Funkcionalna verifikacija hardvera

Vežba 10
Razvoj *scoreboard* komponente

Sadržaj

1 Podela verifikacionih okruženja **4**
1.1 "Zlatni vektori" 4
1.2 Referenti modeli 4

2 *Scoreboard* **6**
2.1 Uloga 6
2.2 Struktura 6
2.3 Provere 8
2.4 Primer 8

3 Verifikacioni plan **10**

4 Zadaci **12**

5 Appendix **13**

Deseta vežba je posvećena *scoreboard*-u. Opisana je podela verfikacionih okruženja na osnovu načina vršenja provere očekivanih rezultata, opisana je implementacija *scoreboard*-a prateći UVM i opisane su često korišćene funkcionalnosti. Takođe je dat primer verifikacionog plana i opisana veza plana sa proverama u samom okruženju.

1 Podela verifikacionih okruženja

Jedna od osnovnih podela verifikacionih okruženja odnosi se na način kako se vrši provera očekivanih rezultata. Prema ovom kriterijumu sva okruženja mogu se podeliti u dve velike grupe:

- Okruženja bazirana na konceptu "zlatnih vektora"
- Okruženja bazirana na referentnim modelima

1.1 "Zlatni vektori"

Kod ovog tipa okruženja stimulus koji je neophodno dovesti na ulaze sistema koji se verifikuje, kao i očekivani odziv na svaki od ovih stimulusa unapred je definisan i smešten u skup takozvanih "zlatnih vektora". Pod zlatnim vektorom podrazumeva se jedan par vrednosti ulaznih signala sistema koji se verifikuje i očekivanog izlaza sistema u tom slučaju. Obzirom da su ulazi i izlazi u sistem zadati u vektorskoj formi (kao niz individualnih vrednosti za svaki od ulaznih portova i niz očekivanih vrednosti na svakom od izlaznih portova sistema koji se verifikuje), jedan ovakav par čini jedan "zlatni vektor". Za obavljanje verifikacije nije dovoljan samo jedan zlatni vektor već je neophodan veliki skup različitih zlatnih vektora. Ovaj skup zlatnih vektora najčešće se smešta u odgovarajuću datoteku koja se na početku simulacije učitava u verifikaciono okruženje. Prilikom simulacije testbenč uzima jedan po jedan zlatni vektor iz skupa i njegovu ulaznu komponentu dovodi na ulaze sistema koji je potrebno verifikovati. Nakon što se u procesu simulacije izračuna odziv sistema koji se verifikuje na ovu ulaznu komponentu zlatnog vektor, on se upoređuje sa izlaznom komponentom tekućeg zlatnog vektora. Ukoliko su oni identični proces verifikacije se nastavlja sa sledećim zlatnim vektorom. U slučaju da dolazi do odstupanja, proces verifikacije sa prekida, a identifikovano odstupanje se saopštava verifikacionom inženjeru kako bi mogao započeti proces lokalizacije greške (*bug tracing*) unutar sistema.

1.2 Referenti modeli

Glavni nedostatak testbenčeva baziranih na "zlatnim vektorima" ogleda se u činjenici da je pre početka procesa potrebno generisati odgovarajući skup "zlatnih vektora". Ovaj postupak se često radi ručno, od strane verifikacionog inženjera, i uključuje ne samo osmišljavanje interesantnih vrednosti ulaznog stimulusa, već i izračunavanje očekivanog ponašanja sistema koji se verifikuje na ovakav stimulus. Pošto se često radi ručno, ovaj proces je spor i podložan greškama koje zatim usporavaju dalji proces verifikacije.

Način kako da se ovaj nedostatak pristupa baziranog na "zlatnim vektorima" može prevazići je korišćenje testbenčeva baziranih na referentim modelima.

Pod referentnim modelom podrazumevamo model sistema koji se verifikuje napisan na visokom nivou apstrakcije. Referentni model treba da poseduje željenu funkcionalnost koja je trebalo da bude implementirana unutar sistema koji se verifikuje. Obzirom da se referentni model piše na visokom nivou apstrakcije on je znatno jednostavniji od modela sistema koji se verifikuje što olakšava njegov razvoj i smanjuje mogućnost unošenja nenamernih grešaka prilikom njegovog pisanja. Referentni model se unutar testbenča koristi kao refernca (otuda i njegovo ima) koja služi za poređenje odziva dobijenih od strane sistema koji se verifikuje na dovedenu pobudu.

Proces verifikacije unutar testbenča baziranog na referentnom modelu odvija se na sledeći način. Unutar testbenča instancionirane su dve komponente: model sistema koji želimo da verifikujemo i njegov referentni model. Prilikom procesa simulacije na ulaze oba modela dovode se iste sekvence ulaznih vektora, a odzivi referentnog modela i modela sistema koji se verifikuje se zatim porede kako bi se utvrdilo da li postoji neko odstupanje. Obzirom da oba modela modeluju istu funkcionalnost odstupanja ne bi trebalo da bude. Ukoliko se da je do odstupanja došlo, to predstavlja indikaciju o postojanju greške unutar nekog od dva modela. Simulacija se u tom trenutku prekida, a zadatak

verifikacionog inženjera je da utvrdi u kojem od modela se nalazi greška.

Svi referentni modeli mogu se podeliti u dve velike grupe, prema tome koliko precizno modeluju funkcionalnost koju sistem koji se verifikuje treba da poseduje:

- Transakcione referentne modele ("transaction level" referentni modeli - ovi referentni modeli modeluju očekivano ponašanje sistema koji se verifikuje do nivoa transakcija, a ne individualnih perioda (ciklusa) klok signala. Transkcija predstavlja apstrakciju procesa obrade podataka unutar nekog sistema, pri čemu se vremenski period potreban za završetak obrade zanemaruje. Jedino što nas kod transakcija interesuje jeste rezultat obrade nekog ulaznog podataka (šta god ta obrada podrazumevala), a ne i vremenski redosled koraka koji dovodi do tog rezultata.
- Referentne modele na nivou ciklusa ("cycle accurate" referentni modeli - ovi modeli, za razliku od transakcionih, modeluju očekivano ponašanje sistema koji se verifikuje sve do nivoa individulanih ciklusa klok signala.

Zbog svih prethodno navedenih osobina, u praksi se često sreće transakcioni referentni model. I u UVM metodologiji se koristi ovaj model koji je uglavnom impementiran u sklopu *scoreboard*-a. U literaturi se sreće i naziv *predictor*.

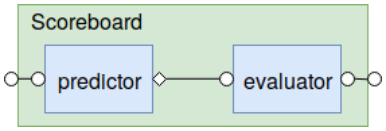
2 *Scoreboard*

Važan element svakog *self-checking* okruženja je upravo *scoreboard*. U njemu se uglavnom vrše provere ispravne funkcionalnosti sistema koji se verifikuje na sistemskom nivou. Sadržaj i uloga *scoreboard*-a dosta zavisi od same implementacije, ali u većini slučajeva je povezan sa ostatkom okruženja preko TLM interfejsa preko kojih prima transakcije. Te transakcije zatim analizira uz pomoć referentnog modela ili zlatnih vektora i poredi očekivane rezultate sa dobijenim transakcijama. Referentni model može biti implementiran u sklopu samog *scoreboard*-a ili kao zasebna komponenta.

Scoreboard se nalazi u UVM *environment* klasi odakle može komunicirati sa monitorima ili ostalim komponentama. *Scoreboard* nije povezan sa DUT-om jer sve potrebne informacije treba da dobija iz odgovarajućih monitora, a ne direktno preko interfejsa.

2.1 Uloga

Scoreboard je često najkompleksniji deo okruženja i najteži za implementaciju. Može se podeliti na dva glavna dela: predikcija i evaluacija. Prvi korak je utvrditi šta je ispravna funkcionalnost odnosno koji rezultat očekujemo. Kada se tačan rezultat utvrdi vrši se provera rezultata DUT-a odnosno da li se observirani rezultati slažu sa očekivanim. Dobra praksa je razdvojiti ova dva glavna zadatka *scoreboard*-a. Time se dobija na fleksibilnosti koda i olakšava se ponovna upotreba.



Slika 1: Primer *scoreboard*-a

Prediktor ili referentni model je komponenta koja modeluje funkcionalnost DUT-a. Prima isti ulazni stimulus kao i DUT i generiše rezultat koji je ispravan po funkcionalnoj specifikaciji. Može biti implemetiran ili u skopu *scoreboard*-a ili kao zasebna klasa. Referenti model može biti povezan sa jednim ili više monitora od kojih prima transakcije i na osnovu njih računa rezultat. Transakcije koje prima od monitora sadrže informacije o stimulusu koji je poslat DUT-u. Nakon procesiranja transakcija i računanja očekivanog rezultata, ove informacije se šalju na dalju evaluaciju odnosno poređenje očekivanih rezultata sa obzerviranim. Pošto referenti modeli treba da implementiraju istu funkcionalnost kao DUT, ali na višem nivou apstrakcije, ukoliko je potrebno, mogu biti pisani i u različitim jezicima (npr. C, C++, SV ili SystemC).

2.2 Struktura

Kostur *scoreboard*-a je dat ispod:

```
class calc_scoreboard extends uvm_scoreboard;

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

  // This TLM port is used to connect the scoreboard to the monitor
  uvm_analysis_imp#(calc_seq_item, calc_scoreboard) item_collected_imp;

  int num_of_tr;

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

```
  function new(string name = "calc_scoreboard", uvm_component parent = null);
    super.new(name,parent);
    item_collected_imp = new("item_collected_imp", this);
  endfunction : new

  function write (calc_seq_item tr);
    calc_seq_item tr_clone;
    $cast(tr_clone, tr.clone());
    if(checks_enable) begin
      // do actual checking here
      // ...
      // ++num_of_tr;
    end
  endfunction : write

  function void report_phase(uvm_phase phase);
    `uvm_info(get_type_name(), $sformatf("Calc scoreboard examined: %0d transactions", num_of_tr),
    UVM_LOW);
  endfunction : report_phase

endclass : calc_scoreboard
```

Kod 1: Kostur *scoreboard*-a

Scoreboard nasleđuje *uvm_scoreboard* klasu. Sadrži jednu ili više TLM konekcija (opisanih u vežbi 8) sa monitorima preko kojih prima transakcije. Dobra praksa je i omogućiti način kontrole provera koje se rade u *scoreboard*-u. Ovo je moguće odraditi ili preko zasebnih kontrolnih polja unutar same klase ili korišćenjem konfiguracione klase koja će sadržati odgovarajuća polja (konfiguracije su opisane u prethodnoj vežbi).

Ukoliko *scoreboard* treba da sadrži više TLM konekcija, ovo je moguće uraditi na sledeći način ("_1" i "_2" su proizvoljno odabrani nazivi):

```
`uvm_analysis_imp_decl(_1)
`uvm_analysis_imp_decl(_2)

class calc_scoreboard extends uvm_scoreboard;

  uvm_analysis_imp_1#(calc_frame, calc_scoreboard) port_1;
  uvm_analysis_imp_2#(calc_frame, calc_scoreboard) port_2;

  function new(string name = "calc_scoreboard", uvm_component parent = null);
    super.new(name, parent);
    port_1 = new("port_1", this);
    port_2 = new("port_2", this);
  endfunction

  function void write_1(calc_frame t);
    // ...
  endfunction

  function void write_2(calc_frame t);
    // ...
  endfunction

endclass
```

Kod 2: Primer vise TLM konekcija

Povezivanje sa odgovarajućim monitorima se uglavnom vrši u *environment* klasi, npr:

```
agent1.mon1.item_collected_port.connect(scbd.port_1);
agent2.mon2.item_collected_port.connect(scbd.port_2);
```

2.3 Provere

Kao što je već navedeno u vežbi o monitoru, dobra praksa je da se za implementaciju provera koriste *immediate assertion* naredbe. Sintaksa ovih naredbi je:

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

A primer upotrebe:

```
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))
```

Prilikom implementacije provera treba voditi računa o vezi sa verifikacionim planom. Svaka napisana provera mora biti dobro dokumentovana u planu, a sam plan referenciran u kodu (pogledati naredno poglavlje).

2.4 Primer

U nastavku je dat primer jednostavnog *scoreboard*-a. *Scoreboard* je povezan sa dve komponente od kojih prima transakcije tipa *Packet*, jedna sadrži očekivani rezultat, a druga obzervirani. Svaki put kada se primi transakcija preko *Drvr2Sb_port*-a ista se smešta u red. Prilikom primanja transakcije na *Rcvr2Sb_port*-u prvo se proverava da li je transakcija očekivana i ukoliko jeste poredi sa prvom očekivanom transakcijom koja je smeštena u red. Primetiti da se za provere koriste *assert* naredbe umesto *if-else* bloka.

```
`uvm_analysis_imp_decl(_rcvd_pkt)
`uvm_analysis_imp_decl(_sent_pkt)

class Scoreboard extends uvm_scoreboard;

  `uvm_component_utils(Scoreboard)

  Packet exp_que[$];

  uvm_analysis_imp_rcvd_pkt #(Packet,Scoreboard) Rcvr2Sb_port;
  uvm_analysis_imp_sent_pkt #(Packet,Scoreboard) Drvr2Sb_port;

  function new(string name = "Scoreboard", uvm_component parent = null);
    super.new(name, parent);
    Rcvr2Sb_port = new("Rcvr2Sb", this);
    Drvr2Sb_port = new("Drvr2Sb", this);
  endfunction : new

  function void write_rcvd_pkt(input Packet pkt);
    Packet exp_pkt;

    asrt_exp_queue_not_empty : assert (exp_que.size()) begin
      exp_pkt = exp_que.pop_front();
      asrt_pkt_compare : assert (pkt.compare(exp_pkt))
        `uvm_info(get_type_name(), "Sent packet and received packet matched", UVM_MEDIUM);
        else
          `uvm_error(get_type_name(), "Sent packet and received packet mismatched");
    end
      else begin
        `uvm_error(get_type_name(), "No more packets to in the queue to compare");
      end
  endfunction : write_rcvd_pkt

  function void write_sent_pkt(input Packet pkt);
    exp_que.push_back(pkt);
```

```
    endfunction : write_sent_pkt

    function void report_phase(uvm_phase phase);
        `uvm_info(get_type_name(), $psprintf("Scoreboard Report \n", this.sprint()), UVM_LOW);
    endfunction : report_phase

endclass : Scoreboard
```

Kod 3: Primer *scoreboard*-a

3 Verifikacioni plan

Prvi korak u postupku verifikovanja nekog dizajna je uvek kreiranje verfikacionog plana. Verifikacioni plan se kreira na osnovu funkcionalne specifikacije i sadrži više poglavlja, pri čemu se svaki odnosi na specifičan zadatak u verifikacionom poslu. Svaki plan treba da sadrži opis verifikacionih nivoa, funkcionalnosti koje treba verifikovati, opis specifičnih testova i metoda, plan za prikupljanje podataka o pokrivenosti, test scenarije, ... Pored ovih tehničkih zahteva, u planu se nalaze i zahtevi vezani za menadžment projekta kao što su potrebni alati, rizici u procesu verifikacije, potrebni resursi, raspored odnosno vreme potrebno da se završi svaki zadatak, itd. Verifikacioni plan se kreira na početku projekta, ali se, po potrebi, može modifikovati i dopunjavati u toku samog rada.

Za primer verifikacije "Calc1" dizajna mnogi delovi verifikacionog plana su nam unapred bili određeni npr. alat koji koristimo (QuestaSim), kao i jezik i metodologija (SystemVerilog/UVM). U ovoj vežbi ćemo se posvetiti delu verifikacionog plana posvećenom opisu funkcionalnosti, dok ćemo se na jednoj od narednih vežbi fokusirati na prikupljanje pokrivenosti. Za detaljan opis kompletnog verifikacionog plana pogledati odgovarajuće predavanje.

Kao što smo već naveli, verifikacioni plan treba da sadrži opis funkionalnosti koje treba proveriti. Za primer "Calc1" dizajna, funkcionalnosti možemo podeliti u tri grupe:

1. U ovu grupu spadaju osnovne funkcionalnosti "Calc1" dizajna koje se moraju verifikovati:
   - 1.1. Osnovi protokol (komanda / odgovor) na svakom portu
   - 1.2. Osnovne operacije svake komande na svakom portu
   - 1.3. *Overflow* i *underflow* za operacije sabiranja i oduzimanja
2. Malo kompleksniji scenariji uključuju:
   - 2.1. Posle svake komande može slediti bilo koja druga komanda (naredna komanda mora takođe biti ispravno obavljena)
     - 2.1.1. Za svaki port
     - 2.1.2. Za sve portove (npr. četiri paralelne komande sabiranja)
   - 2.2. Nijedan port nema prioritet
   - 2.3. Viših 27 bitova drugog operanda se ignorišu za obe *shift* komande
   - 2.4. Granični slučajevi za svaku komandu:
     - 2.4.1. Sabiranje dva broja gde se desi *overflow* za 1 (32'h FFFFFFFF + 1)
     - 2.4.2. Sabiranje dva broja gde je rezultat 32'h FFFFFFFF
     - 2.4.3. Oduzimanje dva jednaka broja
     - 2.4.4. Oduzimanje broja gde se desi *underflow* za 1 (drugi operand je veći od prvog operanda)
     - 2.4.5. *Shift* za nula mesta (rezultat je operand 1)
     - 2.4.6. *Shift* za 31 mesto (maksimalno dozvoljeno)
   - 2.5. Dizajn ignoriše ulazne podatke osim kada su oni validni (kada postoji i komanda i u narednom ciklusu)
3. Klasične provere koje se nalaze u većini verifikacionih planova:
   - 3.1. Ispravno rukovanje ilegalnih komandi
   - 3.2. Neaktivnost izlaza. "Calc1" treba da generiše izlazne vrednosti samo kao rezultat izvršavanja neke komande
   - 3.3. Dizajn se ispravno reaguje na reset signal

Nakon što se izdvoje funkcionalnosti, moraju se odrediti specifične provere i/ili testovi koji služe za njihovu verifikaciju. Svaki član u verifikacionom planu, ali i svaki čeker u okruženju moraju biti dobro dokumentovani i povezani. Ovo znači da za svaku funkcionalnost u planu mora postojati implementiran čeker i/ili test koji je pokriva. Takođe, svi čekeri u okruženju treba da su povezani sa određenim delom plana. Dobra dokumentacija olakšava proces verifikacije i omogućava brže i lakše pronalazenje i izolovanje grešaka. Na primer, deo plana koji služi za proveru *overflow*-a za operaciju sabiranja mogao bi biti nalik:

Group	**Feature**	**Description**	**Checker type**	**Checker name**	**Test**	**Pass/Fail**
1	overflow	check overflow for add command	immediate assertion	asrt_add_overflow	test_add_op	pass

Tabela 1: Primer *overflow* provere

Dok bi odgovarajuća provera mogla biti implementirana u *scoreboard*-u, npr:

```
// group: 1
// feature: overflow
// description: check overflow for add command
asrt_add_overflow : assert ( ... )
else
  `uvm_error(get_type_name(), "Overflow error")
```

Ovde se takođe vidi prednost korišćenja labela u *assert* tvrđenjima jer se mogu vrlo lako povezati sa verifikacionim planom. Takođe, dobra praksa je pisati što više komentara koje opisuju proveru i vezu sa planom (ime čekera, opis, mesto u planu, ...).

4 Zadaci

Zadatak Napisati verifikacioni plan za "Calc1" dizajn. Šta će plan sadržati? Na koje funkcionalnosti se treba fokusirati? (deo za prikupljanje pokrivenosti ostaviti za naredne vežbe).

Zadatak Implementirati *scoreboard* za Calc1 dizajn. Povezati provere sa verfikacionim planom.

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

```
class calc_agent extends uvm_agent;

   // components
   calc_driver drv;
   calc_sequencer seqr;
   calc_monitor mon;

   // configuration
   calc_config cfg;

   `uvm_component_utils_begin (calc_agent)
      `uvm_field_object(cfg, UVM_DEFAULT)
   `uvm_component_utils_end

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

   function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      if (!uvm_config_db#(calc_config)::get(this, "", "calc_config", cfg))
        `uvm_fatal("NO_CFG",{"Config object must be set for: ",get_full_name(),".cfg"})

      mon = calc_monitor::type_id::create("mon", this);
      if(cfg.is_active == UVM_ACTIVE) begin
         drv = calc_driver::type_id::create("drv", this);
         seqr = calc_sequencer::type_id::create("seqr", this);
      end

   endfunction : build_phase

   function void connect_phase(uvm_phase phase);
```

```
      super.connect_phase(phase);
      if(cfg.is_active == UVM_ACTIVE) begin
        drv.seq_item_port.connect(seqr.seq_item_export);
      end
  endfunction : connect_phase

endclass : calc_agent
```

Kod 5: v10_calc_agent

```
class calc_config extends uvm_object;

  uvm_active_passive_enum is_active = UVM_ACTIVE;

  `uvm_object_utils_begin (calc_config)
    `uvm_field_enum(uvm_active_passive_enum, is_active, UVM_DEFAULT)
  `uvm_object_utils_end

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

endclass : calc_config
```

Kod 6: v10_calc_config

```
`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("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);
      `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 7: v10_calc_driver

```
`ifndef CALC_ENV_SV
 `define CALC_ENV_SV
```

```
class calc_env extends uvm_env;

    calc_agent agent;
    calc_scoreboard scbd;

    `uvm_component_utils (calc_env)

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

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        agent = calc_agent::type_id::create("agent", this);
        scbd = calc_scoreboard::type_id::create("scbd", this);
    endfunction : build_phase

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        agent.mon.item_collected_port.connect(scbd.item_collected_imp);
    endfunction : connect_phase

endclass : calc_env

`endif
```

Kod 8: v10_calc_env

```
class calc_scoreboard extends uvm_scoreboard;

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

    // This TLM port is used to connect the scoreboard to the monitor
    uvm_analysis_imp#(calc_seq_item, calc_scoreboard) item_collected_imp;

    int num_of_tr;

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

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

    function write (calc_seq_item tr);
        calc_seq_item tr_clone;
        $cast(tr_clone, tr.clone());
        if(checks_enable) begin
            // do actual checking here
            // ...
            // ++num_of_tr;
        end
    endfunction : write

    function void report_phase(uvm_phase phase);
        `uvm_info(get_type_name(), $sformatf("Calc scoreboard examined: %0d transactions", num_of_tr),
       UVM_LOW);
    endfunction : report_phase

endclass : calc_scoreboard
```

Kod 9: v10_calc_scoreboard

```
`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 10: v10_calc_seq_item

```
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("NO_IF",{"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 11: v10_calc_monitor

```
`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 12: v10_calc_sequencer

```
`ifndef TEST_BASE_SV
`define TEST_BASE_SV

class test_base extends uvm_test;

   calc_env env;
   calc_config cfg;

   `uvm_component_utils(test_base)

   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);
      env = calc_env::type_id::create("env", this);
      cfg = calc_config::type_id::create("cfg");
      uvm_config_db#(calc_config)::set(this, "*", "calc_config", cfg);
   endfunction : build_phase

   function void end_of_elaboration_phase(uvm_phase phase);
      super.end_of_elaboration_phase(phase);
      uvm_top.print_topology();
   endfunction : end_of_elaboration_phase

endclass : test_base

`endif
```

Kod 13: v10_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(env.agent.seqr);
    phase.drop_objection(this);
  endtask : main_phase

endclass

`endif
```

Kod 14: v10_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 15: v10_test_simple_2

```
`ifndef TEST_LIB_SV
`define TEST_LIB_SV

`include "tests/v10_test_base.sv"
`include "tests/v10_test_simple.sv"
`include "tests/v10_test_simple_2.sv"

`endif
```

Kod 16: v10_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 17: v10_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 18: v10_calc_simple_seq

```
`ifndef CALC_SEQ_LIB_SV
 `define CALC_SEQ_LIB_SV

 `include "sequences/v10_calc_base_seq.sv"
 `include "sequences/v10_calc_simple_seq.sv"

`endif
```

Kod 19: v10_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 "v10_calc_config.sv"

 `include "v10_calc_seq_item.sv"
 `include "v10_calc_driver.sv"
 `include "v10_calc_sequencer.sv"
 `include "v10_calc_monitor.sv"
 `include "v10_calc_agent.sv"

 `include "v10_calc_scoreboard.sv"
 `include "v10_calc_env.sv"

 `include "sequences/v10_calc_seq_lib.sv"
 `include "tests/v10_test_lib.sv"
```

```
endpackage : calc_verif_pkg

`include "calc_if.sv"

`endif
```

Kod 20: v10_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 21: v10_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 22: calc_run