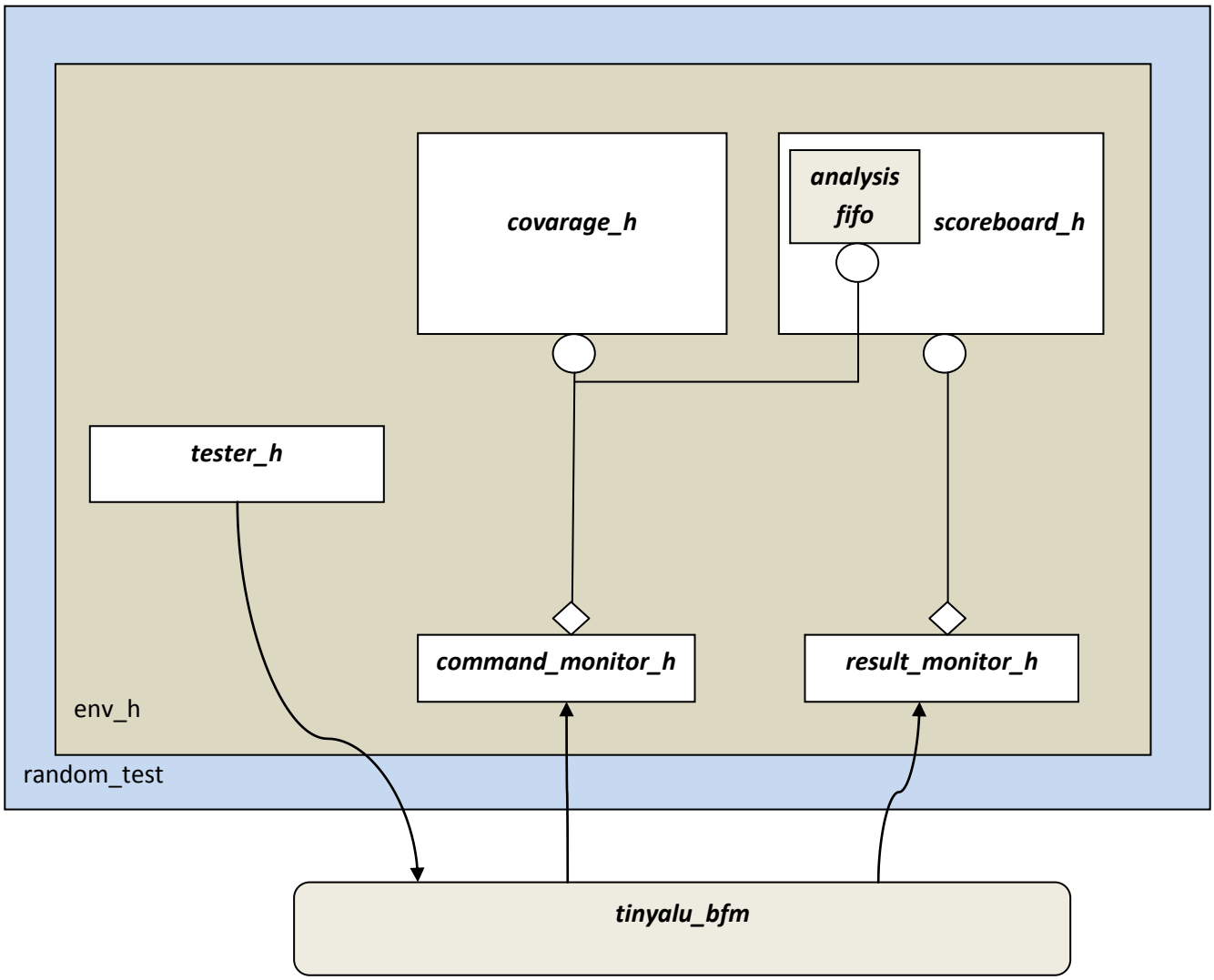
VEŽBA 16

U okviru ove vežbe upoznaćemo se sa korišćenjem ***analysis_port*** –a u testbenču. U prethodnoj vežbi (br 15) upoznali smo se sa konceptom ***analysis_port***-a na pedagoškom primeru povezivanja simulatora bacanja dve kocke (***dice_roller***-a) sa pratiocima – subscriber objektima ***coverage, histogram, average***. Sada nam je cilj da ovo znanje upotrebimo na našem osnovom zadatku, a to je UVM verifikacija tinyalu DUT-a. Podsetimo se vežbe 13 u kojoj smo povezivali naše objekte ***tester, coverage i scoreboard*** u jedinstvenu ***env*** klasu koja definiše strukturu testbenča. Veza između ova tri objekta realizovana je povezivanjem na bfm. Iz hardverske perspektive to je nešto elementarno, međutim na taj način u ovim objektima (pre svega sada govorimo o coverage i scoreboard objektima) uz direktno kačenje na bfm moramo da pratimo signalni protokol na magistrali i pored toga da proveravamo integritet podataka. Dakle opterećujemo ove objekte dvostruko! Ideja OOP-a je da za svaki zadatak opredelimo poseban objekat. Stoga ćemo uraditi razdvajanje praćenja signala na bfm magistrali (***command_monitor*** i ***result_monitor*** se direktno vezuju na ***bfm*** i preuzimaju komande i podatke respektivno) od same analize izdatih komandi i dobijenih rezultata, te aktivnosti prepuštamo ***coverage*** i ***scoreboard*** objektima, koji sada postaju specijalizovani samo za data analysis. Slika 1 prikazuje koncept po kome radimo vežbu.



Slika 1

U direktorijumima

16_Analysis_Ports_In_the_Testbench,

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

16_Analysis_Ports_In_the_Testbench\tb_classes

nalaze se fajlovi:

add_test.svh, add_tester.svh, base_tester.svh, command_monitor.svh, coverage.svh, env.svh, random_test.svh, random_tester.svh, result_monitor.svh, scoreboard.svh, vcs_base_tester.svh.

Dodatna motivacija za razdvajanje bfm – protokola na signalnom nivou od analize informacionog sadržaja ogleda se i u tome što smo u 13. vežbi preuzimali podatke sa bfm-a praćenjem transakcija na magistrali i u bloku ***coverage*** i u bloku ***scoreboard.*** Ovakva praksa može imati vrlo nepovoljne posledice u slučaju da u toku razvoja projekta dođe do promena u načinu signalizacije na magistrali, u tom slučaju na dva mesta bi morali da ažuriramo komunikacioni protokol. U slučaju složenijih magistrala i protokola, ovaj problem odneo bi veliku količinu radnih sati i bio bi potencijalan izvor velikog broja grešaka... Iz tog razloga opredeljujemo se da ekstrakciju podataka i komandi implementiramo unutar bfm bloka, gde joj je i mesto.

Dakle proširujemo ***tinyalu_bfm*** da radi protokol monitoring i prosleđuje podatke prema ***command_monitor***-u, odnosno ***result_monitor***-u. Ovi objekti ubacuju podatke preko ***analysis_port***-ova i možemo da ih pratimo kroz UVM mehanizam izvor – pratilac (***analysis_port*** – ***analysis_export***). U konkretnom slučaju ***analysis_port*** iz ***command_monitor***-a prosleđuje detektovane komande ka coverage bloku i analysis_fifo; dok scoreboard prati (***subscribes***) result_monitor.

Vidimo da ***covarage*** i ***analysis fifo*** iz ***scoreboard*** su ***subscribe analysis_port***, takođe primećujemo da ***scoarboard subscribe*** u ***result_monitor***. Mogli bi smo dodati još jedan objekat iznad i nazvati ga ***printer*** i imati ***subscribe*** u ***analysis_port***, u pozadini bi smo dobili kompatibilno štampanje sa ostatkom ***testbench***-a. Sve je ovo primer proširenja mogućnosti testbench-a kako bi bio još moćniji i fleksibilniji

Pogledajmo sada ***testbench***, krenućemo od ***tinyalu_bfm***-a (iz fajla ***tinyalu_bfm.sv***), koji ima i nešto novo u sebi, a to su promenljive ***command_monitor i result_monitor***, odnonso hendleri tih tipova.

```
interface tinyalu_bfm;
  import tinyalu_pkg::*;

  command_monitor command_monitor_h;
  result_monitor  result_monitor_h;
```

Ako pogledemo pri dnu fajla kako izgleda monitoring loop, odnosno ***cmd_monitor*** process:

```
  always @(posedge clk) begin : cmd_monitor
    bit new_command;
    if (!start)
      new_command = 1;
```

```
    else
      if (new_command) begin
        command_monitor_h.write_to_monitor(A, B, op);
        new_command = (op == 3'b000); // handle no_op
      end
  end : cmd_monitor
```

proverava se start signal i ukoliko je on visok proverava se da li smo u novoj komandi, ako jesmo upisujemo podatke (***A,B,op***) u ***command_monitor***, metodom ***write_to_monitor***. Nakon tog upisa resetuje se new_command bit, osim u slučaju da na magistrali imamo ***no_op*** komandu. Dakle ako smo imali regularnu komandu, upisujemo je i resetujemo new_command bit, zatim čekamo pojavu novog start signala…

Slično u procesu koji monitoruje reset, videti kod procesa niže

```
  always @(negedge reset_n) begin : rst_monitor
    if (command_monitor_h != null) //guard against VCS time 0 negedge
      command_monitor_h.write_to_monitor(A, B, rst_op);
  end : rst_monitor
```

Proces detektuje silaznu ivicu ***reset_n*** signala i kada se to desi upisuju se podaci istom metodom u ***command_monitor***. Zbog različite prirode reset-a u odnosu na ostale komande (ovde imamo klasičan asinhroni reset niskim nivoom signala, morali smo formirati ovakav process koji ga detektuje.

Proces niže detektuje okončanje komande i postavljanje rezultata na magistralu, ukoliko je done signal visok na rastućoj ivici takta. Taj uslov se detektuje u procesu i poziva se metoda ***write_to_monitor***, unutar objekta ***result_monitor.*** Dakle rezultati se hendlaju ***result_monitor-***om, a komande ***command_monitor-***om.

```
  always @(posedge clk) begin : rslt_monitor
      if (done)
        result_monitor_h.write_to_monitor(result);
  end : rslt_monitor
```

Pogledajmo sada kako pokazivači na ove objekte rukuju sa ovim objektima. Ako pogledamo u ***command_monitor*** primetićemo da ***command_monitor*** kao što smo videli na slici 1 ima ***uvm_analysis_port***, zatim u ***build_phase*** preuzima pokazivač na bfm iz ***uvm_config_db***; ali sada u narednoj liniji predaje sopstveni hendler ***bfm***-u! Na ovaj način ***command_monitor*** "dobrovoljno" ustupa kontrolu nad sobom bfm-u. U sledećoj liniji kreiramo novi ***analysis_port*** s nazivom ***ap*** i završavamo ***build_phase***.

```
class command_monitor extends uvm_component;
  `uvm_component_utils(command_monitor);

 uvm_analysis_port #(command_s) ap;

 function void build_phase(uvm_phase phase);
   virtual tinyalu_bfm bfm;

   if(!uvm_config_db #(virtual tinyalu_bfm)::get(null, "*","bfm", bfm))
     $fatal("Failed to get BFM");

   bfm.command_monitor_h = this;
```

```
   ap  = new("ap",this);

  endfunction : build_phase
```

Sledi metoda ***write_to_monitor***:

```
  function void write_to_monitor(byte A, byte B, bit[2:0] op);
    command_s cmd;
    cmd.A = A;
    cmd.B = B;
    cmd.op = op2enum(op);
    $display("COMMAND MONITOR: A:0x%2h B:0x%2h op: %s", A, B, cmd.op.name());
    ap.write(cmd);
  endfunction : write_to_monitor
```

Ova metoda ima tri argumenta; A i B operandi su tipa byte, što je poznato, zatim sledi trobitni signal op, on se u telu metode konvertuje funkcijom ***op2enum*** u operaciju našeg tinyalu. Zatim smo popunili strukturu cmd sa dva operanda i operacijom (kao enumerated tipom). Konačno smo ovako popunjenu strukturu upisali na naš ***ap*** metodom ***write***.
Sledi kod metode za konverziju trobitnog signala u naš enumerisani tip op:

```
  function operation_t op2enum(bit[2:0] op);
    case(op)
      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'b111 : return rst_op;
      default : $fatal($sformatf("Illegal operation on op bus: %3b",op));
    endcase // case (op)
  endfunction : op2enum
```

najzad sledi konstruktor naše klase koja je podsećamo se ***naslednica uvm_component klase***:

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

endclass : command_monitor
```

U primeru smo koristili ***command_s*** , strukuturu, koja je definisane u našem ***tinyalu_pkg*** a sve sa ciljem da grupišemo operande i komandu koju smo primili.
Ako pogledamo ***tinyalu _pkg*** kod vidimo na koji način je struktura deklarisana.

```
package tinyalu_pkg;
  import uvm_pkg::*;
`include "uvm_macros.svh"

    typedef enum bit[2:0] {no_op  = 3'b000,
                           add_op = 3'b001,
                           and_op = 3'b010,
                           xor_op = 3'b011,
                           mul_op = 3'b100,
                           rst_op = 3'b111} operation_t;
```

```
typedef struct {
  byte unsigned      A;
  byte unsigned      B;
  operation_t op;
} command_s;
```

Za razliku od ***tinyalu _pkg*** iz prthodnih vežbi, ovde imamo samo novi tip (strukturu) ***command_s*** koji sadrži ***A, B*** i ***op***. Odlučili smo da kapsuliramo sve relevantne informacije vezane za naše komande i to je urađeno u ovom tipu. Ova struktura prolazi kroz ***ap.***

Iz ***write_to_monitor*** i iz ***command_monitor.svh***, uzimamo podatke.

Na sličan način radi i ***result_monitor***. Ovde na identičan način uzimamo handler na ***bfm*** iz ***uvm_config_db*** -a i ***bfm***-u predajemo handler na ***result_monitor***, kroz ***uvm_analysis_port*** prebacivaćemo promenljivui tipa ***shortint***. Vidimo da smo stoga morali da deklarišemo isti tip argumenta i pri deklaraciji uvm_analysis_port-a i pri deklaraciji write_to_monitor metode (ovoga se sećamo iz vežbe broj 15 u kojoj smo se upoznali sa ***analysis_port-***om). ***Shortint*** je 16-bitna promenljiva kao što i zahteva naš rezultat iz ***tinyalu*** bloka. Sve je vidljivo u segment koda datom niže:

```
class result_monitor extends uvm_component;
  `uvm_component_utils(result_monitor);

  uvm_analysis_port #(shortint) ap;

  function void write_to_monitor(shortint r);
    $display ("RESULT MONITOR: resultA: 0x%0h",r);
    ap.write(r);
  endfunction : write_to_monitor
```

Ponovnim pogledom na sliku 1 vidimo da smo obradili ***command_monitor*** i ***result_monitor***, sada obratimo pažnju i na ***scoreboard*** i ***coverage***. Prvo ćemo pogledati ***coverage*** klasu:

```
class coverage extends uvm_subscriber #(command_s);
  `uvm_component_utils(coverage)
```

Primećujemo da ***coverage*** nasleđuje ***uvm_subscriber*** i kažemo da ovaj ***subscriber*** prima argument ***command_s*** što predstavlja vezu između coverage klase i ***command_monitor*** klase. Monitor prosleđuje željeni podatak (u ovom slučaju strukturu tipa ***command_s*** svojim pratiocima (subscriber-ima).

Dalje imamo write metodu, koja naravno mora imati isti argument kao i ***subscriber*** klasa u kojoj je deklarisana.

```
function void write(command_s t);
    A = t.A;
    B = t.B;
    op_set = t.op;
    op_cov.sample();
    zeros_or_ones_on_ops.sample();
  endfunction : write
```

Vidimo da poziv ove metode preuzima ***t.A, t.B*** i ***t.op***, i ove argumente sa operacijom "sempluje" u ***cover_group***-u op_cov i ***cover_group***-u po imenu ***zeros_or_ones_on_ops***.
Slično ovome treba da pogledamo i ***scoreboard.svh, scoreboard*** je specifičniji zato što je u ulozi pratioca (subscriber-a) ***result_monitor***-a, a pored toga potrebno je da ima informaciju i o operandima i komandi, dakle potrebna mu je informacija koja se čuva u ***command_s***. Ovo rešavamo tako što u ovoj klasi dodajemo ***uvm_tlm_analysis_fifo*** koji predstavljaUVM objekat tipa FIFO i u njega ćemo čuvati ***command_s*** podatke po redosledu pristizanja.

```
class scoreboard extends uvm_subscriber #(shortint);
  `uvm_component_utils(scoreboard);

  uvm_tlm_analysis_fifo #(command_s) cmd_f;

  function void build_phase(uvm_phase phase);
    cmd_f = new ("cmd_f", this);
  endfunction : build_phase

  function void write(shortint t);
    shortint predicted_result;
    command_s cmd;
    cmd.op = no_op;
    do
     if (!cmd_f.try_get(cmd)) $fatal(1, "No command in self checker");
    while ((cmd.op == no_op) || (cmd.op == rst_op));

    case (cmd.op)
      add_op: predicted_result = cmd.A + cmd.B;
      and_op: predicted_result = cmd.A & cmd.B;
      xor_op: predicted_result = cmd.A ^ cmd.B;
      mul_op: predicted_result = cmd.A * cmd.B;
    endcase // case (op_set)

    if (predicted_result != t)
     $error (
   "FAILED: A: %2h  B: %2h  op: %s actual result: %4h   expected: %4h",
       cmd.A, cmd.B, cmd.op.name(), t,  predicted_result);
  endfunction : write


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

endclass : scoreboard
```

U ***build_phase*** kreiramo novu instancu ***analysis_fifo*** za sebe, njeno povezivanje na izvor komandnih informacija obradićemo u env bloku, za sad pretpostavimo da je sadržaj ovog FIFO validan. Naša ***write*** metoda u kojoj radimo sav posao; ima isti argument ***t*** tipa ***shortint*** isto kao sama ***scoreboard*** klasa. Biće pozvana kada izvor informacija upiše novi rezultat na adekvatan ***analysis_port***, zatim pokrećemo petlju, ***cmd_f*** iščitava novu FIFO vrednost, čeka

sve dok se ne pojavi operacija različita od ***no_op*** ili ***rst_op***, zatim zavisno od operacije koja je iščitana proračunava ***predicted_result*** i na kraju proverava da li je rezultat upisan kao argument ove ***write*** metode jednak očekivanom.

Sve ovo zajedno je povezano je u ***env***, bloku:

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

  random_tester    random_tester_h;
  coverage   coverage_h;
  scoreboard scoreboard_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);
    random_tester_h    = random_tester::type_id::create("random_tester_h",this);
    coverage_h  =  coverage::type_id::create ("coverage_h",this);
    scoreboard_h = scoreboard::type_id::create("scoreboard_h",this);
    command_monitor_h   = command_monitor::type_id::create("command_monitor_h",this);
    result_monitor_h= result_monitor::type_id::create("result_monitor_h",this);


  endfunction : build_phase

  function void connect_phase(uvm_phase phase);

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

  endfunction : connect_phase

endclass
```

U našem ***env*** objektu imamo ***random_tester, coverage, scoreboard, command_monitor*** i ***result_monitor***, njih u ***build_phase*** kreiramo i dodeljujemo adekvatnim hendlerima. U ***connect_phase***, koju pozivamo nakon ***build_phase,*** povezujemo ***analysis_port-ove*** sa adekvatnim ***analysis_export***–ima, odnosno u ovom primeru vidimo kako povezujemo na sličan način i naš ***analysis_fifo*** objekat, unutar postojećeg scoreboard-a. Na taj način omogućili smo ***scoreboard*** bloku da prati rezultat direktno, a ***command_s*** upotrebom FIFO tehnike.
Dakle uspeli smo da pored direktne veze izvor-pratilac unutar ***scoreboard***-a obezbedimo informacije koje su nam neophodne putem našeg ***analysis_fifo*** objekata.
Ovde je potrebno otvoriti pitanje realnog vremena, odnosno potrebno je da imamo na umu sledeće:

- Vremenski trenutak u kome random_tester generiše operande i komandu prednjači trenutku u kome DUT formira rezultat. Stoga imamo razdvajanje početneg seta informacija koje se pakuju u ***command_s*** jer se one generišu ranije.
- Vremenski trenutak u kome DUT formira rezultat tražene operacije generiše se kasnije. Otuda odluka da se u našem sistemu prilagodimo ovim real-time okvirima.

- Obzirom da smo u našem ***scoreboard*** bloku želeli da ispratimo koji rezultat smo dobili, ali svakako i da znamo zahtevane operande i operaciju, prirodno je da prethodno formirane ***command_s*** podatke složimo u adekvatan FIFO koji "priključujemo" na ***command_monitor*** izvor. Na ovaj način obezbeđujemo da će unutar FIFO bloka postojati aktuelni operandi i zahtevana operacija u trenutku kada nam kroz direktni ***analysis_port*** pristigne rezultat. Pri pristizanju rezultata kroz kanal izvor-pratilac, iščitavamo iz FIFO aktuelne operande i operaciju i finalno proveravamo rezultat.