VEŽBA 13

U okviru ove vežbe upoznaćemo se sa ***UVM environment*** klasom (***env***), i načinom njenog korišćenja za kreiranje fleksibilnijeg ***testbench-a.***

U direktorijumu:

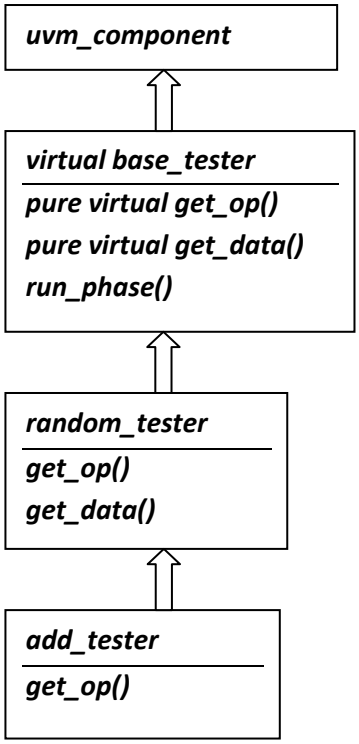
13_UVM_Environments,
nalaze se fajlovi:***tinyalu_bfm.sv, tinyalu_macros.svh, tinyalu_pkg.sv, top.sv***

13_UVM_Environments\tb_classes

nalaze se fajlovi: ***add_test.svh, add_tester.svh, base_tester.svh, coverage.svh, env.svh, random_test.svh, random_tester.svh, scoreboard.svh, vcs_base_tester.svh***

Na prethodnoj vežbi (broj 12) upoznali smo se sa UVM komponentama, konkretno smo se pozabavili sa blokovima tester, scoreboard i coverage koje smo formirali kao naslednike moćne klase UVM komponenti (***uvm_component***). Iz našeg prethodnog iskustva sa HW, poznato nam je da ove blokove možemo statički povezati na naš DUT preko bfm-a i formirati tvrdo definisanu arhitekturu, a zatim pozivanjem testova u toku trajanja simulacije možemo da pokrećemo testove. To nije loše međutim problem se pojavljuje u tome što imamo fiksiranu strukturu povezivanja blokova koje koristimo u simulaciji. Jasno nam je da to ograničenje može da nam značajno suzi kreativnost. U ovoj vežbi, kao što joj i ime kaže Environments, pozabavićemo se upotrebom fleksibilnih UVM mehanizama za kreiranje i menjanje okruženja u toku izvršavanja naše simulacije.

Na slici 1 prikazan je blok dijagram koji pojašnjava formiranje fleksibilnog rešenja za ***tester*** blokove.



Slika 1

Formirali smo osnovnu virtualnu klasu base_tester (zašto virtualnu, zato što nam neće nikad trebati kao izvorna, već ćemo u simulacijama koristiti neku od njenih naslednica, a s druge strane treba nam unificiran pojavni oblik naših testera). Unutar te klase formiramo prvu naslednicu - random_tester klasu koja implementira sve što nam je potrebno uz kao što joj ime kaže slučajni izbor operacije i operanada. Obzirom da nam je potrebna i klasa add_tester koja će ispitivati samo operaciju sabiranja, nju formiramo kao naslednicu random_tester klase uz prepisivanje metode get_op tako što će ova metoda unutar klase add_tester uvek vraćati kod operacije sabiranja, odnosno add_op.

Ako pogledamo kod iz fajla ***base_tester.svh***

```
virtual class base_tester extends uvm_component;
`uvm_component_utils(base_tester)
  virtual tinyalu_bfm bfm;

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

  pure virtual function operation_t get_op();

  pure virtual function byte get_data();

  task run_phase(uvm_phase phase);
    byte       unsigned      iA;
    byte       unsigned      iB;
    operation_t              op_set;
    shortint    result;

    phase.raise_objection(this);
    bfm.reset_alu();
    repeat (1000) begin : random_loop
      op_set = get_op();
      iA = get_data();
      iB = get_data();
      bfm.send_op(iA, iB, op_set, result);
    end : random_loop
    #500;
    phase.drop_objection(this);
  endtask : run_phase

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

```
endclass : base_tester
```

Ovde vidimo da imamo **virtual** klasu ***base_tester*** koja nasleđuje **uvm_component.** Reč virtual znači da ovu klasu ne možemo da instanciramo direktno, možemo samo da je redefinišemo u naslednicama. Klasa sadrži pokazivač na ***bfm,*** zatim sadrži ***build_phase*** koja povlači **bfm** iz ***config_db***-a, zatim ima **pure virtual** funkcije za ***get_op() i get_data()***. Zašto su ove metode pure virtual tipa, zato što su članice osnovne apstraktne klase i svakako će biti prepisane implementacijom unutar klasa naslednica, kao takve ne zahtevaju implementaciju unutar osnovne klase. Ove metode će biti popunjene u klasama naslednicama koje će ih redefinisati. Konačno ovde imamo ***run_phase*** koja kreira **1000** operacija nad operandima koje dobijamo pozivanjem metoda ***get_op()*** i ***get_data()***.

Ako pogledamo ***random_tester*** (iz fajla ***random_tester.svh***) zasnovan na ovome,

```
class random_tester extends base_tester;
  `uvm_component_utils (random_tester)
    function byte get_data();
   bit [1:0] zero_ones;
    zero_ones = $random;
    if (zero_ones == 2'b00)
      return 8'h00;
    else if (zero_ones == 2'b11)
      return 8'hFF;
    else
      return $random;
  endfunction : get_data

  function operation_t get_op();
    bit [2:0] op_choice;
    op_choice = $random;
    case (op_choice)
      3'b000 : return no_op;
      3'b001 : return add_op;
      3'b010 : return and_op;
      3'b011 : return xor_op;
      3'b100 : return mul_op;
      3'b101 : return no_op;
      3'b110 : return rst_op;
      3'b111 : return rst_op;
    endcase // case (op_choice)
  endfunction : get_op

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

endclass : random_tester
```

primećujemo da ***random_tester*** nasleđuje ***base_tester***, a pošto ***base_tester*** nasleđuje ***uvm_component***, i ***random_tester*** nasleđuje ***uvm_component***. Takođe registruje se pri uvm ***factory***-ju i redefiniše metode ***get_op()*** i ***get_data()***. Ima konstruktor ali ne mora da brine o ***build_phase*** ili ***run_phase*** zato što su one nasleđene iz roditeljske klase ***base_tester***.

Pogledajmo sada ***add_tester*** koji nasleđuje ***random_tester*** (fajl ***add_tester.svh***)

```
class add_tester extends random_tester;
  `uvm_component_utils(add_tester)

  function operation_t get_op();
    bit [2:0] op_choice;
    return add_op;
  endfunction : get_op

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

endclass : add_tester
```

Ovde se sve nasleđuje od random_tester-a osim get_op() metode, koja u njemu uvek vraća add_op, dakle uvek prepiseujemo kod operacije kodom operacijom sabiranja.

Sada imamo dve klase add_tester i random_tester koje želimo na najbolji i najlakši način da smestimo u test_bench, tu nam pomaže ***UVM environment***.

Ako pogledamo našu ***environment*** klasu iz fajla ***env.svh***

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

  base_tester   tester_h;
  coverage      coverage_h;
  scoreboard    scoreboard_h;

  function void build_phase(uvm_phase phase);
    tester_h     = base_tester::type_id::create("tester_h",this);
    coverage_h   = coverage::type_id::create ("coverage_h",this);
    scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);
  endfunction : build_phase

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

endclass
```

vidimo da klasa ***env*** ima deklarisane promenljive tipa ***base_tester, covarage*** i ***scoreboard***, zatim funkciju ***build_phase*** u kojoj instanciramo ***base_tester***, ***covarage*** i **scoreboard**. Znamo da ***base_tester*** ne može biti instanciran zato što je virtualna klasa, pa se pitamo kako smo kreirali nešto što koristi ***base_tester***. Vidimo da ovde ne koristimo standardno instanciranje pozivanjem konstruktora, već zapravo koristimo factory koncept.

Env klasa koristi base_tester varijablu kao placeholder, to nam ***uvm_env*** klasa dozvoljava uz naravno podrazumevanu naknadnu odluku od strane factory-ja koji naslednik base_tester klase će biti vraćen. Na taj način nam je omogućeno da kasnije opredelimo vrstu testera koji će biti fabrikovan.

Razmislimo šta ovim dobijamo:
- pre svega imamo jedinstvenu strukturu našeg testbenča u koji smo postavili tester opšteg tipa (samo smo rezervisali mesto za njega)
- na nivou testa donosimo odluku o tome koji tip testera nam odgovara, odnosno unutar build_phase našeg testa pozivamo factory i saopštavamo mu našu odluku da je željeni tester random_tester ukoliko pozivamo random_test; odnosno da je željeni tester add_tester ukoliko pozivamo add_test. Ovo razrešavamo u build_phase našeg testa, pogledati delove koda niže:

build_phase iz add_test.svh:
```
function void build_phase(uvm_phase phase);
  base_tester::type_id::set_type_override(add_tester::get_type());
  env_h = env::type_id::create("env_h",this);
endfunction : build_phase
```

build_phase iz random_test.svh:
```
function void build_phase(uvm_phase phase);
  base_tester::type_id::set_type_override(random_tester::get_type());
  env_h = env::type_id::create("env_h",this);
endfunction : build_phase
```

Dakle env klasa uz korišćenje factory metode ***set_type_override*** nam omogućava da u hodu prilikom pozivanja testa u samom testu donesemo odluku o tester-u koji će se postaviti u naš aktuelni env (našu aktuelnu environment strukturu).

U skripti run.do podsećamo se da pozivanje dva različita testa radimo nakon faze kompajliranja dizajna, u liniji:
vsim top_optimized -coverage +UVM_TESTNAME=random_test
odnosno:
vsim top_optimized -coverage +UVM_TESTNAME=add_test

U ovoj vežbi upoznali smo se sa veoma efikasnom tehnikom korišćenja polimorfizma primenjenog na testere, ukoliko možemo da ih izvedemo iz iste osnovne klase, na raspolaganju nam ostaje sloboda da u toku izvršavanja naše simulacije po potrebi menjamo testere bez potrebe da menjamo strukuturu testbenča.
Uveli smo veoma značajno razdvajanje funkcije koju test obavlja od structure testbenča. Jedinstven testbenč u okviru koga mogu da se tokom izvršavanja simulacije smenjuju testovi.