VEŽBA 10

U okviru ove vežbe upoznaćemo se sa kreiranjem **objektno-orjentisanog testbench-a**. Na predhodnih nekoliko časova smo pričali o objektno-orjentisanom programiranju i objektno-orjentisanim tehnikama, sad ćemo to iskoristiti za kreiranje **testbench-a.**

U direktorijumu:

10_An_Object_Oriented_Testbench

nalaze se fajlovi**:** ***tinyaly_bfm.sv, tinyaly_pkg.sv, top.sv,***

Dok su u:

10_An_Object_Oriented_Testbench\tb_classes

smešteni: ***covarage.svh, scoreboard.svh, testbench.svh, tester.svh***.

Ako pogledamo fajl ***top.sv***

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

  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));

  tinyalu_bfm     bfm();

  testbench    testbench_h;

  initial begin
    testbench_h = new(bfm);
    testbench_h.execute();
  end

endmodule : top
```

vidimo dve instance, prva je ***tinyaly***, **DUT**, a sledeća je sistem verilog interface instanca **bfm** tipa ***tinyalu_bfm.*** Svi signali iz našeg ***bfm-a*** prosleđeni su jedan po jedan našem ***DUT-***u.

Ono što je novo ovde je promenljiva **testbench** koja sadrži objekat **testbench_h,** odnosno hendler objekta tipa testbench. Između ***initial begin*** i ***end*** linija vidimo da je **testbench_h** vezan na upravo kreirani **testbench.** Ako se prisetimo osnovnog klasičnog testbencha sa početne vežbe iz: ***02_Conventional_Testbench\tinyalu_dut***

u njemu smo direktno povezali ***tester, scoreboard i covarage,*** sa tinyalu DUT-om – prilično statično.

Sada umesto toga kreiramo objekat testbench, prilikom pozivanja njegovog konstruktora, prosleđujemo mu bfm; zatim pozivamo metodu ***testbench***-a zvanu ***execute***.

Sada ako pogledamo ***testbench*** klasu videćemo kako se zapravo testira naš ***DUT***. Pogledati ***testbench.svh.***

```
class testbench;

  virtual tinyalu_bfm bfm;

  tester    tester_h;
  coverage  coverage_h;
  scoreboard scoreboard_h;

  function new (virtual tinyalu_bfm b);
     bfm = b;
  endfunction : new

  task execute();
    tester_h    = new(bfm);
    coverage_h   = new(bfm);
    scoreboard_h = new(bfm);

    fork
      tester_h.execute();
      coverage_h.execute();
      scoreboard_h.execute();
    join_none
  endtask : execute
endclass : testbench
```

Vidimo konstruktore koji se praktično kače na ***bfm.*** Imamo ***task execute(),*** koji prvo kreira nove objekte ***tester_h, scoreboard_h, covarage_h***, a zatim korišćenjem fork join konstrukcije paralelno pokrećemo ova tri taska kao metode adekvatnih klasa. Svaki od ovih objekata ima pristup signalima **DUT-a** preko **bfm-a**. Podsetimo se da je pri pozivanju konstruktora sve tri navedene klase prosleđivan isti ***bfm,*** da bi svi osluškivali iste signale.

Ako pogledamo ***tester.svh*** vidimo da se naš tester objekat preko svog konstruktora povezuje na ***tinyaly.bfm*** za razliku od klasičnog povezivanja statičkog tester modula, kako je rađeno u 3. vežbi (videti direktorijum ***03_Interfaces_and_BFMs***).

```
class tester;

  virtual tinyalu_bfm bfm;

  function new (virtual tinyalu_bfm b);
     bfm = b;
  endfunction : new

  protected 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

  protected 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

  task execute();
    byte       unsigned       iA;
    byte       unsigned       iB;
    shortint   unsigned       result;
    operation_t               op_set;
    bfm.reset_alu();
    op_set = rst_op;
    iA = get_data();
    iB = get_data();
    bfm.send_op(iA, iB, op_set, result);
    op_set = mul_op;
    bfm.send_op(iA, iB, op_set, result);
    bfm.send_op(iA, iB, op_set, result);
    op_set = rst_op;
    bfm.send_op(iA, iB, op_set, result);
    repeat (10) begin : random_loop
      op_set = get_op();
      iA = get_data();
      iB = get_data();
      bfm.send_op(iA, iB, op_set, result );
      $display("%2h %6s %2h = %4h",iA, op_set.name(), iB, result);
    end : random_loop
    $stop;
  endtask : execute
endclass : tester
```

Primetićemo da ostatak koda izgleda isto kao u vežbi 3., imamo task execute koji radi isto kao i pre, koristi ***send_op*** task iz **bfm-a,** šalje sledeću sekvencu komandi, prvo **reset,** zatim operaciju množenja, ponovo reset, zatim 10 puta u petlji šalje slučajnu komandu i slučajne operande.

scoreboard.svh i ***covarage.svh*** rade isto kao i ***scoreboard.sv*** i ***covarage.sv*** pa ih nećemo detaljno opisvati.

Sistemski, ova vežba nam pokazuje kako sistem verilog omogućava povezivanje apstraktnih objekata na sintetizibilne module koje ispitujemo; za razliku od klasičnog pristupa u kome su tester, coverage i scoreboard bili verilog moduli – dakle potpuno statički klasično.