Funkcionalna verifikacija hardvera

Vežba 6
Rad sa sekvencama

Sadržaj

Ova vežba je posvećena opisu komunikacije između drajvera i sekvencera, opisu sekvenci i opisu raznih mogućnosti UVM-a u pogledu pisanja i korišćenja sekvenci. Dat je i opis mehanizma završetka testa.

1 Drajver - sekvencer interakcija

U ovom poglavlju je dat kratak pregled drajvera i sekvencera, ukratko je opisana njihova konekcija i opisane su često korišćene UVM konstrukcije. Sledeća vežba je posvećena UVM drajveru, a u ovoj vežbi je dat samo kratak pregled kako bi se razumela struktura sekvenci.

1.1 Drajver

Drajver je aktivna komponenta koja emulira signale koji se šalju dizajnu. Na osnovu podataka iz *sequence item*-a i semplovanja signal na interfejsu, postavlja signale na ulaze dizajna koji se verifikuje. Transakcije koje prima su obično na višem nivou apstrakcije, pa se vrši konverzija na pin-nivo apstrakcije kako bi se ispravno generisali signali. Drajver je preko TLM porta povezan na sekvencer kako bi vršio komunikaciju i sadrži virtuelni interfejs kako bi generisao signale. U UVM-u se drajver implementira nasleđujući *uvm_ driver #(REQ,RSP)* klasu, koja je parametrizovana tipom transakcija koju će drajver zahtevati i odgovarati (engl. *request*, *response*). Ukoliko se navede samo REQ i RSP će biti istog tipa kao i REQ. Npr:

```
class calc_driver extends uvm_driver#(calc_seq_item);
  // ...
endclass : calc_driver
```

Napomena: u SystemVerilog-u se parametrizacija vrši koristeći "#" znak. Nalik C++-su ovim mehanizmom se omogućava pisanje generičkih klasi kojima će se stvarna vrednost parametra proslediti tek prilikom korišćenja. Primer deklarisanja parametrizovane klase:

```
class param_example #(type T = int, type G = bit, int A = 10);
```

1.2 Sekvencer

Sekvencer koristi sekvence kako bi poslao podatke drajveru, ali i prima odgovor od drajvera ukoliko je to potrebno čime se omogućava kreiranje korisnih stimulusa. Sekvencer služi kao ariber čiji je zadatak kontrola slanja transakcija iz jedne ili više sekvenci.

U praksi najčešće nije potrebno modifikovati sekvencer jer su funkcionalnosti koje se nalaze u *uvm_ sequencer* klasi obično dovoljne. Kao i drajver, i sekvencer je moguće parametrizovati sa tipom transakcije, npr:

```
class calc_sequencer extends uvm_sequencer#(calc_seq_item);
  // ...
endclass : calc_sequencer
```

Ukoliko, osim parametrizacije, nije potrebno ubacivati dodatni sadržaj u sekvencer, često se samo koristi *typedef* naredba kako bi se izbegao nepotrebni kod, npr:

```
typedef uvm_sequencer#(calc_seq_item) calc_sequencer;
```

1.3 Komunikacija

UVM komponente komuniciraju preko standardnih TLM (engl. *transaction level modeling*) interfejsa koji olakšava povezivanje i ponovno korišćenje. Komponenta može komunicirati preko svog interfejsa sa bilo kojom drugom komponentom koja implementira taj interfejs. U praksi je ovaj način komunikacije veoma čest za drajver - sekvencer interakciju i monitor - *scoreboard* komunikaciju. U ovoj i narednoj vežbi je fokus na komunikaciji između drajvera i sekvencera, dok je detaljan opis rada TLM konekcija ostavljen za vežbu o monitoru.

U *uvm_ driver* i *uvm_ sequencer* klasama postoji deklaracija odgovarajućih portova i ugrađena logika oko komunikacije, tako da je upotreba od strane korisnika veoma jednostavna. Potrebno je

samo povezati drajver i sekvencer, koristeći metodu *connect*. Povezivanje se tipično vrši u *connect* fazi, gde je, na primeru ispod, *driver* instanca prethodno kreiranog drajvera, a *sequencer* instanca prethodno kreiranog sekvencera. *seq_item_port* i *seq_item_export* su imena nasleđenih TLM portova.

```
function void connect_phase(uvm_phase phase);
  drviver.seq_item_port.connect(sequencer.seq_item_export);
endfunction : connect_phase
```

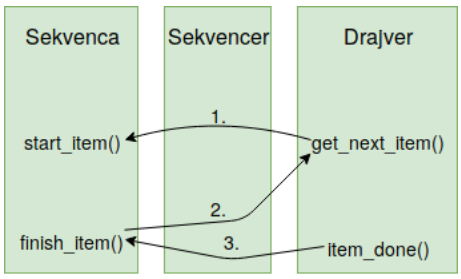
Sam UVM ne pruža samo mogućnost korišćenja bibilioteke, već i definiše način upotrebe kako bi se kreirale univerzalne verifikacione komponente koje će pratiti datu metodologiju. S tim u vezi, postoji jasno definisana komunikacija između drajvera i sekvencera, koristeći određene UVM funkcije, čiji je opis dat u nastavku.

UVM drajver sadrži mnoge metode za komunikaciju sa sekvencerom. Najčešće korišćene su:

- *get_next_item()* - blokirajuća metoda koja traži novi objekat; vraća pokazivač na novi REQ objekat
- *try_next_item()* - neblokirajuća verzija koja vraća *null* pokazivač ukoliko nema dostupnog objekta
- *item_done()* - neblokirajuća metoda koja završava proces rukovanja između drajvera i sekvencera; poziva se nakon *get_next_item()* ili uspešnog *try_next_item()*

Sam sekvencer služi za arbitraciju, dok se *sequence item* objekti kreiraju i kontrolišu iz sekvenci. Da bi se paket poslao drajveru, on mora proći kroz nekoliko koraka:

1. Kreiranje - veoma je bitno kreirati paket koristeći *factory* kako bi se olakšala ponovna upotreba odnosno kako bi se dozvolio *override* metod ukoliko je to potrebno (objašnjeno u narednim vežbama)
2. *Start_item()* - blokirajući poziv koji čeka da se uspostavi veza sa drajverom. Sekvenceru šalje pokazivač na sledeći objekat
3. Randomizacija paketa ukoliko je to potrebno
4. *Finish_item()* - blokirajući poziv koji čeka da drajver završi sa paketom (napomena: *start_item()* i *finish_item()* čekaju na drajver kako bi se osigurao korektan transfer paketa, ali oni ne troše simulaciono vreme)
5. *Get_response()* - opcioni poziv ako je potrebno da drajver pošalje odgovor sekvenceru



Slika 1: Drajver-sekvencer *flow*

Preporučeni *flow* je prikazan na slici 1. Drajver traži transakciju od sekvencera koristeći *get_next_item()*, obrađuje istu i zatim kompletira *handshake* koristeći *item_done()* čime signalizira sekvenceru da je objekat obrađen. Sa druge strane, u sekvenci se kreiraju transakcije prateći prethodne korake. Kada

drajver pozove *item_ done()*, odblokiraće se *finish_ item()* metoda u sekvenci i može se nastaviti sa izvršavanjem.

Pored ovog načina, moguće je ostvariti komunikaciju koristeći *peek()*, *put()* i *get()* metode u drajveru koje su opisane u narednoj vežbi.

2 Sekvence

Sekvence su objekti (ne komponente) koje sadrže logiku generisanja stimulusa. Kao i većina UVM okruženja, i kod sekvenci postoji preporučen način upotrebe, kako u pogledu strukture same sekvence, tako i za sam način generisanja stimulusa. U ovom poglavlju je opisan mehanizam generisanja stimulusa, dat je pregled strukture sekvenci i objašnjene često korišćene funkcije.

2.1 Struktura

Sve sekvence u okruženju treba da, direktno ili indirektno, nasleđuju *uvm_sequence* klasu. Pošto sekvence nisu komponente one ne sadrže UVM faze, međutim ipak postoji sličan mehanizam koji garantuje ispravan rad i ispravnu komunikaciju sa drajverom. Tri najčešće korišćene metode su *pre_body()*, *body()* i *post_body()*. Glavna logika treba da se nalazi u *body()* tasku. *Pre_body()* i *post_body()* su taskovi koji se pozivaju pre, odnosno posle *body()* taska. Kao i kod pisanja testova, dobra je praksa da postoji bazna klasa u kojoj se deklariše sekvencer, vrši eventualno podizanje/spuštanje prigovora (*raise / drop objection*, opisano u narednom poglavlju), kao i sve ostale radnje koje će se ponavljati u svim sekvencama.

```
class calc_seq extends uvm_sequence#(calc_seq_item);

  `uvm_object_utils(calc_seq)
  `uvm_declare_p_sequencer(calc_sequencer)

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

  virtual task pre_body();
    // ...
  endtask : pre_body

  // transaction generating logic in body
  virtual task body();
    // calls to uvm_do or uvm_do with macro
    // or start / finish item
    // ...
  endtask : body

  virtual task post_body();
    // ...
  endtask : post_body

endclass : calc_seq
```

Kao i kod drajvera i sekvencera, potrebno je parametrizovati sekvence tipom *sequence_item*-a koji će se generisati. Ovo, između ostalog, omogućava korišćenje podrazumevanog objekta u sekvencama. Pored *factory* registracije (` *uvm_object_utils(calc_seq)* u primeru), potrebno je i deklarisati sekvencer koji će se koristiti. Svaka sekvenca se pri pokretanju veže za odgovarajući sekvencer kome se može pristupati preko pokazivača. Ovaj korak je moguće i izostaviti, ili deklarisati sekvencere na drugi način, međutim najjednostavniji način za naše potrebe je da se deklariše takozvani *p sekvencer* koristeći odgovarajući makro (` *uvm_declare_p_sequencer(calc_sequencer)*, gde *calc_sequencer* predstavlja ime korišćenog sekvencera).

2.2 Generisanje stimulusa

U prethodnom poglavlju smo videli da je za generisanje transakcije potrebno ispratiti nekoliko koraka:

1. Kreiranje - veoma je bitno kreirati paket koristeći *factory* kako bi se olakšala ponovna upotreba odnosno kako bi se dozvolio *override* metod ukoliko je to potrebno (objašnjeno u narednim vežbama)

2. *Start_ item()* - blokirajući poziv koji čeka da se uspostavi veza sa drajverom. Sekvenceru šalje pokazivač na sledeći objekat
3. Randomizacija paketa ukoliko je to potrebno
4. *Finish_ item()* - blokirajući poziv koji čeka da drajver završi sa paketom (napomena: *start_ item()* i *finish_ item()* čekaju na drajver kako bi se osigurao korektan transfer paketa, ali oni ne troše simulaciono vreme)
5. *Get_ response()* - opcioni poziv ako je potrebno da drajver pošalje odgovor sekvenceru

Prva četiri koraka su obavezna, dok je peti opcioni i koristi se samo ukoliko za dato okruženje postoji potreba prikupljanja odgovora od drajvera. Logika generisanja stimulusa treba da se nalazi u *body()* tasku, npr:

```
virtual task body();

  calc_seq_item calc_it;

  // prvi korak kreiranje transakcije
  calc_it = calc_seq_item::type_id::create("calc_it");

  // drugi korak - start
  start_item(calc_it);

  // treci korak priprema
  // po potrebi moguce prosiriti sa npr. inline ogranicenjima
  assert(calc_it.randomize());

  // cetvrti korak - finish
  finish_item(calc_it);

endtask: body
```

Pošto se ova četiri koraka često ponavljaju tj. standardni su za interakciju drajvera i sekvencera, razvijeni su odgovarajući UVM makroi kao bi se izbeglo nepotrebno ponavljanje koda i povećala čitljivost. Iako veoma korisni, makroi ipak uvode neka ograničenja pri korišćenju, tako da je nekad ipak potrebno ručno odraditi ove korake. "UVM Cookbook" ne preporučuje upotrebu makroa, ali su oni česti u praksi.

2.2.1 ` *uvm_ do*

` *uvm_ do* makro kao argument prima *sequence_ item*. Kada se nasledi i parametrizuje *uvm_ sequence* klasa, nasleđuje se i instanca objekta pod nazivom *req* čiji je tip upravo parametar koji smo prosledili. Što znači da u svakoj sekvenci već postoji objekat *req* koji se može koristiti za komunikaciju. Slično postoji i objekat *rsp* koji treba koristiti za odgovor ukoliko je isti potreban. Upotreba ` *uvm_ do* makroa je:

```
virtual task body();
  `uvm_do(req)
endtask : body
```

Ovim se postiže generisanje i slanje jedne transakcije drajveru, koja će se zatim generisati na interfejsu.

2.2.2 ` *uvm_ do_ with*

Ukoliko je potrebna veća kontrola prilikom slanja transakcije, moguće je koristiti ` *uvm_ do_ with* makro, koji osim *sequence_ item* argumenta prima i bilo koje validno inline ograničenje. Npr:

```
virtual task body();
  `uvm_do_with(req, { req.data == 16'h5A5A; } )
endtask : body
```

U ovom primeru se šalje jedna transakcija pri čemu je vrednost *data* polja ograničena na 16'h5A5A.

2.2.3 Kontrola

U prethodnim primerima smo slali samo jednu transakciju. Često je neophodno omogućiti veću kontrolu nad izvršavanjem sekvenci - omogućiti kontrolisanu randomizaciju, slanje željenog broja transakcija itd. Prilikom pisanja sekvenci moguće je koristiti sve mogućnosti koje SystemVerilog pruža. U nastavku je dato par konstrukcija koje se često sreću i koje daju uvid u ove mogućnosti.

Kontrola prilikom randomizacije je često vrši pomoću *rand* polja u sekvenci, kako bi se omogućila laka kontrola direktno iz testa koji startuje sekvencu. Sledeći primer ilustruje slanje više transakcija.

```
class calc_simple_seq extends calc_base_seq;

  `uvm_object_utils (calc_simple_seq)

  rand int unsigned num_of_tr;

  constraint num_of_tr_con { num_of_tr inside { [1:100] }; }

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

  virtual task body();
    repeat(num_of_tr) begin
      `uvm_do(req);
    end
  endtask : body

endclass : calc_simple_seq
```

Takođe je moguće pozivati jednu sekvencu iz druge, pri čemu je moguće koristiti prethodno opisane makroe i nad sekvencama (ili koristiti metodu *start* opisanu u narednom poglavlju).

```
class calc_simple_seq extends calc_base_seq;

  `uvm_object_utils (calc_simple_seq)

  calc_sub_seq sub_seq;

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

  virtual task body();
    `uvm_do(sub_seq);
  endtask : body

endclass : calc_simple_seq
```

2.3 Pokretanje sekvenci

Sekvence se pokreću ili iz testa ili iz drugih sekvenci. Postoje dva glavna načina pokretanja sekvenci - eksplicitno koristeći *start* metodu ili implicitno kao podrazumevanu sekvencu.

Ukoliko je potrebna veća kontrola pokretanja sekvenci (npr. podešavanje parametara, trenutak startovanja, ...) preporučuje se eksplicitno pokretanje sekvence korišćenjem *start* metode, npr:

```
`ifndef TEST_SIMPLE_SV
`define TEST_SIMPLE_SV

class test_simple extends test_base;
```

```
  `uvm_component_utils(test_simple)

  calc_simple_seq simple_seq;

  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);
    simple_seq = calc_simple_seq::type_id::create("simple_seq");
  endfunction : build_phase

  task main_phase(uvm_phase phase);
    phase.raise_objection(this);
    simple_seq.start(seqr);
    phase.drop_objection(this);
  endtask : main_phase

endclass

`endif
```

Kod 1: *Start*_metoda

Prilikom poziva *start* metode jedini obavezan argument je sekvencer na kome će pokrenuti sekvenca.

Što se tiče podrazumevanih sekvenci, one se uglavnom navode u *build* fazi testa, npr:

```
`ifndef TEST_SIMPLE_2_SV
`define TEST_SIMPLE_2_SV

class test_simple_2 extends test_base;

  `uvm_component_utils(test_simple_2)

  function new(string name = "test_simple_2", uvm_component parent = null);
    super.new(name,parent);
  endfunction : new

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    uvm_config_db#(uvm_object_wrapper)::set(this,
                                            "seqr.main_phase",
                                            "default_sequence",
                                            calc_simple_seq::type_id::get());
  endfunction : build_phase

endclass

`endif
```

Kod 2: Podrazumevane_sekvence

Za navođenje podrazumevane sekvence se koriti UVM *configuration database* koji omogućava postavljanje konfiguracionih parametara u okruženju, u našem slučaju podrazumevane sekvence (*uvm_config_db* će biti detaljnije obrađen u narednim vežbama). U prethodnom primeru je za *main* fazu prethodno instanciranog *seqr* sekvencera podešena *calc_simple_seq* kao podrazumevana sekvenca.

Pokretanje podrazumevanih sekvenci je u UVM-u nasleđeno iz OVM-a i zadržano je radi lakšeg prelaza sa jedne metodologije na drugu. Iako i dalje često korišćeno, preporuka je koristiti *start*

metodu radi bolje kontrole.

3 Mehanizam završetka testa

Svaki testbenč koji koristi standardne UVM faze se sastoji od nekoliko faza za kreiranje i povezivanje koje se izvršavaju u nultom vremenu, zatim određen broj faza koje troše vreme, kao i nekoliko faza za završetak testa koje se izvršavaju u nultom vremenu. Do kraja testa će doći kada se završe sve faze koje troše vreme, a svaka faza se završava kada nema prigovora za tu fazu što znači da se kraj testa kontroliše podizanjem i spuštanjem prigovora. Komponenta ili sekvenca će podići prigovor na početku aktivnosti koja se mora završiti u datoj fazi, a spustiti prigovor po završetku te aktivnosti. Tek kada se svi podignuti prigovori spuste, moguće je završiti datu fazu. U praksi se često dešava da je faza samo trenutno bez prigovora tj. da će se uskoro podići još neki prigovor i da ne treba odmah završiti sa trenutnom fazom. U tom slučaju se može postaviti vreme čekanja (engl. *drain time*) koje unosi kašnjenje za završetak faze. Ako u tom vremenskom intervalu podigne prigovor, faza se neće završiti i opet će se čekati na spuštanje svih prigovora.

Kontrola prigovora se uglavnom vrši ili iz samog testa ili iz sekvenci. Kontrola iz testa se često sreće kada se sekvence eksplicitno pokreću. Jedan primer je prikazan ispod.

```
task main_phase(uvm_phase phase);
  phase.raise_objection(); //rasing objection
  example_seq.start(example_agent.sequencer);
  phase.drop_objection(); //droping objection
endtask
```

Što se tiče kontrole iz sekvenci sekvenci, pošto se čitava logika generisanja stimulusa nalazi u *body()* tasku, prigovor je potrebno podići pre odnosno posle izvršavanja *body()* taska i ovo se najčešće radi u *pre_ body()* odnosno *post_ body()* taskovima. Ovaj kod se najčešće nalazi u baznoj sekvenci kako bi se prigovori generisali prilikom pokretanja svake sekvence. Test se tada neće završiti do god postoje sekvence u okruženju koje nisu završile sa generisanjem stimulusa.

```
// objections are raised in pre_body
virtual task pre_body();
  uvm_phase phase = get_starting_phase();
  if (phase != null)
    phase.raise_objection(this, {"Running sequence '", get_full_name(), "'"});
    uvm_test_done.set_drain_time(this, 200ns)
endtask : pre_body

// objections are dropped in post_body
virtual task post_body();
  uvm_phase phase = get_starting_phase();
  if (phase != null)
    phase.drop_objection(this, {"Completed sequence '", get_full_name(), "'"});
endtask : post_body
```

Jedan način podizanja / spuštanja prigovora je prikazan kodom iznad. Koriste se *raise_ objection/drop_ objection* metode. *starting_ phase* je član bazne klase koji će biti ispravno postavljen jedino ukoliko je sekvenca pokrenuta kao podrazumevana (*default*) za neku fazu.

Pomoću metode *set_ drain_ time* se postavlja takozvano *drain* vreme koje u datom primeru unosi toleranciju od 200 ns, odnosno nakon spuštanja poslednjeg prigovora test se neće odmah završiti nego će se sačekati još 200 ns zbog mogućnosti da se u tom periodu ponovo podigne neki prigovor.

Napomena: pošto je sekvenca ručno pokrenuta, *starting_ phase* polje iz prethodnog kodnog fragmenta bi imalo vrednost *null* tako da se korišćenjem oba kodna fragmenta ispovremeno ne bi podigla dva prigovora za istu sekvencu.

Napomena: podizanje / spuštanje prigovora u sekvenci se često koristi uz UVM 1.1 biblioteku. Za UVM 1.2 preporučen način je podizanje / spuštanje prigovora iz testa.

4 Zadaci

Zadatak Napisati sekvence koje će se koristiti za verifikaciju "Calc1" DUT-a, uključujući:

- Sekvenca za generisanje jedne transakcije na nasumično izabranom portu, sa nasumičnim podacima i komandom
- Sekvenca za generisanje više nasumičnih transakcija na istom portu
- Sekvenca za generisanje više nasumičnih transakcija, pri čemu se broj transakcija nalazi u opsegu (1, 10), a svaka naredna transakcija se nalazi na portu različitom od prethodnog
- Sekvenca za generisanje više nasumičnih transakcija pri čemu su komande uvek za jednu ALU, bez kašnjenja između generisanja transakcija
- smisliti još nekoliko primera sekvenci koji bi mogle biti korisne za verifikaciju

Zadatak Napisati test (uz eventualne pomoćne testove ili sekvence) u kome će se poslati tačno 8 transakcija, koristeći neku od prethodno razvijenih sekvenci. Na koje je načine ovo moguće odraditi?

5 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_SEQUENCER_SV
 `define CALC_SEQUENCER_SV

class calc_sequencer extends uvm_sequencer#(calc_seq_item);

   `uvm_component_utils(calc_sequencer)

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

endclass : calc_sequencer

`endif
```

Kod 4: v6_calc_sequencer

```
`ifndef CALC_DRIVER_SV
 `define CALC_DRIVER_SV

class calc_driver extends uvm_driver#(calc_seq_item);

   `uvm_component_utils(calc_driver)

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

   task main_phase(uvm_phase phase);
      forever begin
         seq_item_port.get_next_item(req);
         `uvm_info(get_type_name(),
```

```
                    $sformatf("Driver sending...\n%s", req.sprint()),
                    UVM_HIGH)
        // do actual driving here
        /* TODO */
        seq_item_port.item_done();
      end
    endtask : main_phase

endclass : calc_driver

`endif
```

Kod 5: v6_calc_driver

```
`ifndef CALC_SEQ_ITEM_SV
 `define CALC_SEQ_ITEM_SV

class calc_seq_item extends uvm_sequence_item;

    /* TODO add fields and methods here */

    `uvm_object_utils_begin(calc_seq_item)
    `uvm_object_utils_end

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

endclass : calc_seq_item

`endif
```

Kod 6: v6_calc_seq_item

```
`ifndef TEST_BASE_SV
 `define TEST_BASE_SV

class test_base extends uvm_test;

    `uvm_component_utils(test_base)

    calc_driver drv;
    calc_sequencer seqr;

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

    function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      drv = calc_driver::type_id::create("drv", this);
      seqr = calc_sequencer::type_id::create("seqr", this);
    endfunction : build_phase

    function void connect_phase(uvm_phase phase);
      drv.seq_item_port.connect(seqr.seq_item_export);
    endfunction : connect_phase

endclass : test_base

`endif
```

Kod 7: v6_test_base

```
`ifndef TEST_SIMPLE_SV
 `define TEST_SIMPLE_SV
```

```
class test_simple extends test_base;

   `uvm_component_utils(test_simple)

   calc_simple_seq simple_seq;

   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);
      simple_seq = calc_simple_seq::type_id::create("simple_seq");
   endfunction : build_phase

   task main_phase(uvm_phase phase);
      phase.raise_objection(this);
      simple_seq.start(seqr);
      phase.drop_objection(this);
   endtask : main_phase

endclass

`endif
```

Kod 8: v6_test_simple

```
`ifndef TEST_SIMPLE_2_SV
`define TEST_SIMPLE_2_SV

class test_simple_2 extends test_base;

   `uvm_component_utils(test_simple_2)

   function new(string name = "test_simple_2", uvm_component parent = null);
      super.new(name,parent);
   endfunction : new

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);

      uvm_config_db#(uvm_object_wrapper)::set(this,
                                              "seqr.main_phase",
                                              "default_sequence",
                                              calc_simple_seq::type_id::get());
   endfunction : build_phase

endclass

`endif
```

Kod 9: v6_test_simple_2

```
`ifndef TEST_LIB_SV
`define TEST_LIB_SV

`include "tests/v6_test_base.sv"
`include "tests/v6_test_simple.sv"
`include "tests/v6_test_simple_2.sv"

`endif
```

Kod 10: v6_test_lib

```
`ifndef CALC_BASE_SEQ_SV
```

```
`define CALC_BASE_SEQ_SV

class calc_base_seq extends uvm_sequence#(calc_seq_item);

  `uvm_object_utils(calc_base_seq)
  `uvm_declare_p_sequencer(calc_sequencer)

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

  // objections are raised in pre_body
  virtual task pre_body();
    uvm_phase phase = get_starting_phase();
    if (phase != null)
      phase.raise_objection(this, {"Running sequence '", get_full_name(), "'"});
  endtask : pre_body

  // objections are dropped in post_body
  virtual task post_body();
    uvm_phase phase = get_starting_phase();
    if (phase != null)
      phase.drop_objection(this, {"Completed sequence '", get_full_name(), "'"});
  endtask : post_body

endclass : calc_base_seq

`endif
```

Kod 11: v6_calc_base_seq

```
`ifndef CALC_SIMPLE_SEQ_SV
`define CALC_SIMPLE_SEQ_SV

class calc_simple_seq extends calc_base_seq;

  `uvm_object_utils (calc_simple_seq)

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

  virtual task body();
    // simple example - just send one item
    `uvm_do(req);
  endtask : body

endclass : calc_simple_seq

`endif
```

Kod 12: v6_calc_simple_seq

```
`ifndef CALC_SEQ_LIB_SV
`define CALC_SEQ_LIB_SV

`include "sequences/v6_calc_base_seq.sv"
`include "sequences/v6_calc_simple_seq.sv"

`endif
```

Kod 13: v6_calc_seq_lib

```
`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 "v6_calc_seq_item.sv"
`include "v6_calc_driver.sv"
`include "v6_calc_sequencer.sv"

`include "sequences/v6_calc_seq_lib.sv"
`include "tests/v6_test_lib.sv"

endpackage : calc_verif_pkg

`include "calc_if.sv"

`endif
```

Kod 14: v6_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 15: v6_calc_verif_top

```
# 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 16: calc_run