VEŽBA 11

U okviru ove vežbe upoznaćemo se sa prednostima rada sa ***UVM***. ***UVM universal verification methodology*** je biblioteka klasa i resursa zasnovanih na tim klasama. Iskoristićemo te resurse iz klasa za kreiranje ***testbench-a***. U prvom korišćenju ***UVM-a*** fokusiraćemo se na pokretanje više **testova** u okviru jednog **kompajliranja**, jer u realnosti obično nemamo dovoljno vremena da svaki put kada nešto želimo da testiramo kompajliramo ***testbench***, što za veće testbench-ove može da traje veoma veoma dugo. Jedna od osnovnih sistemskih prednosti UVM metodologije ogleda se upravo u tome da jednom iskompajliran testbench može biti pozivan više puta za pokretanje različitih testova.

Ovu prednost možemo da vidimo posmatranjem komandne skripte našeg Questa simulatora pod nazivom "**run.do**".

U osmoj liniji skripta radi kompajliranje svih izvornih VHD fajlova koji učestvuju u simulaciji.

vcom -f dut.f

U trinaestoj liniji skripta radi kompajliranje svih izvornih Verilog I System Verilog fajlova.

vlog -f tb.f

Naredna linija opredeljuje nivo optimizacije top modula koji se ispituje.

vopt top -o top_optimized +acc +cover=sbfec+tinyalu(rtl).

Šesnaesta linija poziva test po imenu random_test

vsim top_optimized -coverage +UVM_TESTNAME=random_test

27. linija poziva test po imenu add_test

vsim top_optimized -coverage +UVM_TESTNAME=add_test

Na koji način UVM omogućava ovu fleksibilnost? Korišćenjem Factory koncepta, obrađivanog u 9. vežbi. String koji prosleđujemo simulatoru u prvom pozivu simulatora glasi "**random_test**" dok u drugom pozivu prosleđuje "**add_test**" i time je selektovan željeni test koji se pokreće. Naravno u ovom primeru nismo direktno mi generisali factory, oslanjamo na UVM tehniku koju nam omogućava **uvm_pkg** biblioteka, u kojoj je definisan run_test(); task. Ovaj task preuzima +UVM_TESTNAME parametar iz komandne linije simulatora i instancira objekt željene klase. Naravno ovako elegantno prosleđivanje naziva testa i njegovo instanciranje i pokretanje automatizovano nam je omogućeno time što su obe klase koje ovako pozivamo, a to su **random_test** i **add_test** deklarisane kao izvedene iz **uvm_test** klase, pa stoga nema potrebe da se spuštamo na nivo direktnog generisanja Factory-ja za ovakvu upotrebu naših testova.

U direktorijumu:

11_UVM_Test\tb_classes

nalaze se fajlovi: ***add_test.svh, add_tester.svh, coverage.svh, random_test.svh, random_tester.svh, scoreboard.svh, selfcheck.svh***.

Prva stvar koju treba da pogledamo je modul ***top*** iz fajla ***top.sv***

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

  import   tinyalu_pkg::*;
`include "tinyalu_macros.svh"

  tinyalu_bfm      bfm();
  tinyalu DUT (.A(bfm.A), .B(bfm.B), .op(bfm.op),
           .clk(bfm.clk), .reset_n(bfm.reset_n),
           .start(bfm.start), .done(bfm.done), .result(bfm.result));

initial begin
 uvm_config_db #(virtual tinyalu_bfm)::set(null, "*", "bfm", bfm);
 run_test();
end

endmodule : top
```

Vidimo da smo ovde uključili **UVM** paket, i ***uvm_macros***. Takođe uključujemo i ***tinyaly*** paket koji sadrži sve osnovne definisane klase, takođe ovde iz pedagoških razloga uključujemo I makroe ***tinyaly_macros*** u konkretnom slučaju ovih makroa nema, ali rezervisali smo fajl za tu svrhu***.*** Instanciramo ***bfm*** i ***DUT*** kao što smo i do sada radili. U **initial begin** iskoristićemo jednu od konstrukcija UVM-a a to je UVM config database ***uvm_config_db***. Pri pozivu prosleđujemo mu virtualni tinyalu_bfm. Virtualni bfm predstavlja tip pokazivača na bfm, nakon toga pokrećemo test ***run_test()***. ***run_test()*** je još jedan metod iz ***UVM*** paketa. U ovoj funkciji pokrećemo test i gledamo na komandnu liniju koja su imena ***UVM*** testova plus argument i pokrećemo test koji hoćemo u ***run_test()***. Pogledajmo sada kako izgleda prvi test, ***random_test*** iz fajla ***random_test.svh***

```
class random_test extends uvm_test;
  `uvm_component_utils(random_test);

 virtual tinyalu_bfm bfm;

  function new (string name, uvm_component parent);
    super.new(name,parent);
    if(!uvm_config_db #(virtual tinyalu_bfm)::get(null, "*","bfm", bfm))
      $fatal("Failed to get BFM");
  endfunction : new


  task run_phase(uvm_phase phase);
    random_tester    random_tester_h;
    coverage  coverage_h;
    scoreboard scoreboard_h;
```

```
    phase.raise_objection(this);

    random_tester_h    = new(bfm);
    coverage_h  = new(bfm);
    scoreboard_h = new(bfm);

    fork
      coverage_h.execute();
      scoreboard_h.execute();
    join_none

    random_tester_h.execute();
    phase.drop_objection(this);
  endtask : run_phase

endclass
```

Klasa ***random_test*** nasleđuje ***uvm_test*** , ***uvm_test*** je osnovna klasa koja nam je data iz ***UVM*** **biblioteke**. U sledećoj liniji ***`uvm_component_utils(random_test);*** registrujemo našu klasu pri UVM fabrici ***(Factory)***, podsetimo se ovde primera sa lavovima i pilićima (vežba 9) gde smo pravili sopstvenu fabriku životinja. Ovde se oslanjamo na UVM metodologiju koja nam omogućava da naše različite testove proizvodimo u jedinstvenoj UVM fabrici; da bi to bilo izvodljivo moramo naš test registrovati na ovaj način. Ovako registrovan test može se pozivati iz komandne linije (pogledati pozivanje testova s početka ove vežbe). Ovako pozvan test iz komandne linije imaće za rezultat kreiranje jedne instance objekta klase ***random_test***.
Zatim vidimo da imamo promenljivu tipa ***virtual, tinyaly_bfm***, koja predstavlja pokazivač na **bfm**. Sledi konstruktor naše klase ***random_test***, koji mora striktno da prati UVM pravila, dakle prve dve linije konstruktora predstavljaju šablon koga se moramo držati. Nakon toga pozivamo ***get*** metodu da bismo iz top level database ***uvm_config_db*** preuzeli pokazivač na bfm. Šta smo ovim uradili, praktično smo preuzeli pokazivač na naš bfm sa top nivoa naše simulacije – podsetimo se da smo u fajlu ***top.sv*** na samom početku naše simulacije postavili naš ***bfm*** u konfiguracioni database, korišćenjem metode ***set*** u okviru klase ***uvm_config_db***. Dakle vidimo da nam UVM omogućava "fabrikovanje" testova i manipulisanje sa pokazivačem na bfm na vrlo visokom apstraktnom nivou u toku izvršavanja simulacije (during run-time).

Sledeće što imamo u ovom testu je task ***run_phase***; na sličan način kao u prethodnoj vežbi (broj 10) gde smo pozivali ***execute()*** task. Ovde pri pozivanju ***run_test()***-a sa vrha hijerarhije iz ***top.sv*** nivoa, UVM generiše test objekt korišćenjem fabrike i pokreće kreirani objekt pozivajući ***run_phase()*** metod. U konkretnom slučaju kreira objekte ***random_tester, covarage*** i ***scoreboard,*** prosleđuje im pointer na ***bfm*** i pokreće ***execute*** metodu svakog od njih.

Bitno je napomenuti da ***uvm_test*** klasa sadrži podrazumevanu ***run_phase()*** metodu i ona se koristi za izvršenje testa. Ukoliko želimo da naš test nešto izvršava, a naravno da je to slučaj, moramo u njemu kreirati sopstvenu ***run_phase()*** metodu u koju smeštamo upravo sve što želimo da se dogodi u našem testu. Dužni smo da se uklopimo u zahteve koje UVM nameće, a to u ovom slučaju znači da naša ***run_phase()*** metoda mora imati jedan argument tipa ***uvm_phase*** i moramo mu dati ime ***phase.*** Kada je ovo zadovoljeno, vidimo da sama metoda poziva praktično iste taskove kao i ***execute()*** metod iz prethodne vežbe (tamo smo imali tester, coverage i scoreboard

klase) uz dodatnu razliku da ovde obzirom da se sve više dižemo u apstraktne nivoe, sad imamo potrebu da opredelimo metodologiju zaustavljanja izvršavanja simulacije. Za tu svrhu koristimo metodu ***phase.raise_objection(this);*** pri čemu kao argument postavljamo hendler na nju samu korišćenjem ključne reči ***this***. Ovim pozivom postavljamo primedbu (objection) zatim na kraju naše faze izvršenja testa sklanjamo primedbu pozivom metode ***phase.drop_objection(this);*** i time više nema prepreka da se naša simulacija okonča.

Ako želimo da pogledamo ostale testove, u našem slučaju add_test vidimo da i on kao i random_test nasleđuje uvm_test, takođe je registrovan pri UVM fabrici pozivanjem makroa ***`uvm_component_utils***, ostatak je takođe identenčan, jedina razlika je što imamo ***add_tester*** objekat umesto ***random_tester*** objekta. Ako pogledamo u naš ***add_tester*** videćemo da on nasleđuje naš ***random_tester***, nasleđuje sve metode iz ***random_tester-a*** ali redefiniše funkciju ***get_op*** tako da uvek vraća ***add_op***.