction

  // ...

endclass : example
```

Slično i za komponente, koristeći odgovarajuće makroe i podrazumevane vrednosti konstruktora.

Sam mehanizam *override*-ovanja će biti obrađen u narednim vežbama.

2.5 UVM poruke

Za ispis poruka u SystemVerilog-u, koristili smo veliki broj ugrađenih funkcija:$*display*, $*write*, $*error*, $*warning*, ... Iako veoma korisne, UVM ipak pruža mnogo veće mogućnosti pri ispisu poruka uključujući kontrolu ispisa pojedinih poruka i kontrolu same simulacije. Zbog toga je preporuka uvek koristiti UVM-ov mehanizam poruka. Korišćenje ovog mehanizma je jako pojednostavljeno i potrebno je samo pozvati određene makroe, kojima prosleđujemo željene argumente.

2.5.1 *Severity*

U zavisnosti od značaja same poruke, postoji izbor od nekoliko makroa. Oni su:

- ‘uvm_info(string <id>, string <mssg>, <verbosity>)
- ‘uvm_warning(string <id>, string <mssg>)
- ‘uvm_error(string <id>, string <mssg>)
- ‘uvm_fatal(string <id>, string <mssg>)

Argumenti su:<id> za identifikaciju klase koja vrši ispis poruke; obično se navodi ime komponente, <mssg> je string koji sadrži samu poruku.<verbosity> služi za podešavanje nivoa i objašnjen je u narednom poglavlju. Razlika između ovih makroa je u akciji koja se vrši njihovim pozivanjem (tabela 3).

Značaj	**Akcija**	**Opis**
UVM_INFO	ispis poruke ukoliko je podešen odgovarajući nivo	koristi se za ispis raznih poruka; moguće podešavanje nivoa
UVM_WARNING	ispis poruke	koristi se za ispis upozorenja
UVM_ERROR	ispis poruke i završetak simulacije ukoliko se prevaziđe max broj dozvoljenih *error*-a	koristi se za ispis pronađenih grešaka u DUT-u
UVM_FATAL	ispis poruke i završetak simulacije	koristi se ukoliko postoji neka greška (bilo u okruženju, bilo u DUT-u) zbog koje nije moguće nastaviti simulaciju

Tabela 3: UVM *severity*

Par primera:

```
`uvm_fatal("NOCONFIG", {"Config object must be set for: ", get_full_name(), ".cfg"})
`uvm_error(get_type_name(), $sformatf("Incorrect value observed: expected %0d, received %0d", expected,
    received))
`uvm_warning(get_type_name(), "Inputs should be held low during reset")
```

get_type_name() funkcija vraća ime tipa koji je poziva i često se korisiti kao prvi argument odnosno ID poruke. $*sformatf* je SystemVerilog-ova funkcija za manipulaciju stringova, nalik sličnim C-ovskim funkcijama, dok se u *fatal* primeru koristi konkatenacija stringova.

Još jedna prednost korišćenja ovih makroa je ispis na kraju simulacije koji prikazuje broj prikazanih poruka, gde lako možemo uočiti da li su pronađene greške tokom simulacije:

```
# ** Report counts by severity
# UVM_INFO :      5
```

```
# UVM_WARNING:    0
# UVM_ERROR:      0
# UVM_FATAL:      0
```

2.5.2 *Verbosity*

Kao što je već napomenuto, UVM pruža mehanizam kontrole ispisa poruka. Ovaj mehanizam je zasnovan na prosleđivanju određene vrednosti svakoj poruci i zatim ispisu samo onih poruka sa odgovarajućim nivoima. Ovi predefinisani nivoi se mogu koristiti na dva načina - za pojedinačne poruke ili za celokupne komponente. Mi ćemo se zadržati na porukama.

Verbosity se podešava preko nekoliko predefinisanih *enum* vrednosti. One su prikazane u tabeli 4.

Nivo	**Vrednost**
UVM_NONE	0
UVM_LOW	100
UVM_MEDIUM	200
UVM_HIGH	300
UVM_FULL	400
UVM_DEBUG	500

Tabela 4: UVM *verbosity*

Kada setujemo nivo u testu, ispisivaće se sve poruke koje imaju taj ili niži nivo. Podrazumevani nivo je UVM_MEDIUM odnosno ispisivaće se poruke sa UVM_MEDIUM, UVM_LOW i UVM_NONE, odnosno filtriraće se sve poruke sa nivoom UVM_HIGH, UVM_FULL i UVM_DEBUG. Ako odaberemo UVM_FULL nivo, ispisivaće se sve poruke osim onih sa UVM_DEBUG nivoom.

Ispis UVM_NONE poruka se ne može isključiti.

Prilikom pisanja informacija treba koristiti ove nivoe na efikasan način, odnosno poruke koje služe samo za testiranje okruženja treba da imaju visoku vrednost, dok one koje će uvek biti od interesa treba da imaju nižu vrednost odnosno da se skoro uvek ispisuju.

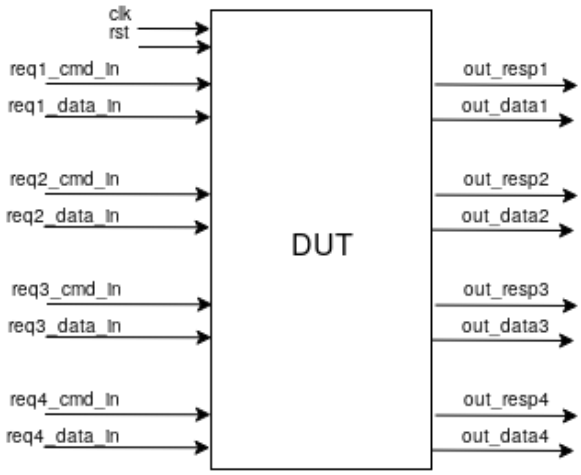
Setovanje nivoa tokom simulacije je objašnjeno u poslednjem poglavlju, a u nastavku je dato par primera poruka sa različitim nivoima:

```
`uvm_info(get_type_name(), "Reset observed", UVM_MEDIUM)
`uvm_info(get_type_name(), "Configuration written", UVM_LOW)
`uvm_info(get_type_name(), "Collected transaction", UVM_HIGH)
```

3 DUT

U ovoj i narednim vežbama ćemo se baviti verifikacijom "Calc1" dizajna. Sve komponente će biti razvijane za ovaj primer. Ovaj dizajn se koristio i kao primer na predavanjima, dok ćemo na vežbama izvršiti kompletnu verifikaciju, prateći UVM metodologiju. U nastavku je data funkcionalna specifikacija DUT-a.

"Calc1" je kalkulator sa četiri porta koji podržava četiri operacije. Na svakom portu se može slati po jedna komanda koja može biti sabiranje, oduzimanje, šiftovanje u levo i desno. Pored komande, šalju se i prateći podaci. Pošto postoje četiri porta, mogu se postojati četiri aktivne komande u jednom trenutku. Komande i podaci se šalju u istom taktu, dok može proći nekoliko taktova pre dobijanja rezultata. Svaka komanda će dobiti odgovor i rezultat ukoliko nije došlo do greške. Do god se ne očita rezultat, nije dozvoljeno slanje nove komande. DUT sadrži dve aritmetičko-logičke jedinice, jedna procesira sve komande sabiranja, a druga je zadužena za komande šiftovanja. Prioritetnom logikom se šalju komande ka aritmetičko-logičkim jedinicama.



Slika 5: DUT

U narednoj tabeli je opisan interfejs našeg DUT-a.

Kao što se vidi u tabeli 5, svaki od četiri porta ima po dva ulazna i dva izlazna signala. *regX_ cmd_ in* signal je četvorobitni ulazni signal za slanje komande. U tabeli 6 su prikazane moguće komande i njihova vrednost.

regX_ data_ in je 32-bitni ulaz za podatke. Dva podatka koji predstavljaju operande se šalju na ovoj liniji, pri čemu se prvo šalje operand 1, a zatim operand 2 u sledećem taktu. Operand 1 se šalje u istom taktu kada i komanda. I odavde vidimo da je potrebno dva takta da bi se poslao zahtev DUT-u. U zavisnosti od komande, ova dva operanda se tumače na različite načine, koji su prikazani u tabeli 7.

Postoje i dve izlazne linije za svaki port. *out_ respX* je dvobitni signal za odgovor koji daje informacije o uspešnosti operacije. Vrednosti su:

Rezultat operacije se nalazi na *out_ dataX* liniji i validan je jedino u ciklusima u kojima je *out_ respX* = 2'b01, odnosno kada je operacija uspešna.

Na slici 6 je prikazana uspešna operacija na portu 1. Komanda i prvi operand se šalju u jednom

Ime	**Širina (br. bita)**	**Smer**	**Opis**
c_clk	1	in	takt
reset	8	in	reset
reg1_cmd_in	4	in	komanda za port 1
reg1_data_in	32	in	podaci za port 1
reg2_cmd_in	4	in	komanda za port 2
reg2_data_in	32	in	podaci za port 2
reg3_cmd_in	4	in	komanda za port 3
reg3_data_in	32	in	podaci za port 3
reg4_cmd_in	4	in	komanda za port 4
reg4_data_in	32	in	podaci za port 4
out_resp1	2	out	odgovor za port 1
out_data1	32	out	rezultat za port 1
out_resp2	2	out	odgovor za port 2
out_data2	32	out	rezultat za port 2
out_resp3	2	out	odgovor za port 3
out_data3	32	out	rezultat za port 3
out_resp4	2	out	odgovor za port 4
out_data4	32	out	rezultat za port 4

Tabela 5: DUT interfejs

Komanda	**Vrednost**
No operation	4'b0000
Add	4'b0001
Subtract	4'b0010
Shift left	4'b0101
Shift right	4'b0110
Invalid	sve preostale vrednosti

Tabela 6: DUT komande

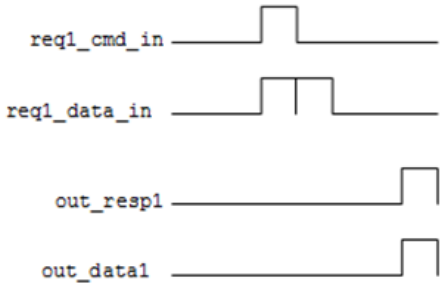
Komanda	**Tumačenje operanada**
Add	Rezultat je operand1 + operand2
Subtract	Rezultat je operand1 - operand2
Shift left	Rezultat je operand1 šiftovan u levo za operand2 mesta, pri čemu se koriste samo 5 bita drugog operanda. Uvek se ubacuju nule.
Shift right	Rezultat je operand1 šiftovan u desno za operand2 mesta, pri čemu se koriste samo 5 bita drugog operanda. Uvek se ubacuju nule.

Tabela 7: DUT operandi

Vrednost	**Značenje**
2'b00	Nema odgovora u ovom ciklusu.
2'b01	Uspešna operacija. Rezultujući podaci se nalaze na *out_dataX* liniji.
2'b10	*Overflow*, *underflow* ili pogrešna komanda. *Overflow/underflow* je moguć samo za operacije sabiranja i oduzimanja.
2'b11	Ne koristi se.

Tabela 8: DUT odgovor

ciklusu, a drugi operand se šalje u narednom ciklusu. Nakon nekoliko taktova dobija se odgovor i rezultat u istom ciklusu. Broj ciklusa koji je potreban da se dobije odgovor zavisi od aktivnosti na ostalim portovima (pošto postoje samo dve ALU), ali će uvek biti minimum 3 ciklusa.



Slika 6: Primer operacije na portu 1

Na svakom portu može biti aktivna samo jedna operacija, odnosno kada se pošalje jedna komanda, sledeća se ne sme slati do god ne primimo odgovor za prvu komandu. Takođe, slanje nove komande je opciono tj. port može biti neaktivan.

Četiri porta su nezavisna. Mogu se paralelno slati komande na svim ili samo nekim portovima, što znači da u bilo kom trenutku DUT može obavljati maksimum 4 operacije. Međutim, brzina odgovora će zavisiti od aktivnosti zbog ograničenih resursa u dizajnu. Kako što je već rečeno, DUT sadrži dve ALU, jednu za sabiranje i oduzimanje, a drugu za šiftovanje. Ukoliko se primi više komandi u paraleli za jednu ALU, DUT će ih serijalizovati i obrađivati jednu po jednu. Svaki port ima isti prioritet. DUT će obrađivati komande po principu *first-come, first-serve*, pri čemu za komande koje stignu u istom taktu nije definisan redosled.

Što se tiče reseta, aktiviranje se vrši postavljanjem reset linije na vrednost 8'b11111111. Ova vrednost se mora držati 7 sukcesivnih taktova kako bi se reset uspešno propagirao kroz dizajn. Tokom reseta, svi ulazi, osim takta, treba da budu nula.

U "Calc1" dizajnu se sve aritmetičke operacije vrše nad *unsigned* podacima. Prilikom sabiranja *overflow* se dešava ukoliko najviši bit ima *carry-out*. Slično *underflow* se pojavljuje prilikom oduzimanja ukoliko se oduzima veći broj od manjeg broja. U nastavku je dato par primera.

Komanda	**Operand 1**	**Operand 2**	**Odgovor**	**Rezultat**
Add	32'h80002345	32'h00010000	Uspešna operacija	32'h80012345
Add	32'hFFFFFFFF	32'h00000001	*Overflow*	Nema
Subtract	32'hFFFFFFFF	32'h11111111	Uspešna operacija	32'hEEEEEEEE
Subtract	32'h11111111	32'h20000000	*Underflow*	Nema

Tabela 9: DUT primeri operacija

4 *Data (sequence) item*

Kao što je već navedeno, na ovim vežbama ćemo izvršiti verifikaciju "Calc1" dizajna. Prilikom razvoja verifikacionih okruženja, jedan od prvih koraka je implementacija klase koja predstavlja transakciju. U literaturi se sreću mnogi termini za ovaj objekat uključujući *data item, sequence item, frame, packet, transaction*, ...

Data (sequence) item predstavlja transakciju koja se koristi kao stimulus. Koristi se na mnogim mestima u verifikacionom okruženju uključujuci slanje podataka od sekvencera ka drajveru. Slanjem više ovih transakcija ka drajveru se kreiraju sekvence koje su osnova za generisanje stimulusa. Veza između drajvera i sekvencera, kao i način pisanja sekvenci će biti detaljno opisani u narednoj vežbi, dok je u ovoj vežbi fokus na modelovanju transakcija. Sama transakcija treba da sadrži sve informacije potrebne drajveru da ispravno generiše signale na interfejsu, ali može i sadržati neke dodatne podatke koji služe za kontrolu.

Ove transakcije će se implementirati kao klase koje nasleđuju (direktno ili indirektno) *uvm_sequence_item* klasu. Polja u klasi zavise od potreba drajvera, nivoa apstrakcije, ali i potreba ostatka okruženja. Uglavnom se ti podaci mogu podeliti u nekoliko grupa:

- Kontrolna polja - npr. tip transfera, veličina, ...
- *Payload* polja - sami podaci koji se šalju, npr. vrednost adrese
- Konfiguraciona polja - npr. ponašanje prilikom grešaka
- Polja za analizu - pomoćna polja koja olakšavaju analizu, npr. vreme

Sam princip generisanja stimulusa u *constrained-random* verifikaciji se zasniva na randomizaciji ovih transackija kako bi se pokrili interesantni scenariji. Samim tim, veliki broj polja je često deklarisan kao *rand*, uz određena ograničenja kako bi se generisale željene transakcije.

Sadržaj transakcija je opisan na primeru transakcije za APB i UART protokol. U ovom primeru se vidi da transackija može sadržati željeni broj polja, ali i metode koje olakšavaju njeno korišćenje (u primeru UART *seq item*-a postoji metoda za računanje parnosti).

```
`ifndef APB_SEQ_ITEM_SV
`define APB_SEQ_ITEM_SV

parameter ADDR_WIDTH  = 32;
parameter RDATA_WIDTH = 32;
parameter WDATA_WIDTH = 32;
typedef enum {
              APB_READ = 0,
              APB_WRITE = 1
              } apb_direction_enum;

class apb_seq_item extends uvm_sequence_item;

  // polja
  rand bit [ADDR_WIDTH - 1 : 0] addr;
  rand apb_direction_enum       dir;
  rand bit [RDATA_WIDTH - 1 : 0] rdata;
  rand bit [WDATA_WIDTH - 1 : 0] wdata;
  rand int unsigned             delay = 0;
  bit          error;

  // ogranicenja
  constraint c_delay { delay <= 10 ; }

  // UVM factory registracija
```

```
    `uvm_object_utils_begin(apb_seq_item)
        `uvm_field_int(addr, UVM_DEFAULT)
        `uvm_field_enum(apb_direction_enum, dir, UVM_DEFAULT)
        `uvm_field_int(rdata, UVM_DEFAULT)
        `uvm_field_int(wdata, UVM_DEFAULT)
        `uvm_field_int(delay, UVM_DEFAULT)
        `uvm_field_int(error, UVM_DEFAULT)
    `uvm_object_utils_end

    // konstruktor
    function new(string name = "apb_seq_item");
        super.new(name);
    endfunction : new

endclass

`endif
```

Kod 1: Primer APB transakcije

```
`ifndef UART_SEQ_ITEM_SV
 `define UART_SEQ_ITEM_SV

typedef enum bit {GOOD_PARITY, BAD_PARITY} parity_e;
parameter PAYLOAD_WIDTH = 8;
parameter STOP_WIDTH = 8;
parameter ERR_WIDTH = 8;

class uart_seq_item extends uvm_sequence_item;

    rand bit start_bit;
    rand bit [PAYLOAD_WIDTH-1:0] payload;
    bit      parity;
    rand bit [STOP_WIDTH-1:0] stop_bits;
    rand bit [ERR_WIDTH-1:0] error_bits;
    rand parity_e parity_type;
    rand int unsigned delay;

    // ogranicenja
    constraint c_delay       {delay < 20;}
    constraint c_start_bit   {start_bit == 0;}
    constraint c_stop_bits   {stop_bits == '1;}
    constraint c_parity_type {parity_type == GOOD_PARITY;}
    constraint c_error_bits  {error_bits == 0;}

    // UVM factory registracija
    `uvm_object_utils_begin(uart_seq_item)
        `uvm_field_int(start_bit, UVM_DEFAULT)
        `uvm_field_int(payload, UVM_DEFAULT)
        `uvm_field_int(parity, UVM_DEFAULT)
        `uvm_field_int(stop_bits, UVM_DEFAULT)
        `uvm_field_int(error_bits, UVM_DEFAULT)
        `uvm_field_enum(parity_e,parity_type, UVM_DEFAULT)
        `uvm_field_int(delay, UVM_DEFAULT)
    `uvm_object_utils_end

    // konstruktor
    function new(string name = "uart_seq_item");
        super.new(name);
    endfunction

    // metoda za racunanje parnosti
    function bit calc_parity(int unsigned num_of_data_bits = 8, bit[1:0] parity_mode = 0);
        bit      parity;

        if (num_of_data_bits == 6)
          parity = ^payload[5:0];
```

```
      else if (num_of_data_bits == 7)
        parity = ^payload[6:0];
      else
        parity = ^payload;

      case(parity_mode[0])
        0: parity = ~parity;
        1: parity = parity;
      endcase

      case(parity_mode[1])
        0: parity = parity;
        1: parity = ~parity_mode[0];
      endcase

      if (parity_type == BAD_PARITY)
        return ~parity;
      else
        return parity;
  endfunction

  // parnost se racuna posle randomizacije, na osnovu dodeljenih vrednosti
  function void post_randomize();
     parity = calc_parity();
  endfunction : post_randomize

endclass

`endif
```

Kod 2: Primer UART transakcije

Nasleđivanjem iz *uvm_sequence_item* klase i korišćenjem raznih UVM field makroa, omogućava se korišćenje mnogobrojnih metoda za rad sa ovom klasom uključujući *print, copy, compare* itd. Primer ispisa print metode je takođe dat ispod. Vidi se da korišćenje ovih metoda može biti veoma korisno i umnogome olakšati *debug*.

```
# ------------------------------------------------
# Name           Type         Size   Value
# ------------------------------------------------
# uart_frame     uart_frame   -      @366
#   start_bit    integral     1      'h0
#   payload      integral     8      'hcc
#   parity       integral     1      'h1
#   stop_bits    integral     2      'h3
#   error_bit    integral     4      'h0
#   parity_type  parity_e     1      GOOD_PARITY
#   delay        integral     32     'd13
# ------------------------------------------------
```

5 Simulacija

5.1 Kompilacija

Prilikom kompajliranja koda koji sadrži UVM biblioteku, potrebno je uključiti direktorijum koji sadrži UVM kod. Npr. (ukoliko UVM_HOME pokazuje na sors kod):

```
vlog -sv +incdir+$env(UVM_HOME) calc_verif_top.sv
```

Napomena: ova podešavanja se odnose na alat koji se koristi na vežbama. Kod novijih verzija alata podešavanja mogu biti drugačija.

5.2 UVM *command line processor*

Command line processor klasa pruža mogućnosti preuzimanja informacija koje se daju preko komandne linije prilikom poziva simulacije. Mogućnosti ovog procesora su mnogobrojne i izlaze iz opsega ovog kursa. Mi ćemo se zadržati na podešavanjima samo nekoliko UVM varijabli sa komandne linije koje umnogome olakšavaju simulaciju - podešavanje opširnosti ispisa poruka (*verbosity*) i definisanje testa koji se pušta. Za detaljnije objašnjenje rada *uvm_cmdline_processor* klase pogledati "UVM Users Guide", poglavlje 6.6.

Test se pušta pozivom *run_test(<test_name>)* rutine u top modulu. Tada se kreira instanca datog testa i pokreće se simulacija. Sledeći primer pokreće test pod nazivom *test_example*:

```
module top;

  import uvm_pkg::*;
  `include "uvm_macros.svh"

  // instantiate the DUT, connect,
   initial begin
     run_test("test_example");
   end
endmodule : top
```

Mane ovakvog pristupa su očigledne - svaki put kada želimo da pustimo drugi test, mora se menjati sors kod. Zbog toga se u praksi uvek koristi komandna linija za izbor testa. Kada se *run_test* pozove bez argumenta, proverava se +UVM_TESTNAME=<test_name> opcija. Sada se ne mora menjati sors kod, odnosno može se puštati više testova bez ponovnog kompajliranja čime se znatno štedi na vremenu. Primer prosleđivanja testa preko komandne linije:

```
vsim top +UVM_TESTNAME=test_example
```

+UVM_VERBOSITY opcija pruža mogućnost promene verbosity u čitavom okruženju odnosno kontrolu ispisa poruka. Nivoi su opisani u drugom poglavlju ove vežbe (UVM_LOW, UVM_HIGH, ...). Primer upotrebe:

```
vsim top +UVM_VERBOSITY=UVM_HIGH
```

+UVM_MAX_QUIT_COUNT opcija pruža mogućnost završetka simulacije kada se dostigne dati broj grešaka u dizajnu. Ova opcija je posebno korisna tokom puštanja regresije. Podrazumevana vrednost je -1 odnosno simulacija se nikad ne završava na osnovu broja grešaka. Pod brojem grešaka se podrazumeva broj prijavljenih *uvm_error*-a.

6 Zadaci

Zadatak U pratećim fajlovima za ovu vežbu je povezan naš DUT sa interfejsom i kreiran kostur za UVM test klasu. Proučiti date fajlove i način puštanja testova. Za dati DUT kreirati *sequence item* klasu koja će sadržati potrebna polja. Na osnovu funkcionalne specifikacije zaključiti koja polja treba da sadrži ova klasa. Da li će polja biti *rand* i zašto? Napisati ograničenja za randomizaciju ukoliko su potrebna.

Zadatak Napisati test koji instancira *sequence item* kreiran u prethodnom zadatku, randomizuje ga i u *main* fazi ispisuje vrednost svih polja. Pokrenuti test i preko komandne linije kontrolisati ispis poruka.

7 Appendix

```
`ifndef CALC_IF_SV
`define CALC_IF_SV

interface calc_if (input clk, logic [6 : 0] rst);

   parameter DATA_WIDTH = 32;
   parameter RESP_WIDTH = 2;
   parameter CMD_WIDTH = 4;

   logic [DATA_WIDTH - 1 : 0] out_data1;
   logic [DATA_WIDTH - 1 : 0] out_data2;
   logic [DATA_WIDTH - 1 : 0] out_data3;
   logic [DATA_WIDTH - 1 : 0] out_data4;
   logic [RESP_WIDTH - 1 : 0] out_resp1;
   logic [RESP_WIDTH - 1 : 0] out_resp2;
   logic [RESP_WIDTH - 1 : 0] out_resp3;
   logic [RESP_WIDTH - 1 : 0] out_resp4;
   logic [CMD_WIDTH - 1 : 0]  req1_cmd_in;
   logic [DATA_WIDTH - 1 : 0] req1_data_in;
   logic [CMD_WIDTH - 1 : 0]  req2_cmd_in;
   logic [DATA_WIDTH - 1 : 0] req2_data_in;
   logic [CMD_WIDTH - 1 : 0]  req3_cmd_in;
   logic [DATA_WIDTH - 1 : 0] req3_data_in;
   logic [CMD_WIDTH - 1 : 0]  req4_cmd_in;
   logic [DATA_WIDTH - 1 : 0] req4_data_in;

endinterface : calc_if

`endif
```

Kod 3: calc_if

```
`ifndef CALC_VERIF_PKG_SV
 `define CALC_VERIF_PKG_SV

package calc_verif_pkg;

   import uvm_pkg::*;      // import the UVM library
 `include "uvm_macros.svh" // Include the UVM macros

 `include "tests/v5_test_simple.sv"

endpackage : calc_verif_pkg

 `include "calc_if.sv"

`endif
```

Kod 4: v5_calc_verif_pkg

```
module calc_verif_top;

   import uvm_pkg::*;     // import the UVM library
`include "uvm_macros.svh" // Include the UVM macros

   import calc_verif_pkg::*;

   logic clk;
   logic [6 : 0] rst;

   // interface
   calc_if calc_vif(clk, rst);

   // DUT
```

```
    calc_top DUT(
                    .c_clk          ( clk ),
                    .reset          ( rst ),
                    .out_data1      ( calc_vif.out_data1 ),
                    .out_data2      ( calc_vif.out_data2 ),
                    .out_data3      ( calc_vif.out_data3 ),
                    .out_data4      ( calc_vif.out_data4 ),
                    .out_resp1      ( calc_vif.out_resp1 ),
                    .out_resp2      ( calc_vif.out_resp2 ),
                    .out_resp3      ( calc_vif.out_resp3 ),
                    .out_resp4      ( calc_vif.out_resp4 ),
                    .req1_cmd_in    ( calc_vif.req1_cmd_in ),
                    .req1_data_in   ( calc_vif.req1_data_in ),
                    .req2_cmd_in    ( calc_vif.req2_cmd_in ),
                    .req2_data_in   ( calc_vif.req2_data_in ),
                    .req3_cmd_in    ( calc_vif.req3_cmd_in ),
                    .req3_data_in   ( calc_vif.req3_data_in ),
                    .req4_cmd_in    ( calc_vif.req4_cmd_in ),
                    .req4_data_in   ( calc_vif.req4_data_in )
                    );

    // run test
     initial  begin
        run_test();
    end

    // clock and reset  init .
     initial  begin
        clk <= 0;
        rst <= 1;
        #50 rst <= 0;
    end

    // clock generation
    always #50 clk = ~clk;

endmodule : calc_verif_top
```

Kod 5: v5_calc_verif_top

```
`ifndef TEST_SIMPLE_SV
 `define TEST_SIMPLE_SV

class test_simple extends uvm_test;

    `uvm_component_utils(test_simple)

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

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // ...
    endfunction : build_phase

    task main_phase(uvm_phase phase);
        super.main_phase(phase);
        `uvm_info(get_type_name(), "Starting main phase...", UVM_HIGH)
        // ...
    endtask : main_phase

endclass : test_simple

`endif
```

Kod 6: v5_test_simple

```
# Create the library
if [file exists work] {
    vdel -all
}
vlib work

# compile DUT
vlog +incdir+../dut \
    ../dut/calc_top.v

# compile testbench
vlog +acc -sv \
    +incdir+$env(UVM_HOME) \
    ./*calc_verif_pkg.sv \
    ./*calc_verif_top.sv

# run simulation
vsim calc_verif_top +UVM_TESTNAME=test_simple +UVM_VERBOSITY=UVM_HIGH -sv_seed random
     -do "run -all"
```

Kod 7: calc_run