VEŽBA 23

U okviru ove vežbe upoznaćemo se sa kreiranjem UVM sequenci.

U direktorijumu:

23_UVM_Sequences,

23_UVM_Sequences\tb_classes

nalaze se fajlovi: ***add_sequence.svh, add_sequence_item.svh, command_monitor.svh, coverage.svh, driver.svh, env.svh, fibonacci_sequence.svh, fibonacci_test.svh, full_test.svh, maxmult_sequence.svh, parallel_sequence.svh, parallel_test.svh, random_sequence.svh, reset_sequence.svh, result_monitor.svh, result_transaction.svh, runall_sequence.svh, scoreboard.svh, sequence_item.svh, short_random_sequence.svh, tinyalu_base_test.svh, tinyalu_run_sequence.svh, tinyalu_sequence.svh***.

U dosadašnjem toku kursa, radili smo na razdvajanju klasa podataka od strukturnih klasa (koje opredeljuju arhitekturu). U tom procesu stigli smo do tačke u kojoj je praktično spoj strukture i podataka bio realizovan na nivou ***tester*** klase. ***Tester*** je obavljao dve aktivnosti, a to su formiranje transakcija i „hranjenje" testbenča datim transakcijama. Ako se setimo osnovne ideje vodilje u okviru OOP tehnike, jedna klasa treba da obavlja jednu aktivnost, jasno nam je da ovakav ***tester*** izlazi iz datog okvira. Ukoliko želimo da promenimo tip transakcije, to možemo da uradimo prepisivanjem randomizacije, međutim ukoliko želimo da promenimo broj transakcija i način na koji se one šalju testbenču, tada moramo da preradimo celu ***tester*** klasu. Da bismo uradili ovo razdvajanje, prvi korak je da uvodimo ***sequence*** klasu, odnosno objekat te klase kojim može da se pobuđuje testbenč (generiše stimulus). ***Sequence*** klasa sadrži željene podatke za pobudu.

```
class sequence_item extends uvm_sequence_item;

  `uvm_object_utils(sequence_item);

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

  rand byte unsigned      A;
  rand byte unsigned      B;
  rand operation_t        op;
  shortint  unsigned      result;
```

Sama ***uvm_sequence_item*** klasa nasleđuje ***uvm_transaction*** klasu, što podrazumeva nasleđivanje poznatih ***do_compare***, ***do_copy***, ***convert2string*** metoda uz proširenje na nivou transakcija.

U našem prvom kreiranju ***sequence_item*** –a uzimamo naš stari ***transaction*** i preimenujemo ga u ***sequence_item***, nasledimo ***uvm_sequence_item*** i naravno registrujemo ga pri fabrici, korišćenjem ***uvm_object_utils.*** Iz prethodnog primera, ***tester*** je unutar ***testbench-***a, generisao ***transaction*** i smeštao ih u ***fifo***. Sada tu ulogu preuzima blok po imenu ***sequencer***. Dakle drugi korak je da zamenimo ***tester*** našim ***sequencer***-om.

Sequencer je veoma jednostavan, i vrlo retko moramo da modifikujemo. Tipično koristimo osnovni ***uvm_sequencer*** , pri čemu ga parametrizujemo time što mu opredelimo da da želimo da pomoću njega baratamo

sa našim ***sequence_item*** –ima. Formalno to uradimo tako što kreiramo naš tip, odnosno parametrizujemo ***uvm_sequencer*** tako da hendla naše ***sequence_item*** -e i opredelimo mu naziv ***sequencer***. Deo koda niže preuzet je iz fajla ***tinyalu_pkg.sv***.

```
`include "sequence_item.svh"
typedef uvm_sequencer #(sequence_item) sequencer;
```

Od sada u našem ***testbench***-u imamo ***sequencer***,što je ***uvm_sequencer*** koji upravlja sa našim ***sequence***-ama. Treći korak je da nadogradimo ***driver***, odnosno prilagodimo ga ***sequencer***-u. Vidimo deo koda iz našeg ***driver.svh*** bloka:

```
class driver extends uvm_driver #(sequence_item);
  `uvm_component_utils(driver)

  virtual tinyalu_bfm bfm;

  function void build_phase(uvm_phase phase);
    if(!uvm_config_db #(virtual tinyalu_bfm)::get(null, "*","bfm", bfm))
      `uvm_fatal("DRIVER", "Failed to get BFM")
  endfunction : build_phase

  task run_phase(uvm_phase phase);
    sequence_item cmd;

    forever begin : cmd_loop
      shortint unsigned result;
      seq_item_port.get_next_item(cmd);
      bfm.send_op(cmd.A, cmd.B, cmd.op, result);
      cmd.result = result;
      seq_item_port.item_done();
    end : cmd_loop
  endtask : run_phase
```

Driver nasleđuje ***uvm_driver*** i kao parametar dobija ***sequence_item***. Dakle paremetrizovali smo ***driver*** sa našim ***sequence_item***-om; koje će driver dobijati preko već automatski kreiranog ***seq_item_port*** –a. Ovako formirani ***uvm_driver*** preuzima ***sequence_item***–e sa svog ***seq_item_port*** –a. U prethodnom primeru, u vežbi broj 22, radili samo sa ***driver***-om, kome smo predavali ***command_transaction***-e preko ***command_port***-a, iščitavanje ovakvog porta je izgledalo ovako: ***command_port.get(command)***. Sličnu analogiju imamo ovde, iščitavanje ***seq_item_port***-a izgleda ovako: ***seq_item_port.get_next_item(cmd)***.

Pogledajmo sada ***environment*** (iz fajla ***env.svh***),

```
class env extends uvm_env;
  `uvm_component_utils(env);

  sequencer      sequencer_h;
  coverage       coverage_h;
  scoreboard     scoreboard_h;
  driver         driver_h;
  command_monitor command_monitor_h;
  result_monitor  result_monitor_h;
```

```
  function new (string name, uvm_component parent);
    super.new(name,parent);
  endfunction : new

  function void build_phase(uvm_phase phase);
    // stimulus
    sequencer_h  = new("sequencer_h",this);
    driver_h     = driver::type_id::create("driver_h",this);
    // monitors
    command_monitor_h    = command_monitor::type_id::create("command_monitor_h",this);
    result_monitor_h = result_monitor::type_id::create("result_monitor",this);
    // analysis
    coverage_h    = coverage::type_id::create ("coverage_h",this);
    scoreboard_h  = scoreboard::type_id::create("scoreboard",this);
  endfunction : build_phase

  function void connect_phase(uvm_phase phase);

    driver_h.seq_item_port.connect(sequencer_h.seq_item_export);

    command_monitor_h.ap.connect(coverage_h.analysis_export);
    command_monitor_h.ap.connect(scoreboard_h.cmd_f.analysis_export);
    result_monitor_h.ap.connect(scoreboard_h.analysis_export);
  endfunction : connect_phase

endclass : env
```

Instancirali smo ***sequencer*** i ***driver***. U ***connect_phase*** povezujemo ***driver_h.seq_item_port*** kao primaoca sekvenci sa izvorom sekvenci, a to je ***sequencer_h.seq_item_export***. Ovo je bio četvrti korak, instanciranje i povezivanje ***driver***-a i ***sequencer***-a u okviru ***environment***-a. Dakle ostvarili smo konekciju putem koje predajemo ***sequence_item***–e ***driver***-u. Sve ovo sada radi zajedno, ***sequencer*** je povezan na ***driver***, ako pogledamo kako izgleda ***driver***, on u beskonačnoj petlji, preuzima ***get_next_item***, što je ***sequence_item*** objekt po imenu ***cmd***. Zatim pročitanu sekvencu predaje bfm-u, pozivanjem taska ***bfm.send_op(cmd.A, cmd.B, cmd.op, result)*** podsetimo se ovde da task ***send_op*** predaje operande i tip operacije ***bfm***-u, dok od njega preuzima rezultat operacije, vidimo da su prva tri argumenta ovog taska ulazno orijentisani, dok je četvrti izlazno orijentisan. Rezultat koji nam ovaj task vraća smeštamo nazad u ***sequence_item*** objakat po imenu ***cmd***. Ovim smo omogućili da se povratna informacija o rezultatu izvršene operacije vrati preko ***sequencer***-a u ***sequence*** objekat. Osnovna prednost u radu sa sequencama svodi se na kreiranje raznolikog ponašanja/pobuda sistema bez kreiranje mnoštva različitih objekata. Ovo vidimo već uvidom u sam direktorijum ***...\tb_classes*** u kom se nalazi desetak različitih sekvenci: ***parallel_sequence.svh, runall_sequence.svh, random_sequence.svh***... njih upravo koristimo da na jednostavan način iskoristimo ovu prednost.

Peti korak je kreiranje sequence, obradićemo najpre ***fibonacci_sequence*** (iz fajla ***fibonacci_sequence.svh***). Poznato nam je da se Fibonačijev niz generiše tako što počinje sa vrednostima 0,1,1... zatim svaki naredni element predstavlja sumu prethodna dva, dakle nastavlja sa 2,3,5,8... Razlog zbog kog analiziramo upravo ***fibonacci_sequence*** je taj što za generisanje ovakve sekvence moramo očitavati rezultate koje nam vraća ***tinyalu*** blok. Većinom prosti testovi samo pobude ***tinyalu*** i oslane se na ***scoreboard***, da nam kaže da li su rezultati očekivani. Za toliko nam je ovde složeniji zadatak, dakle naša sekvenca mora da čita podatke iz ***tinyalu***. Implementacija klase ***fibonacci_sequence*** data je u listingu niže:

```
class fibonacci_sequence extends uvm_sequence #(sequence_item);
  `uvm_object_utils(fibonacci_sequence);

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


  task body();
    byte unsigned n_minus_2=0;
    byte unsigned n_minus_1=1;
    sequence_item command;

    command = sequence_item::type_id::create("command");

    start_item(command);
    command.op = rst_op;
    finish_item(command);

    `uvm_info("FIBONACCI", " Fib(01) = 00", UVM_MEDIUM);
    `uvm_info("FIBONACCI", " Fib(02) = 01", UVM_MEDIUM);
    for(int ff = 3; ff<=14; ff++) begin
     start_item(command);
     command.A = n_minus_2;
     command.B = n_minus_1;
     command.op = add_op;
     finish_item(command);
     n_minus_2 = n_minus_1;
     n_minus_1 = command.result;
     `uvm_info("FIBONACCI", $sformatf("Fib(%02d) = %02d", ff, n_minus_1),
          UVM_MEDIUM);
    end
  endtask : body
endclass : fibonacci_sequence
```

Ovde imamo klasu ***fibonacci_sequence*** koja nasleđuje ***uvm_sequence***, paremetrizovana je ***sequence_item*** tipom, što znači da barata sa našim ***sequence_item*** –ima. U drugoj liniji koda vidimo da smo ovu sekvencu registrovali pri fabrici, ***`uvm_object_utils(fibonacci_sequence)***. Sve ***sequence*** sadrže task po imenu ***body, body*** je sličan ***run_phase*** iz ***component***-e, on se automatski se pokreće i izvršava sve što je u njemu navedeno. Ključnu funkciju željene ***sequence*** realizuje upravo ***body*** task. Body task deklariše, a potom kreira ***sequence_item*** po imenu ***command*** i u nastavku prosleđuje ovakve ***sequence_item***-e sequencer-u. Pri nasleđivanju ***uvm_sequence***, odmah dobijemo nekoliko dostupnih metoda, kao što su ***start_item i finish_item***. Kada kreiramo naš ***sequence_item command*** i predamo ga ***start_item*** metodi: ***start_item(command)***, događa se sledeće: blokira se sve dok ***sequencer*** ne bude spreman da nas pusti da pristupimo ***testbench***-u, odnosno predamo dotični ***command driver***-u. Kada nam ***sequencer*** preda pristup ***driver***-u, postavljamo željenu operaciju (***rst_op***) u naš ***command***. Zatim pozivamo ***finish_item(command)***, dakle ponovo blokiramo sve dok se komanda ne izvrši.

Sada možemo početi sa generisanjem Fibonačijevog niza, smeštamo u varijable ***n_minus_2*** i ***n_minus_1*** člana, to su nula i jedan respektivno. Pokrećemo petlju, koristimo fibonacci brojeve od tri do četrnaest, i pozivamo *start_item*, čekamo da testbench postane spreman, kada ***sequencer*** kaže da je "red na nas", popunjavamo ***command.A*** sa ***n_minus_2, command.B*** sa ***n_minus_1***, postavljamo ***command_op*** na ***add_op***. Zatim čekamo na ***finish_item*** (sve blokirano dok se komanda ne izvrši) i iz njega čitamo nazad vraćeni ***command*** iz ***testbench***-a, odnosno ***driver***-a. Podsećamo se na realizaciju driver-a, u kom smo ažurirali vrednost rezultata i preko porta

seq_item_port.item_done() vraćali rezultat izvršene komande. Blokiranost pri pozivu ***start_item*** , odnosno ***finish_item*** u okviru ***sequence*** bloka je ekvivalentna blokiranosti pri pozivu ***seq_item_port.get_next_item(cmd), odnosno seq_item_port.item_done()*** u okviru ***drivera.***
Kada nam je ***finish_item(command)*** vratio ažurirani sequence item – sa upisanim rezultatom operacije sabiranja, kopiramo ***n_minus_1*** u ***n_minus_2***, pročitamo rezultat iz ***command.result*** i kopiramo ga u ***n_minus_1***. Dakle preuzeli smo rezultat operacije sabiranja i postavili ga u željenu varijablu, nakon toga iterativno se vrtimo u petlji.

Ovako smo kreirali sekvencu - ***(fibonacci_sequence)*** koja će nam generisati Fibonačijev niz, ako želimo da je pokrenemo u ***testu,*** potrebno je da kažemo ***sequencer*** –u koju sekvencu da preuzme.
Šesti korak podrazumeva kreiranje testa koji pokreće sekvence korišćenjem sequencer-a.
Pogledajmo sada kako izgleda ***test,*** iz fajla ***tinyalu_base_test.svh***

```
`ifdef QUESTA
virtual class tinyalu_base_test extends uvm_test;
`else
class tinyalu_base_test extends uvm_test;
`endif

  env      env_h;
  sequencer sequencer_h;

  function void build_phase(uvm_phase phase);
    env_h = env::type_id::create("env_h",this);
  endfunction : build_phase

  function void end_of_elaboration_phase(uvm_phase phase);
    sequencer_h = env_h.sequencer_h;
  endfunction : end_of_elaboration_phase

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

endclass
```

Nasleđujemo ***uvm_test***, kao i u dosadašnjoj praksi, izuzev što je on sada ***virtual*** klasa, što znači da ga ne možemo pokrenuti, već ga moramo naslediti, pa ćemo pokretati naslednike. U ***build_phase*** kreira environment i postavlja handler na njega ***env_h***.
U ***end_of_elaboration_phase***, nakon što je ***env_h*** kompletiran, lokalni hendler ***sequencer_h***, povezujemo na ***env_h.sequencer_h***. Podsetimo se da smo u ***environment-u***, kreirali i povezali naš sequencer. Šta se sada događa? Svaki test koji nasleđuje naš ***tinyalu_base_test*** moći će da preuzme handler ***sequencer***-a.
Pogledajmo pokretanje ***fibonacci_test***-a ***(***iz fajla ***fibonacci_test.svh***):

```
class fibonacci_test extends tinyalu_base_test;
  `uvm_component_utils(fibonacci_test);

  task run_phase(uvm_phase phase);
    fibonacci_sequence fibonacci;
    fibonacci = new("fibonacci");

    phase.raise_objection(this);
    fibonacci.start(sequencer_h);
```

```
    phase.drop_objection(this);
  endtask : run_phase

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

endclass
```

Nakon registracije ovog testa pri fabrici, u ***run_phase***, kreiramo ***fibonacci_sequence,*** i hendleru damo ime ***fibonacci*** , zatim pozovemo ***raise_objection*** da time obezbedimo neprekidnost izvršavanja taskova unutar našeg ***sequencer***-a. Zatim u okviru ***fibonacci sequence*** pozivamo ***start*** metodu, ovde naglašavamo da svaka sekvenca naslednica ***uvm_sequence*** ima ***start*** metodu, kojoj prosleđujemo handler ***sequencer***-a. Ovaj poziv ***fibonacci.start(sequencer_h)*** zapravo pokreće ***body*** metod unutar naše ***fibonacci_sequence.***
Ubrzo stižemo do ***start_item(command)*** linije, u kojoj predajemo upravo formirani ***sequence_item*** po imenu ***command*** našem testbenču preko ovde referenciranog ***sequencer***-a. Ovaj mehanizam smo već opisivali nekoliko pasusa ranije u okviru ove lab. vežbe.
Upotrebom ovakvih mehanizama možemo da formiramo vrlo fleksibilne testbenčeve bez potrebe da menjamo bilo šta unutar hardverske structure koju imamo. Kao primer navedene fleksibilnosti, pogledajmo ***runall_sequence*** koja sadrži tri različite ***sequence*** u sebi.

```
class runall_sequence extends uvm_sequence #(uvm_sequence_item);
  `uvm_object_utils(runall_sequence);

  protected reset_sequence reset;
  protected maxmult_sequence maxmult;
  protected random_sequence random;
  protected sequencer sequencer_h;
  protected uvm_component uvm_component_h;

 function new(string name = "runall_sequence");
  super.new(name);

   uvm_component_h =  uvm_top.find("*.env_h.sequencer_h");

   if (uvm_component_h == null)
    `uvm_fatal("RUNALL SEQUENCE", "Failed to get the sequencer")

   if (!$cast(sequencer_h, uvm_component_h))
     `uvm_fatal("RUNALL SEQUENCE", "Failed to cast from uvm_component_h.")


   reset = reset_sequence::type_id::create("reset");
   maxmult = maxmult_sequence::type_id::create("maxmult");
   random = random_sequence::type_id::create("random");
 endfunction : new

 task body();
   reset.start(sequencer_h);
   maxmult.start(sequencer_h);
   random.start(sequencer_h);
 endtask : body
```

endclass : runall_sequence

Primećujemo da smo ovde kreirali sve tri različite sekvence: ***reset_seqence, maxmult_sequence*** i ***random_sequence***, zatim u našem ***body*** tasku redom pozivamo ove tri sekvence jednu za drugom.

Ukoliko je potrebon možemo uraditi paralelno pozivanje sekvenci, kako je pokazano u primeru ***parallel_sequence***. U okvuru ***body*** taska (izlistanog u kodu niže) u ovom primeru vidimo da nakon pozivanja ***reset*** sekvence ***reset.start(m_sequencer)*** koja će obezbediti reset našeg DUT-a, pozivamo ***fork – join*** konstrukciju u koju smeštamo paralelno pozivanje ***fibonacci*** i ***short_random*** sekvence. Podsećamo se da će se izlazak iz ovog ***fork-join***-a desiti nakon završetka duže sekvence.

```
task body();
    reset.start(m_sequencer);
    fork
            fibonacci.start(m_sequencer);
            short_random.start(m_sequencer);
    join
  endtask : body
```