# Funkcionalna verifikacija hardvera

## Vežba 8
## Razvoj monitora

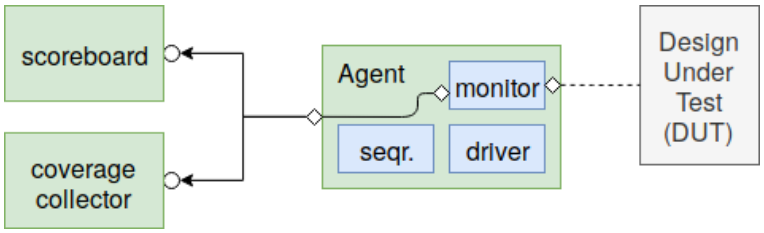
# Sadržaj

Osma vežba je posvećena monitoru. Detaljnije su objašnjene TLM konekcije, dat je opis strukutre i funkcionalnosti UVM monitora i objašnjen preporučen način pisanja čekera.

# 1 TLM

Prilikom razvoja verifikacionog okruženja potrebno je problemu pristupiti na odgovarajućem nivou apstrakcije. Iako je interfejs ka DUT-u reprezentovan na nivou signala, u praksi se pokazalo da je za većinu taskova u verifikaciji (generisanje stimulusa, analiza podataka, prikupljanje pokrivenosti, ...) efikasnije posmatrati problem na nivou transakcija. UVM pruža velik skup interfejsa za komunikaciju na nivou transakcija. Korišćenje ovih TLM (*Transaction Level Modeling*) interfejsa omogućava izolaciju komponenti tako da promene u okruženju ne utiču na datu komponentu. Ovim postupkom se omogućava laka ponovna upotreba, ali i jednostavna zamena komponenti do god sadrže isti interfejs.

TLM-1 i TLM-2.0 su dva TLM sistema modelovanja implementirana po industrijskim standardima. Oba su razvijena u SystemC-u. Deo oba ova sistema je implementiran i u SystemVerilog-u i dostupni su kao deo UVM-a. Mogućnosti ovog sistema su mnogobrojne i za detaljnije informacije pogledati drugo poglavlje "UVM Users Guide"-a, dok ćemo se na ovim vežbama zadržati na metodama i mehanizmu slanja i primanja transakcija između komponenti.




Slika 1: Primer komponenti za analizu

Korišćenje TLM interfejsa je veoma često za analizu tj. u komponentama koje služe za nadgledanje aktivnosti DUT-a i analizu prikupljenih podataka. Te komonente su uglavnom monitori, *scoreboard*-i, eventualni sakupljači pokrivenosti, itd. Neke od ovih komponenti su prikazane na slici 1. Monitor će nadledati same signale preko virtuelnog interfejsa i vršiće konverziju iz nivoa signala u nivo transackcija. Te prikupljene transakcije će tada preko portova za analizu (*analysis port*) slati ostalim komponentama na dalju obradu. U *scoreboard*-u će se prikupljati transackcije i vršiti provera da li DUT funkcioniše u skladu sa funkcionalnom specifikacijom, *coverage collector* će prikupljati podatke o pokrivenosti, itd. Kako bi ovaj mehanizam ispravno funkcionisao potrebno je u svakoj komponenti dobro definisati TLM interfejs i zatim izvršiti njihovo povezivanje. Sledi opis ovog mehanizma.

Napomena: iako je moguće sve prethodne funkcionalnosti obaviti u jednoj komponenti, dobra je praksa da se odvoji na više komponenti kako bi se omogućila lakša ponovna upotreba i veća čitljivost koda.

Česta topologija koja se sreće u praksi je *one-to-many* odnosno iz jednog izvora se šalju podaci na više mesta (monitor ka *scoreboard*-u, *coverage collector*-u itd), pri čemu ne postoji ograničenje za broj primaoca. U UVM-u postoje tri objekta koji se mogu koristiti za ove svrhe *analysis port*, *analysis export* i *analysis fifo*.

*Analyisis port* se koristi kako bi se obavio neblokirajući *broadcast* transakcija. Za svaki port se može vezati više komponenti koje će primati poruke, ali je moguće i da nijedna komponenta ne bude povezana. *uvm_analysis_port* sadrži jednu funkciju, *write()*, čija se implementacija nalazi u komponenti koja prima transakcije. Pozivom odgovarajuće *write()* funkcije se vrši slanje transakcija. U nastavku je data sintaksa deklarisanja i primer korišćenja:

```
class monitor extends uvm_monitor;
```

```
  uvm_analysis_port#(trans) ap;

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

  task main_phase(uvm_phase phase);
    trans t;
    // ...
    ap.write(t);
  endfunction

endclass
```

Sintaksa za deklarisanje analysis porta je: *uvm_analysis_port #(<tip>) <ime>;* Zatim je potrebno kreirati port u konstruktoru. Slanje podatka svim povezanim interfejsima vrši se pozivom funkcije *write* čiji je prototip: *function void write (input T t).* Sama implementacija ove funkcije nalazi se u komponenti koja treba da prima transakcije odnosno sadrži *uvm_analysis_imp*. Primer ovakve komponente je dat u nastavku:

```
class scbd extends uvm_scoreboard;

  uvm_analysis_imp#(trans, scbd) ap;

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

  function void write(trans t);
    // ...
  endfunction

endclass
```

Svaki put kada se u *mon* klasi želimo da pošaljemo transakciju *sb* komponenti pozove se funkcija *write*. Međutim, postoje određena pravila za implementaciju ove funkcije. Pošto je u pitanju funkcija, a ne task ne sme se trošiti simulaciono vreme. Takođe, nije dozvoljeno modifikovanje prosleđene vrednosti odnosno objekta *t*.

Povezivanje *uvm_analysis_port*-a i *uvm_analysis_imp*-a se uglavnom vrši u komponentama na višem nivou hijerarhije. Na primer, u *connect* fazi agenta ili okruženja. Kako bi se izvršilo povezivanje, potrebno je pozvati metodu *connect*, npr (*mon* i *sb* su imena instanci):

```
mon.ap.connect(sb.ap);
```

Pozivom ove funkcije smo povezali i drajver i sekvencer na prethodnim vežbama. Ukoliko je potrebno povezati više koponenti na isti port, sintaksa ostaje ista, pri čemu treba napomenuti da su sve komponente koje se povezuju na ovaj port nezavisne odnosno svaka mora sadržati implementaciju svoje *write* funkcije. Npr.

```
mon.ap.connect(sb1.ap);
mon.ap.connect(sb2.ap);
mon.ap.connect(sb3.ap);
```

Česta situacija je i da jedna komponenta treba da prima transakcije iz više izvora. Npr. *scoreboard* koji će primati transakcije iz više monitora u okruženju i vršiti poređenje. U ovom slučaju je potrebno koristiti posebne makroe za *uvm_analysis_imp* kako bi se jasno definisalo koja *write* funkcija se odnosi na koji port.

U primeru ispod je data implementacija *scoreboard* komponente koja sadrži dva *analysis_imp*-a.

Koristi ` *uvm_analysis_imp_decl(<ime>)* makro kako bi se deklarisala *uvm_analysis_imp<ime>* klasa. Ovim se zatim kreira *write<ime >()* funkcija koju zatim možemo implementirati u skladu sa datim potrebama.

Napomena: Imena korišćena u primeru (*_in* i *_out*) su proizvoljna, ali je dobra praksa da započinju znakom "_" pošto tada rezultujuće *write* funkcije imaju jasna imena (*write_in* i *write_out*).

```
`uvm_analysis_imp_decl(_in)
`uvm_analysis_imp_decl(_out)

class scbd extends uvm_scoreboard;

  uvm_analysis_imp_in#(trans, scbd) port_in;
  uvm_analysis_imp_out#(trans, scbd) port_out;

  function new(string name = "scbd", uvm_component parent = null);
    super.new(name, parent);
    port_in = new("port_in", this);
    port_out = new("port_out", this);
  endfunction

  function void write_in(trans t);
    // ...
  endfunction

  function void write_out(trans t);
    // ...
  endfunction

endclass
```

Povezivanje ove komponente bi se vršilo na isti način, tj:

```
mon1.ap.connect(sb.port_in);
mon2.ap.connect(sb.port_out);
```

# 2 Monitor

Monitor komponenta je zadužena za izvlačenje informacija iz signala i prevođenje u viši nivo apstrakcije bilo u vidu transakcija, događaja ili nekih statusnih informacija. Ove informacije treba da su dostupne ostatku okruženja preko TLM interfejsa. Monitor je uvek pasivna komponenta zadužena za nadgledanje signala, a ne i njihovo generisanje. Iako se deo funkcionalnosti monitora često preklapa sa funkcionalnošću drugih komponenti (uglavnom drajvera), monitor mora biti nezavisna komponenta. Funkcionalnost monitora treba ograničiti na osnovno nadgledanje koje će uvek biti potrebno, ali koje može biti i lako kontrolisano npr. provere protokola ili sakupljanje pokrivenosti. Sve naprednije funkcionalnosti (npr. provere vezane ne samo za protokol, već za naprednu funkcionalnost DUT-a) treba implementirati u odvojenim komponentama - *scoreboard*-u, prikupljaču pokrivenosti, globalnim monitorima i sl. Takođe se često odvaja i izvlačenje informacija iz signala od aktivnosti nad transakcijama. Za sve komunikacije između pod-komponenti treba koristiti TLM portove.

## 2.1 Struktura

U najosnovnijem obliku, monitor će sadržati virtualni interfejs preko koga pristupa signalima, *analysis port* za slanje prikupljenih transakcija i *factory* registraciju. Kostur UVM monitora je dat ispod. U *connect* fazi se vrši preuzimanje interfejsa in baze, dok se u *run* ili *main* fazi vrši prikupljanje podataka i slanje preko TLM porta.

```
class calc_monitor extends uvm_monitor;

  // control fileds
  bit checks_enable = 1;
  bit coverage_enable = 1;

  uvm_analysis_port #(calc_seq_item) item_collected_port;

  `uvm_component_utils_begin(calc_monitor)
    `uvm_field_int(checks_enable, UVM_DEFAULT)
    `uvm_field_int(coverage_enable, UVM_DEFAULT)
  `uvm_component_utils_end

  // The virtual interface used to drive and view HDL signals.
  virtual interface calc_if vif;

  // current transaction
  calc_seq_item curr_it;

  // coverage can go here
  // ...

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

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if (!uvm_config_db#(virtual calc_if)::get(this, "*", "calc_if", vif))
      `uvm_fatal("NOVIF",{"virtual interface must be set:",get_full_name(),".vif"})
  endfunction : connect_phase

  task main_phase(uvm_phase phase);
    // forever begin
    // curr_it = calc_seq_item::type_id::create("curr_it", this);
    // ...
    // collect transactions
    // ...
    // item_collected_port.write(curr_it);
    // end
```

```
  endtask : main_phase

endclass : calc_monitor
```

Kod 1: Kostur monitora

Česta, ali ne i obavezna, praksa je da monitor sadrži kontrolna polja (u primeru *checks_enable* i *coverage_enable*) koja služe za kontorlu obavljanja čekiranja i prikupljanja pokrivenosti. Ove operacije mogu veoma uticati na brzinu simulacije, pa je potrebno obezbediti način da se one, po potrebi, isključe.

```
if (checks_enable)
  perform_transfer_checks();
if (coverage_enable)
  perform_transfer_coverage();
```

Kontrolu je tada lako vršiti koristeći mehanizam konfiguracija, npr:

```
uvm_config_db#(int)::set(this,"*.monitor", "checks_enable", 0);
```

## 2.2 Funkcionalnost

Glavna razlika između drajvera i monitora je što je monitor pasivna komponenta. Ovo znači da je monitor zadužen samo za nadgledanje signala, a ne i njihovo generisanje. Kako bi ispravno prepoznao aktivnosti na signalima, monitor mora poznavati protokol koji se koristi. Praćenje određenog protokola se često implementira kao mašina stanja u *run/main* fazi. Čeka se na određene događaje odnosno prate se stanja signala preko virtuelnog interfejsa. Nakon što se uoči šablon datog protokola, potrebno je kreirati objekat koji predstavlja transakciju i dodeliti mu odgovarajuće vrednosti (npr. trenutna operacija, adresa, vrednost podataka, ...). Nakon što se objekat uspešno kreira, šalje se svim ostalim komponentama pomoću TLM interfejsa. Nadgledanje signala se obično vrši u beskonačnoj petlji u *run/main* fazi. Prilikom svakog prolaska kroz petlju, šalje se uočena transakcija. Pošto objektu pristupamo preko pokazivača, problem koji se može javiti je da se vrednosti prebrišu u narednoj iteraciji. Ovo je moguće izbeći na dva načina:

- Kreiranje novog objekta u svakoj iteraciji petlje
- Koristiti isti objekat, ali izvršiti kloniranje pre poziva *write* funkcije i poslati klon

U nastavku je dat primer *main* faze APB monitora:

```
task main_phase(uvm_phase phase);

  apb_transfer trans_collected, trans_clone;
  trans_collected = apb_transfer::type_id::create("trans_collected");

  forever begin

    @(posedge vif.pclock iff (vif.psel != 0));
    trans_collected.addr = vif.paddr;
    case (vif.prwd)
      1'b0 : trans_collected.direction = APB_READ;
      1'b1 : trans_collected.direction = APB_WRITE;
    endcase

    @(posedge vif.pclock);
    if(trans_collected.direction == APB_READ)
      trans_collected.data = vif.prdata;
    if (trans_collected.direction == APB_WRITE)
      trans_collected.data = vif.pwdata;

    @(posedge vif.pclock);
    if(trans_collected.direction == APB_READ) begin
```

```
            if(vif.pready != 1'b1)
              @(posedge vif.pclock);
            trans_collected.data = vif.prdata;
        end

        $cast(trans_clone, trans_collected.clone());
        item_collected_port.write(trans_clone);
    end

endtask
```

Kod 2: Main faza APB monitora

## 2.3 Implementacija čekera

Dve glavne funkcionalnosti monitora, pored sakupljanja transakcija, su vršenje provera i prikupljanje podataka o pokrivenosti. Prikupljanju pokrivenosti (*coverage*) će biti posvećena posebna vežba, dok ćemo se u ovoj vežbi zadržati na preporučenom načinu implementacije čekera.

### 2.3.1 *Assert* naredbe

*Assert* naredbe se primarno koriste za proveru funkcionalnosti dizajna, ali i za proveru samog verifikacionog okruženja (npr. provera uspešnosti randomizacije). U SystemVerilog-u postoje dve vrste ovih tvrdnji:

- trenutne (*immediate*): proceduralne naredbe koje se uglavnom koriste tokom simulacije; tvrdnja koja govori da neki izraz mora biti tačan, nalik if naredbi; veoma liče na *assert* naredbe u VHDL-u
- konkurentne (*concurrent*): naredbe koje govore da neke osobine dizajna moraju biti ispunjene (npr. *read* i *write* signali ne smeju biti aktivni u isto vreme ili posle svakog *request*-a sledi *acknowledge*).

Pisanje *concurrent assertion*-a izlazi iz opsega ovog kursa, međutim za implementaciju čekera je preporučeno koristiti *immediate assertion*, što je opisano u nastavku.

U monitoru, čekere je moguće pisati koristeći običan proceduralni kod ili trenutne tvrdnje. U primerima ispod proverava se da li je vrednost A jednaka vrednosti B. Razlika je sledeća: korišćenjem if naredbe se samo proverava da li je A == B, a sve dalje akcije je potrebno posebno definisati, dok druga naredba tvrdi da je A == B i vratiće grešku ukoliko ovo nije ispunjeno.

```
if (A == B) // ...
assert (A == B);
```

Prednosti korišćenja tvrdnji su mnogobrojne. Ovim naredbama se povećava čitljivost koda, ali i smanjuje vreme potrebno za *debug* jer je greške lakše izolovati i brže uočiti. Simulatori pružaju mogućnosti nadgledanja broja tvrdnji koje prolaze ili ne, pauziranje simulacije ukoliko se pronađe greška, ... Takođe pružaju veliku mogućnost kontrole tokom samog testa npr. moguće je uključiti ili isključiti ove provere. Moguća je i interakcija sa C funkcijama. Prednosti se takođe ogledaju u pogledu dokumentacije jer olakšavaju opis čekera i dizajn specifikacije.

Sintaksa immediate tvrdnje je sledeća:

```
assertion_label : assert (expression)
  // pass block code
else
  // fail block code
```

gde je labela opciona, kao i *pass* i *fail* blokovi koda. Dobra praksa je da se u *fail* bloku ispiše poruka koja objašnjava zašto je došlo do greške. Takođe je korisno da se koristi labela kako bi se lakše pratile sve tvrdnje u okruženju (često ima prefiks ili sufiks *asrt*). Primer je dat ispod:

```
asrt_a_eq_b : assert (A == B)
  `uvm_info(get_type_name(), "Check succesfull: A == B", UVM_HIGH)
else
  `uvm_error(get_type_name(), $sformatf("Observed A and B mismatch: A = %0d, B = %0d", A, B))
```

# 3 Zadaci

**Zadatak** Implementirati monitor za “Calc1” dizajn.

# 4 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: v8_calc_sequencer

```
`ifndef CALC_DRIVER_SV
 `define CALC_DRIVER_SV

class calc_driver extends uvm_driver#(calc_seq_item);

   `uvm_component_utils(calc_driver)

   // The virtual interface used to drive and view HDL signals.
   virtual interface calc_if vif;

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

   function void connect_phase(uvm_phase phase);
```

```
    super.connect_phase(phase);
    if (!uvm_config_db#(virtual calc_if)::get(this, "*", "calc_if", vif))
      `uvm_fatal("NOVIF",{"virtual interface must be set for: ",get_full_name(),".vif"})
  endfunction : connect_phase

  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: v8_calc_driver

```
class calc_monitor extends uvm_monitor;

  // control fileds
  bit checks_enable = 1;
  bit coverage_enable = 1;

  uvm_analysis_port #(calc_seq_item) item_collected_port;

  `uvm_component_utils_begin(calc_monitor)
    `uvm_field_int(checks_enable, UVM_DEFAULT)
    `uvm_field_int(coverage_enable, UVM_DEFAULT)
  `uvm_component_utils_end

  // The virtual interface used to drive and view HDL signals.
  virtual interface calc_if vif;

  // current transaction
  calc_seq_item curr_it;

  // coverage can go here
  //  ...

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

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if (!uvm_config_db#(virtual calc_if)::get(this, "*", "calc_if", vif))
      `uvm_fatal("NOVIF",{"virtual interface must be set:",get_full_name(),".vif"})
  endfunction : connect_phase

  task main_phase(uvm_phase phase);
    // forever begin
    // curr_it = calc_seq_item::type_id::create("curr_it", this);
    //  ...
    // collect transactions
    //  ...
    // item_collected_port.write(curr_it);
    // end
  endtask : main_phase
```

```
endclass : calc_monitor
```

Kod 6: v8_calc_monitor

```
`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 7: v8_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;
   calc_monitor mon;

   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);
      mon = calc_monitor::type_id::create("mon", 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 8: v8_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 9: v8_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 10: v8_test_simple_2

```
`ifndef TEST_LIB_SV
`define TEST_LIB_SV

`include "tests/v8_test_base.sv"
`include "tests/v8_test_simple.sv"
`include "tests/v8_test_simple_2.sv"

`endif
```

Kod 11: v8_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 12: v8_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 13: v8_calc_simple_seq

```
`ifndef CALC_SEQ_LIB_SV
`define CALC_SEQ_LIB_SV

`include "sequences/v8_calc_base_seq.sv"
`include "sequences/v8_calc_simple_seq.sv"

`endif
```

Kod 14: v8_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 "v8_calc_seq_item.sv"
`include "v8_calc_driver.sv"
`include "v8_calc_sequencer.sv"
`include "v8_calc_monitor.sv"
```

```
`include "sequences/v8_calc_seq_lib.sv"
`include "tests/v8_test_lib.sv"

endpackage : calc_verif_pkg

`include "calc_if.sv"

`endif
```

Kod 15: v8_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 )
                );

   initial begin
      uvm_config_db#(virtual calc_if)::set(null, "*", "calc_if", calc_vif);
      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 16: v8_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 17: calc_run