# Funkcionalna verifikacija hardvera

## Vežba 7
## Razvoj drajvera

# Sadržaj

Sedma vežba je posvećena implementaciji drajvera. Dat je pregled mogućih načina komunikacije sa sekvencerom, klasična strukutra drajvera i opisani česti modeli koji se sreću u praksi. Pored opisa drajvera dat je i opis konfiguracione baze podataka i virtuelnih interfejsa.

# 1 Virtuelni interfejs

Pošto je dizajn koji se testira statička instanca, a testbenč dinamička oni se ne mogu direktno povezati. Za povezivanje se koristi virtualni interfejs, pri čemu je dizajn povezan na instancu ovog interfejsa u glavnom modulu. Ovo omogućava drajveru i monitoru direktan pristup signalima dizajna koji se verifikuje na sledeći način: povezivanje testbenča i dizajna se mora uraditi bez dodatnih modifikacija u samom dizajnu, u vidu dodatnih konekcija ili nekih drugih modifikacija, a zatim se informacije o konekciji prosleđuju svim agentima kojima je potreban pristup interfejsu. Povezivanje i prosleđivanje informacija se dešava samo u glavnom modulu kako bi se lakše reagovalo na promene u dizajnu i poboljšala stabilnost i ponovna upotreba testbenča.

Interfejs se definiše u posebnoj SystemVerilog-ovoj konstrukciji istoimenog naziva. U ovom bloku se, pored samih deklaracija portova, nalaze i sve provere vezane za same signale. Te provere su implementirane kao tvrđenja (engl. *assertions*) i mogu biti raznolike: provere da pojedini signali nikad nemaju nepoznatu vrednost, da su stablini u nekom intervalu, da imaju odgovarajuće vrednosti posle reseta, ... Pisanje tvrđenja izlazi iz okvira ovog kursa. Za dodatne informacije pogledati šesnaesto poglavlje SystemVerilog standarda.

Interfejsi se prosleđuju komponentama koristeći UVM-ov mehanizam za konfiguracije opisan u narednom poglavlju.

# 2 Konfiguracioni *database*

Kao što je ranije navedeno, verifikacione komponente se kreiraju imajući u vidu ponovnu upotrebu i konfigurabilnost. Mogu sadržati mnoge funkcionalnosti koje nisu potrebne za svaki projekat. Pomoću UVM-ovog mehanizma za konfiguracije, može se lako definisati potrebna konfiguracija koja će se koristiti u datom projektu. U tipičnom testbenču postoji nekoliko konfiguracionih parametara koji su vezani za neke komponente. UVM vodi računa i o opsegu za koji se definišu vrednosti konfiguracionih parametara, tako da je moguće setovati vrednost samo za neke komponente, a ne za sve koje koriste taj objekat. Na primer da li pojedine komponente treba da vrše proveru ispravnosti ili da sakupljaju podatke o pokrivenosti. Setovanje ovih parametara se obično vrši u samom testu, ali je moguće definisati podrazumevane vrednosti za sve parametre ukoliko se eksplicitno ne odredi njihova vrednost. Moguće je koristiti i konfiguracione klase koje olakšavaju randomizaciju konfiguracionih atributa i rešavanje zavisnosti među atributima. Olakšava se i dodavanje novih atributa ili zavisnosti tako što se klasa nasledi i dodaju novi parametri. Željena konfiguracija se bira samo u testu, a zatim se, pomocu UVM-ovog mehanizma, prosledi željenim komponentama.

*uvm_ config_ db* je UVM-ov konfiguracioni mehanizam baziran na tipovima koji nudi mogućnost hijerarhijskog specificiranja konfiguracionih parametara na željene vrednosti. *uvm_ config_ db* može podešavati skalarne objekte, pokazivače na klase, redove, nizove, čak i virtuelne interfejse. Metode za setovanje odnosno preuzimanje vrednosti su *uvm_ config_ db::set* i *uvm_ config_ db::get*. Njihovi prototipi su:

```
void uvm_config_db#(type T = int)::set(uvm_component cntxt, string inst_name, string field_name, T value);
bit uvm_config_db#(type T=int)::get(uvm_component cntxt, string inst_name, string field_name, ref T value);
```

gde je:

- T - tip podatka koji se prosleđuje (skalarni objekat, red, klasa, virtuelni interfejs, ...)
- Cntxt - hijerarhijska putanja za koju se podešava / preuzima vrednost
- Inst_name - hijerarhijska putanja koja limitira pristup bazi
- Field_name - labela preko koje se traži unos
- Value - vrednost koja će se upisati u bazu ili pročitati iz baze

*Get* metoda vraća 1 ukoliko je vrednost uspešno preuzeta iz baze, a 0 u suprotnom.

Na primer, u UVM se virtuelni interfejs uvek prosleđuje koristeći ovaj mehanizam. U top modulu se instancira i poveže interfejs sa dizajnom koji se verifikuje, a zatim se prosledi ostalim podkomponentama koristeći *uvm_ connfig_ db*. Primer ispod demonstrira setovanje instance *calc_ vif* u bazi uz labelu *calc_ if* za globalni opseg. Prikazan je jedino deo koda koji demonstrira *config_ db*.

```
module calc_verif_top;

  // ...

  // 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

    // ...

endmodule : calc_verif_top
```

Kod 1: Primer setovanja virtualnog interfejsa

U svim komponentama u kojima želimo da koristimo taj interfejs (npr. drajver, monitor, ...), preuzimanje iz baze bi se moglo obaviti nalik:

```
virtual  interface  calc_if  vif;

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("NO_IF",{"virtual interface must be set for: ", get_full_name(), ".vif"})
endfunction : connect_phase
```

U primeru *vif* preuzima vrednost iz baze koristeći labelu *calc_if*. Takođe se, kao dobra praksa, vrši provera uspešnosti dodele i javlja *fatal* i zaustavlja simulacija ukoliko preuzimanje nije uspešno.

U nastavku je dato još par primera korišćenja konfiguracione baze:

```
uvm_config_db#(int)::set(this,"*.masters[0]", "master_id", 0);
uvm_config_db#(uvm_object_wrapper)::set(this,
                                        "*.ubus0.masters[0].sequencer.main_phase",
                                        "default_sequence",
                                        read_modify_write_seq::type_id::get());
```

Prvi primer postavlja int vrednost za *master_id* polje svih master komponenti u okruženju čije se ime instance završava sa *masters[0]*. U drugom primeru se govori *masters[0].sequencer*-u da izvrši sekvencu tipa *read_modify_write_seq* kada uđe u *main* fazu.

U i jednoj od narednih vežbi ćemo se takođe povetiti korišćenju konfiguracija i uočiti prednosti korišćenja ovog mehanizma.

Ipak postoje neki parametri koji moraju biti poznati u vreme kompajliranja, na primer širina magistrale. Ovakvi parametri se ne mogu implementirati kao konfiguracije, već se moraju proglasiti za klasične parametre koristeći ključnu reč *parameter*. Na primer u interfejsu definišemo širinu magistrale za podatke:

```
parameter DATA_WIDTH = 32;
```

# 3 Drajver

U ovom poglavlju je dat opis drajvera u UVM-u. Dat je pregled klasične strukture i sadržaja, objašnjen API i opisani česti načini korišćenja.

## 3.1 Struktura

Kao i sve komponente unutar UVM verifikacionog okruženja, i drajver se implementira tako što se nasledi *uvm_driver* klasa, parametrizovana tipom *sequence item*-a koji će se koristiti za zahtev odnosno odgovor. Pored ovog drajver treba da sadrži kod za *factory* registraciju, konstruktor, kao i virtuelni interfejs preko koga će generisati signale. Kostur većine drajvera je dat ispod.

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

  `uvm_component_utils(example_driver)

  virtual interface example_if vif;

  function new(string name = "example_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 example_if)::get(this, "*", "example_if", vif))
      `uvm_fatal("NO_IF",{"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);
      drive_tr();
      seq_item_port.item_done();
    end
  endtask : main_phase

  task drive_tr();
    // do actual driving here
    // ...
  endtask : drive_tr

endclass : example_driver
```

Nasleđivanjem *uvm_driver* klase, nasleđuje se i TLM port za komunikaciju sa sekvencerom (*seq_item_port*), ali i *req* i *rsp* objekti koji se mogu koristiti za generisanje stimulusa. Ovi objekti će poprimiti tip kojim je parametrizovan drajver (*example_seq_item* u primeru).

## 3.2 API

API za komunikaciju sa sekvencerom je:

- *get_next_item* - blokirajuća metoda koja čeka da REQ transakcija postane dostupna i vraća pokazivač na taj objekat; nakon izvršavanja ove metode obavezno je pozvati *item_done* pre sledećeg poziva *get_next_item*
- *try_next_item* - neblokirajuća metoda koja proverava da li postoji transakcija. Vraća *null* ukoliko nema dostupne transakcije u sekvenceru
- *item_done* - neblokirajuća metoda koja finalizira *handshake* sa sekvencerom; treba je pozvati nakon *get_next_item* ili uspešnog *try_next_item*; moguće joj je proslediti RSP transakciju kao argument

- *peek* - blokirajuća metoda koja čeka na dostupnu REQ transakciju od sekvencera i vraća pokazivač na taj objekat; naredni pozivi ka *peek* pre *get* ili *item_done* poziva će vratiti pokazivač na isti REQ objekat
- *get* - blokirajuća metoda koja čeka na dostupnu REQ transakciju od sekvencera; kada primi transakciju kompletira *handshake* i vraća pokazivač na REQ objekat
- *put* - neblokirajuća metoda koja vraća RSP transakciju kao odgovor sekvenceru; može se pozvati u bilo kom trenutku i nije deo *handshake*-a između drajvera i sekvencera

Na osnovu ovih metoda, postoje dva načina da se ostvari komunikacija sa sekvencerom - koristeći *get_next_item / item_done* ili *get / put* mehanizam.

### 3.2.1 *Get_next_item / item_done*

Ovaj mehanizam omogućava drajveru da uzme transakciju od sekvencera iz sekvenci, da je procesira i zatim kompletira rukovanje koristeći *item_done*. Nije preporučeno proleđivanje argumenata uz *item_done*. Ovaj način komunikacije je preferiran i čest u praksi jer obezbeđuje jasno odvajanje između drajvera i sekvence.

Odgovarajuća implementacija sekvence bi sadržala *start_item / finish_item* pozive, objašnjene na prethodnim vežbama. Pošto i drajver i sekvence imaju pokazivač na isti objekat, bilo koje promene unutar drajvera će biti vidljive i u sekvenci. Odnosno kada se pokazivač na transakciju prosledi *finish_item* pozivu, *get_next_item* u drajveru će vratiti pokazivač na isti objekat. Kada se u sekvenci odblokira *finish_item* (pozivom *item_done* u drajveru), u sekvenci se može pristupati objektu i videti sve promene koje su načinjene u drajveru.

```
// main faza u drajveru
task main_phase( uvm_phase phase );
  forever begin
    seq_item_port.get_next_item(req); // blokirajuci poziv koji vraca sl. transackiju
    // ... drajvovanje ...
    // na primer
    @(posedge vif.clk);
    vif.addr = req.address; // vif je virtualni interfejs
    // itd
    if(req_item.read_or_write == READ) begin // postaviti polja za odgovor
      req.rdata = vif.rdata;
    end
    seq_item_port.item_done(); // javiti sekvenceru da je zavrsena transakcija
  end
endtask : main_phase
```

### 3.2.2 *Get / put*

Koristeći ovaj model, u drajveru se poziva *get* metoda koja preuzima sledeću transakciju i odmah završava rukovanje (pre nego što drajver može procesirati transakciju). U drajveru se zatim poziva *put* metoda kako bi se označilo da je završeno sa datom transakcijom i da se vrati odgovor.Tada je u sekvenci potrebno pozvati *get_responese* nakon *finish_item* koji će blokirati do god se u drajveru ne pozove *put*. Mana ovog modela je što će implementacija drajvera biti komplikovanija i što se u sekvenci mora posebno procesirati odgovor. Takođe problemi mogu nastati ukoliko se koristi više sekvenci, a transackije nisu ispravno označene. Više o ovoj problematici se može naći u "UVM Cookbook"-u, u poglavlju o sekvencama.

```
// main faza u drajveru
task main_phase( uvm_phase phase );
  forever begin

    // uzima se sl. transakcija iz sekvencera
    seq_item_port.get(req);
```

```
    // ... drajvovanje ...
    // na primer
    @(posedge vif.clk);
    vif.addr = req.address; // vif je virtualni interfejs
    // itd
    if(req_item.read_or_write == READ) begin // postaviti polja za odgovor
      req.rdata = vif.rdata;
    end

    // odgovor se vraca sekvenceru
    seq_item_port.put(req);

  end
endtask : main_phase

// body task u sekvenci
virtual task body();
  example_it = example_seq_item::type_id::create("example_it");

  start_item(example_it);
  example_it.randomize();
  finish_item(example_it);

  get_response(tx); // blokirajuci poziv koji ceka da drajver vrati odgovor
endtask : body
```

## 3.3 *Use* modeli

Ispravno generisanje stimulusa u UVM-u zavisi od dobre veze između sekvenci i drajvera. Sekvence i drajver se moraju usaglasiti kako ne bi došlo do *deadlock* situacija kada se jedna strana zauvek blokira čekajući na odgovor. Kako bi se izbegle ovakve situacije i omogućila laka upotreba, ako i ponovna upotreba drajvera potrebno je dobro izmodelovati i dokumentovati funkcionalnost drajvera. Postoji veliki broj modela za generisanje stimulusa, međutim neki od najčešće korišćenjih modela za drajvere su:

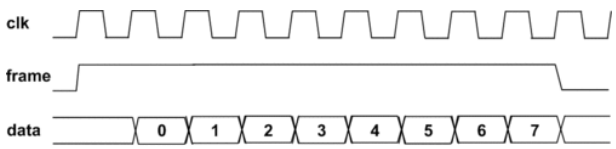
- *Unidirectional Non-Pipelined* - zahtevi se šalju drajveru, ali se ne primaju odgovori od drajvera; drajver može koristiti neki vid rukovanja, ali je proces slanja podataka uvek u jednom smeru; ovaj model se koristi za jednostrane komunikacione kanale nalik PCM interfejsu
- *Bidirectional Non-Pipelined* - podaci se šalju u oba smera, sekvencer šalje transakcije drajveru, dok drajver šalje odgovor; uvek je samo jedna transakcija aktivna, odgovor se uvek šalje pre slanja sledeće transakcije; ovaj model se koristi za jednostavnije protokole npr. APB
- *Pipelined* - podaci se šalju u oba smera, ali se faza slanja zahteva preklapa sa fazom slanja odgovora na prethodni zahtev; protočna obrada omogućava bolje performanse, ali komplikuje sam kod jer se zahtevi i odgovori moraju posebno obrađivati; primer upotrebe bio bi za AHB protokol
- *Out Of Order Pipelined* - u nekim primenama odgovori se ne šalju u redosledu u kom su primeljni zahtevi; ovo rezultuje u dosta komplikovanijem modelu gde se uglavnom koriste redovi za praćenje odgovora; još jedna varijacija bila bi *burst* transferi za više transakcija istovremeno; primer upotebe npr. AXI

U nastavku je su data dva primera modela drajvera. Za kompletan opis svih modela kao i alternativne implementacije datih modela pogledati "UVM Cookbook", poglavlje "*Driver/Use Models*".

### 3.3.1 *Unidirectional Non-Pipelined*

Kod ovog modela drajver kontroliše flow koristeći *get_next_item* poziv kako bi primio sledeću transakciju, a poziv *item_done* metode se vrši tek kada je završeno sa procesiranjem transakcije.

U nastavku je dat korišćenja ovog modela i odgovarajuće sekvence. U primeru se šalju ADPCM paketi koristeći PCM protokol.




Slika 1: Protokol

```
class unidir_driver extends uvm_driver #(unidir_seq_item);

    `uvm_component_utils(unidir_driver)

    virtual unidir_if vif;

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

    task main_phase(uvm_phase phase);
        int top_idx = 0;
        // pocetno stanje
        vif.frame <= 0;
        vif.data <= 0;

        forever begin
            seq_item_port.get_next_item(req); // zahtev za transakcijom
            repeat(req.delay) begin // kasnjenje izmedju paketa
                @(posedge vif.clk);
            end

            vif.frame <= 1; // pocetak slanja
            for(int i = 0; i < 8; i++) begin // slanje podataka
                @(posedge vif.clk);
                vif.data <= req.data[3:0];
                req.data = req.data >> 4;
            end

            vif.frame <= 0; // kraj slanja
            seq_item_port.item_done(); // transakcija je zavrsena
        end
    endtask: run

endclass: unidir_driver


// generisanje transakcija u petlji
class unidir_tx_seq extends uvm_sequence #(unidir_seq_item);

    `uvm_object_utils(unidir_tx_seq)

    // unidir sequence_item
    unidir_seq_item req;

    // kontrola za broj transakcija
    rand int no_reqs = 10;

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

    task body();
        req = unidir_seq_item::type_id::create("req");
```

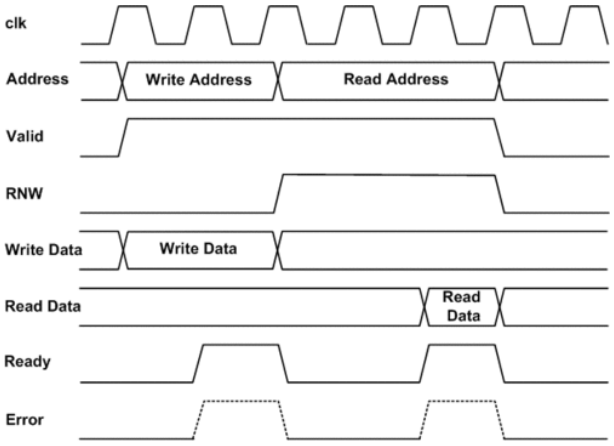
```
    repeat(no_reqs) begin
      start_item(req);
      assert(req.randomize());
      finish_item(req);
    end
  endtask: body

endclass: unidir_tx_seq
```

Kod 2: Primer *use* modela

### 3.3.2 *Bidirectional Non-Pipelined*

Najčešće korišćen model je upravo bidirekcioni. Sekvencer šalje zahteve drajveru koji ih izvršava, a zatim šalje odgovor nazad. Nova transakcija ne može biti započeta do god se ne primi odgovor i ne završi prethodna. Ovi modeli se koriste za jednostavnije protokole kao što je npr. AMPA APB.

Drajver prima transakciju, po protokolu koji se implementira pošalje podatke uz eventualno čekanje i zatim postavi polja u transakciji koja služe za odgovor i završi rukovanje sa sekvencerom. Pošto se i u drajveru i u sekvenci koristi pokazivač na isti objekat, nakon što se sekvenca odblokira (posle *finish_item*) može koristiti objekat sa novim vrednostima i vršiti dalju analizu. Primer je dat ispod:




Slika 2: Protokol

```
class bidir_driver extends uvm_driver #(bidir_seq_item);

  `uvm_component_utils(bidir_driver)

  virtual bidir_if vif;

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

  task main_phase(uvm_phase phase);
    // pocetno stanje
    vif.valid <= 0;
    vif.rnw <= 1;
    // cekanje da se reset zavrsi
    @(posedge vif.resetn);

    forever begin
      seq_item_port.get_next_item(req); // zahtev za transakcijom
```

```
        repeat(req.delay) begin
            @(posedge vif.clk);
        end
        vif.valid <= 1;
        vif.addr <= req.addr;
        vif.rnw <= req.read_not_write;
        if(req.read_not_write == 0) begin
            vif.write_data <= req.write_data;
        end
        while(vif.ready != 1) begin
            @(posedge vif.clk);
        end

        // na kraju transakcije podesiti polja za odgovor
        if(req.read_not_write == 1) begin
            req.read_data = vif.read_data; // vratiti procitani podatak, ukoliko je u pitanju citanje
        end
        req.error = vif.error; // podesiti status
        vif.valid <= 0; // kraj slanja

        seq_item_port.item_done(); // transakcija je zavrsena
    end
  endtask

endclass

class bidir_seq extends uvm_sequence #(bidir_seq_item);

  `uvm_object_utils(bidir_seq)

  bidir_seq_item req;

  rand int limit = 40; // broj iteracija

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

  task body();
    req = bidir_seq_item::type_id::create("req");

    repeat(limit) begin
      start_item(req);
      // adresa je ogranicena na validan opseg
      assert(req.randomize() with {addr inside {[16'h0100:16'h1000]};});
      finish_item(req);
      // req pokazuje na objekat sa novim vrednostima podesenim u drajveru
      uvm_report_info("seq_body", req.convert2string());
    end
  endtask

endclass
```

Kod 3: Primer *use* modela

# 4 Zadaci

**Zadatak** Implementirati drajver za primer "Cacl1" dizajna. Proveriti rad drajvera koristeći sekvence razvijene na prethodnim vežbama. Pogledati *waveform*-e. Da li se DUV ponaša kao što je očekivano?

# 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 4: 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 5: v7_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 6: v7_calc_driver

```
`ifndef CALC_SEQ_ITEM_SV
 `define CALC_SEQ_ITEM_SV

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

class calc_seq_item extends uvm_sequence_item;



  `uvm_object_utils_begin(calc_seq_item)
  `uvm_object_utils_end

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

endclass : calc_seq_item

`endif
```

Kod 7: v7_calc_seq_item

```
`ifndef CALC_AGENT_PKG
`define CALC_AGENT_PKG

package calc_agent_pkg;

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

  ////////////////////////////////////////////////////////
  // include Agent components : driver,monitor,sequencer
  ////////////////////////////////////////////////////////
  `include "v7_calc_seq_item.sv"
  `include "v7_calc_sequencer.sv"
  `include "v7_calc_driver.sv"

endpackage

`endif
```

Kod 8: v7_calc_agent_pkg

```
`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_seq_item seq_item1;
    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);
        seq_item1 = calc_seq_item::type_id::create("seq_item", this);
        seqr = calc_sequencer::type_id::create("seqr", this);
        seq_item1.randomize();
        seq_item1.print();
    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 9: v7_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 10: v7_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 11: v7_test_simple_2

```
`ifndef CALC_TEST_PKG_SV
 `define CALC_TEST_PKG_SV

package calc_test_pkg;

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

    import calc_agent_pkg::*;
    import calc_seq_pkg::*;

`include "v7_test_base.sv"
 `include "v7_test_simple.sv"
 `include "v7_test_simple_2.sv"


endpackage : calc_test_pkg

 `include "calc_if.sv"

`endif
```

Kod 12: v7_calc_test_pkg

```
`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 13: v7_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 14: v7_calc_simple_seq

```
`ifndef CALC_SEQ_PKG_SV
 `define CALC_SEQ_PKG_SV
package calc_seq_pkg;
   import uvm_pkg::*;      // import the UVM library
 `include "uvm_macros.svh" // Include the UVM macros
  import calc_agent_pkg::calc_seq_item;
   import calc_agent_pkg::calc_sequencer;
 `include "v7_calc_base_seq.sv"
 `include "v7_calc_simple_seq.sv"
     endpackage
`endif
```

Kod 15: v7_calc_seq_pkg

```
`ifndef CALC_TEST_PKG_SV
 `define CALC_TEST_PKG_SV

package calc_test_pkg;

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

   import calc_agent_pkg::*;
   import calc_seq_pkg::*;

`include "v7_test_base.sv"
 `include "v7_test_simple.sv"
 `include "v7_test_simple_2.sv"


endpackage : calc_test_pkg

 `include "calc_if.sv"

`endif
```

Kod 16: v7_calc_test_pkg

```
module calc_verif_top;
```

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

    import calc_test_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 17: v7_calc_verif_top