Univerzitet u Nišu

ELEKTRONSKI FAKULTET

Postupak kreiranja i testiranja valid-ready UVC-a

Seminarski rad
Predmet: Metodologija u verifikaciji
Studijski program: Elektronika i mikrosistemi

Mentor: Student:

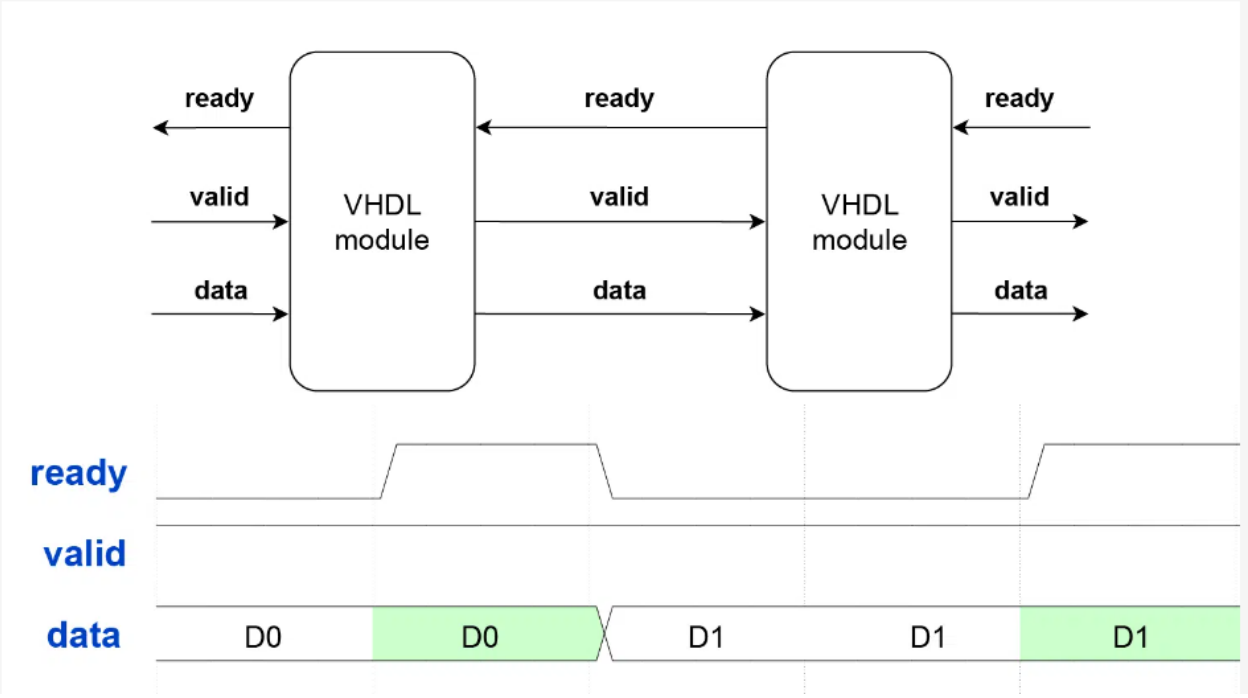
Prof. dr Miona Andrejević Stošović Aleksandar Pantović, br.ind. 1593

Niš, jun 2024. godina

Sadržaj

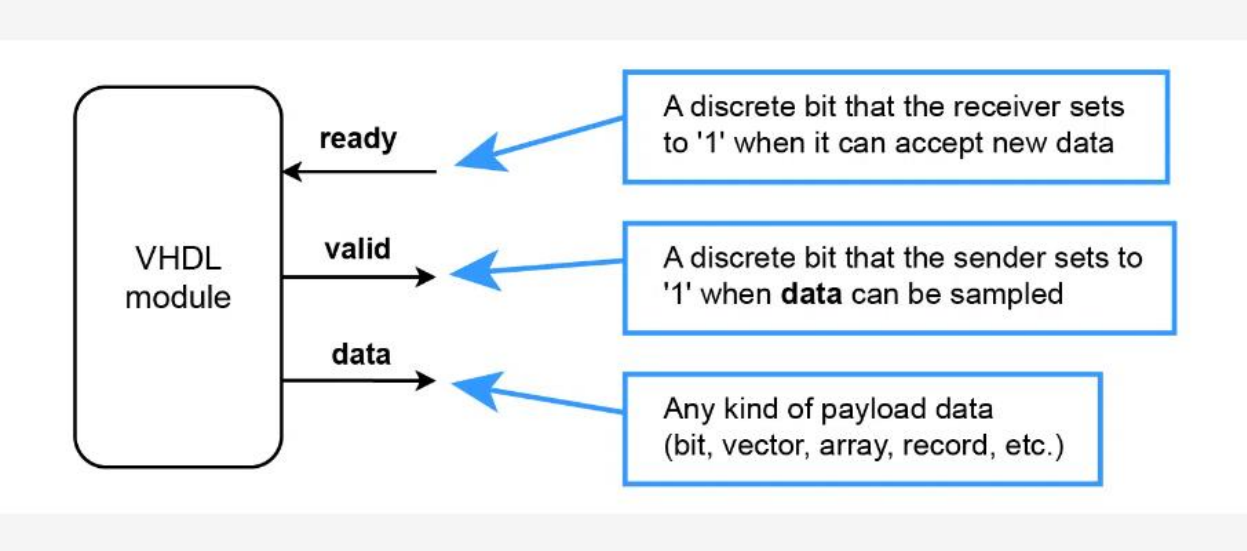
Valid-ready protokol

Valid-Ready protokol je uobičajen hardverski protokol za slanje podataka između dva uređaja. Obezbeđujući kontrolu protoka sa samo dva kontrolna signala. Pravila su jednostavna: prenos podataka se dešava samo kada su i "ready" i "valid" **HIGH** tokom istog takta.



Slika 1. Valid-ready protokol

Slika 2. prikazuje VHDL modul sa jednom izlaznom magistralom koja koristi **ready/valid** handshaking (metoda komunikacije između dva uređaja). Dok je signal ready vezan za prijemnik da ograniči protok podataka ka njemu, predajnik kontroliše signale valid i data. Oba učesnika mogu regulisati brzinu prenosa podataka, a transferi se dešavaju samo kada su oba saglasna.



Slika 2. Modul sa valid-ready protokolom

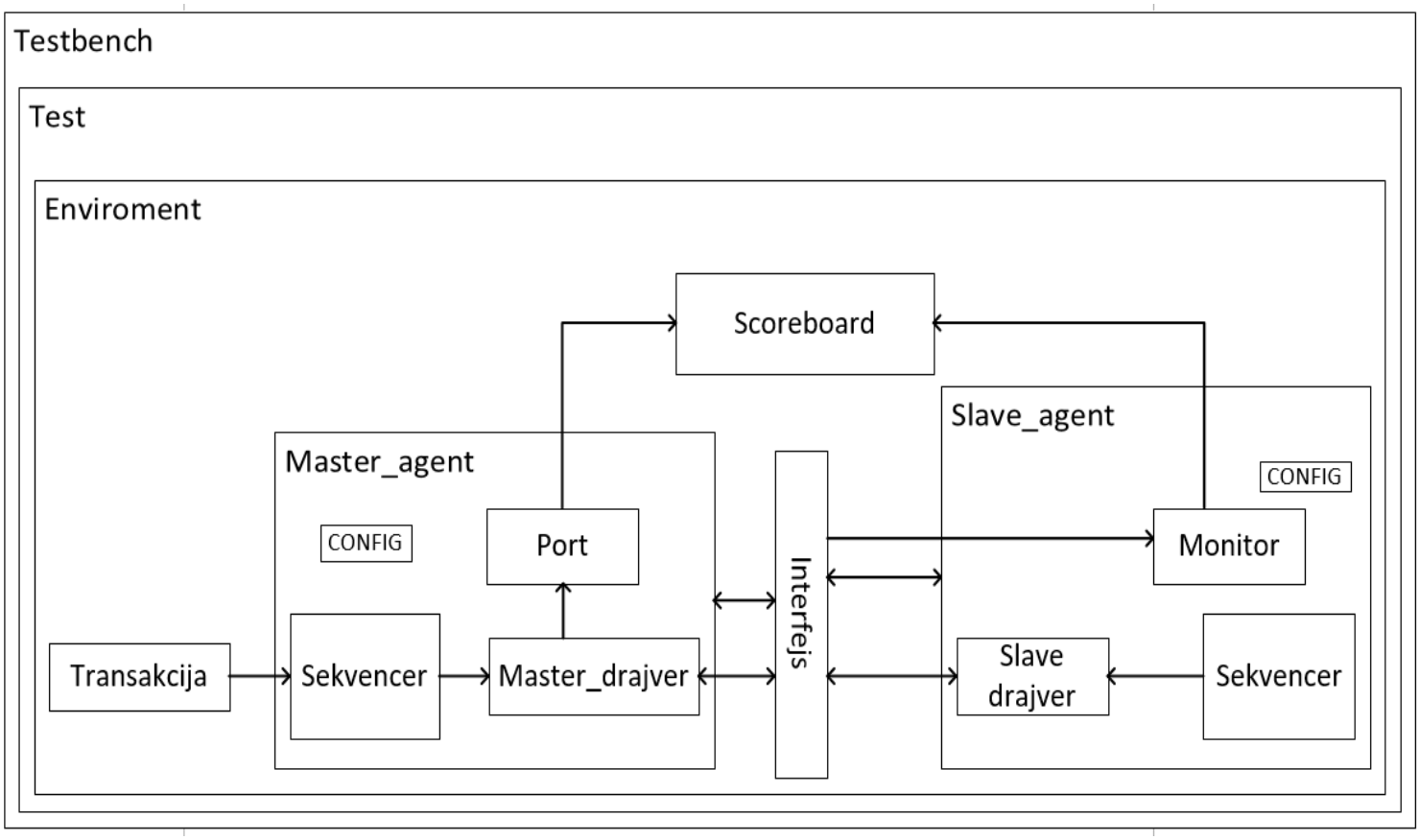
Arhitektura verifikacionog okruženja

Arhitektura verifikacionog okruženja za master-slave komunikaciju zasnovanu na valid-ready protokolu uključuje nekoliko ključnih komponenti i slojeva. Ovo okruženje je obično implementirano koristeći UVM (Universal Verification Methodology), koji pruža standardizovane i fleksibilne metode za verifikaciju složenih digitalnih dizajna. Komponente koje je potrebno realizovati da bi se verifikovala komunikacija između master i slejv uređaja bazirana na valid-ready protokolu obuhvata:

- **Transakcija** (vr_transaction) - Transakcije definišu podatke koji se razmenjuju između komponenti.
- **Sekvenca** (vr_sequence) - Sekvence generišu nizove transakcija koje se šalju drajverima.
- **Drajveri** (vr_master_driver, vr_slave_driver) - Drajveri uzimaju transakcije iz sekvencera i pokreću. Ovde imamo master_driver i slave_driver.

Master_driver šalje podatke na interfejs i postavlja valid signal, sa druge strane slave_driver postavlja ready signal i prima podatke.

- **Monitori** (vr_master_slave_monitor) - Monitori pasivno prate signalne linije i prikupljaju transakcije za analizu.
- **Scoreboard** (vr_scoreboard) - Scoreboard upoređuje transakcije koje dolaze iz monitora mastera i slejva kako bi verifikovao ispravnost podataka.
- **Agenti** (vr_master_agent, vr_slave_agent) - Agenti su odgovorni za generisanje, pokretanje i praćenje transakcija. Agent se sastoji od: sekvencera (u ovoj realizaciji koristi se uvm_sequencer), drajvera i monitora.
- **Enviroment** (vr_env) - Enviroment je glavna komponenta koja instancira sve agente i scoreboard.
- **Test** (vr_test) - Test je odgovoran za konfiguraciju, generisanje i pokretanje testova.
- **Testbench** (tb) - To je skup komponenti, uključujući sekvencere, drajvere, monitore, agente i ostale komponente koje zajedno simuliraju ili emuliraju rad hardverskog dizajna. Glavna uloga testbench-a je da generiše ulazne podatke, upravlja simulacijom, prikuplja rezultate i proverava ispravnost izlaznih podataka iz dizajna.
- **Interfejs** (master_slave_interface) - Interfejs je komponenta koja služi kao komunikacijski kanal između testbench-a i dizajna koji se verifikuje. U ovoj realizaciji nemamo dizajn. Definiciju interfejsa smeštamo u fajlu design.sv
- **Konfiguracioni fajl** (vr_config.svh) – Konfiguracioni fajl služi za centralizovano definisanje i podešavanje konfiguracionih parametara i makroa koji se koriste u verifikacionom okruženju. Ovaj fajl obično sadrži sve potrebne konfiguracije.



Slika 3. Arhitektura verifikacionog okruženja

Transakcija

(vr_transaction.svh)

Ovaj kod predstavlja definiciju SystemVerilog klase pod nazivom vr_transaction, koja nasleđuje uvm_sequence_item iz UVM (Universal Verification Methodology) biblioteke. Ova klasa se koristi za modelovanje transakcija u verifikacionom okruženju. Klasa vr_transaction nasleđuje uvm_sequence_item, što znači da će imati sve atribute i metode definisane u uvm_sequence_item klasi, plus dodatne koje ćemo definisati unutar vr_transaction. Unutar ove klase definišemo polja:

- *rand bit [31:0] data;* - Ovaj atribut predstavlja 32-bitni podatak koji može biti nasumično generisan.

- *rand int randomDelay;* - Ovaj atribut predstavlja celobrojnu vrednost koja može biti nasumično generisana i koja će biti korišćena za simulaciju nasumičnog kašnjenja.

Unutar koda definišemo makroe uvm_object_utils_begin (vr_transaction) i uvm_object_utils_end - Ovi makroi generišu osnovne funkcionalnosti potrebne za UVM objekte, kao što su funkcije za kopiranje, poređenje i pakovanje/raspakovanje objekata, uvm_field_int (data, UVM_ALL_ON) - Ovaj makro registruje polje data kao polje koje može biti automatski upravljano u okviru UVM mehanizama. Da bi ograničili polje randomDelay u opsegu od 0 - 5, moramo dodati constaint blok. Ovaj blok definiše da vrednost randomDelay mora biti između 0 i 5, uključujući ove vrednosti. Ograničenja se koriste za kontrolu nasumično generisanih vrednosti, osiguravajući da budu u određenim granicama.

Konstruktor klase vr_transaction poziva konstruktor nadklase uvm_sequence_item sa zadatim imenom. Konstruktor je funkcija koja se poziva prilikom kreiranja novog objekta klase. U ovom slučaju, on inicijalizuje ime transakcije.

```
class vr_transaction extends uvm_sequence_item;


  // Podatak koji salje master
  rand bit [31:0] data;
  rand int randomDelay;

  //  Ovo je makro koji generiše nekoliko korisnih funkcija za datu klasu
  `uvm_object_utils_begin(vr_transaction)
    `uvm_field_int(data, UVM_ALL_ON)
  `uvm_object_utils_end

  constraint valid_randomDelay {
    randomDelay inside {[0:5]};
  }

  // Konstruktor
  function new(string name = "vr_transaction");
    super.new(name);
  endfunction

endclass
```

Sekvenca

(vr_sequence.svh)

Klasu pod nazivom vr_sequence, koja nasleđuje uvm_sequence parametarski zadatu sa vr_transaction. Ova klasa se koristi za generisanje sekvenci transakcija. Klasa vr_sequence nasleđuje uvm_sequence, pri čemu je tip sekvence definisan kao vr_transaction. Ovo znači da će sekvenca raditi sa objektima tipa vr_transaction. Makro u ovoj klasi `uvm_object_utils(vr_sequence) generiše osnovne funkcionalnosti potrebne za UVM objekte, kao što su funkcije za kopiranje, poređenje i pakovanje/raspakovanje objekata za klasu vr_sequence.

- *virtual task body()* - Ovo je glavna funkcija (task) koja definiše ponašanje sekvence. Funkcija je deklarisana kao virtual, što znači da može biti redefinisana u podklasama.
- *req = vr_transaction::type_id::create("req")* - Kreira novi objekat tipa vr_transaction sa imenom req.
- *start_item(req)* - Signalizira početak transakcije. Ova funkcija postavlja stanje sekvence i priprema transakciju za slanje.
- *if (!req.randomize()) begin ... end* - Pokušava da randomizuje vrednosti unutar objekta req. Ako randomizacija ne uspe, generiše fatalnu grešku sa porukom "Error randomizing sequence".
- *finish_item(req)* - Signalizira završetak transakcije. Ova funkcija označava da je transakcija završena i da se može dalje obrađivati.

```
class vr_sequence extends uvm_sequence #(vr_transaction);

  `uvm_object_utils(vr_sequence)

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

  virtual task body();
```

```
    // Kreiranje transakcije
    req = vr_transaction::type_id::create("req");

    // Početak transakcije
    start_item(req);

    // Randomizacija transakcije
    if (!req.randomize()) begin
      `uvm_fatal("SEQ", "Error randomizing sequence")
    end

    // Završetak transakcije
    finish_item(req);
  endtask
endclass
```

Drajveri

(vr_master_driver.svh)

Klasa vr_master_driver, koja nasleđuje uvm_driver parametarski zadatu sa vr_transaction. Ova klasa se koristi za upravljanje transakcijama u master komponenti verifikacionog okruženja. Klasa vr_master_driver nasleđuje uvm_driver, pri čemu je tip drajvera definisan kao vr_transaction. Ovo znači da će drajver raditi sa objektima tipa vr_transaction. Makro `uvm_component_utils(vr_master_driver) - generiše osnovne funkcionalnosti potrebne za UVM komponente, kao što su funkcije za registraciju, kopiranje, poređenje i pakovanje/raspakovanje objekata za klasu vr_master_driver. Atribute klase koje ovde trebamo imati su:

- *virtual interface master_slave_interface msi* - Virtualni interfejs za komunikaciju sa Device Under Test (služi sa komunikaciju između mastera i slejva.
- *vr_transaction sampled_driver_item* - Promenljiva za čuvanje transakcije koja je uzorkovana.
- *uvm_analysis_port#(vr_transaction) ana_port* - Analizni port za slanje transakcija na scoreboard.
- *vr_config cfg* - Konfiguracija koja se koristi za testiranje.

Unutar build_phase čija je uloga da se postavljaju svi potrebni parametri za komponentu. Prva if petlja pokušava da dobije virtualni interfejs msi iz konfiguracione baze podataka. Ako to ne uspe, generiše fatalnu grešku. Druga if petlja pokušava da dobije konfiguraciju cfg iz konfiguracione baze podataka. Ako to ne uspe, generiše fatalnu grešku.

Unutar run_phase definišemo slanje podataka na osnovu valid-ready protokola. Pritom moramo voditi računa i da pošaljemo podatke preko analysis porta, pre nego što se drajvuju. Ovo radimo iz razloga da možemo da poredimo u scoreboard da li drajver lepo drajvuje, a sa druge strane da li monitor monitoruje kako treba. Celokupnu logiku valid-ready protokola možemo videti u kodu.

```
class vr_master_driver extends uvm_driver #(vr_transaction);

  `uvm_component_utils(vr_master_driver)

  // Virtualni interfejs za komunikaciju s DUT-om
  virtual interface master_slave_interface msi;

  vr_transaction sampled_driver_item;

  // Analysis port za slanje transakcija na scoreboard
   uvm_analysis_port#(vr_transaction) ana_port;

  vr_config cfg;

  // Konstruktor
    function new(string name = "vr_master_driver", uvm_component parent);
    super.new(name, parent);

        ana_port = new("analysis_port", this);

  endfunction

  // Postavljanje virtualnog interfejsa i konfiguracionog fajla
  function void build_phase(uvm_phase phase);
   super.build_phase(phase);
```

```
          if(!uvm_config_db#(virtual  master_slave_interface)::get(this,  "",
"vif", msi)) begin
            `uvm_fatal("MASTER  DRIVER",  "Could  not  retreive  the  interface
handle from tb.")
        end
          if(!uvm_config_db#(vr_config)::get(this, "", "cfg", cfg)) begin
            `uvm_fatal("MASTER_DRIVER", "Could not retreive the configuration
handle from test.")
            end
      endfunction

        // Izvršavanje transakcije
      task run_phase(uvm_phase phase);

        super.run_phase(phase);
        sampled_driver_item  =  vr_transaction::type_id::create("sampled_item",
this);

        msi.valid <= 0;

        wait(msi.rst == 0);
          `uvm_info("Master-driver RST","",UVM_LOW)
          // Petlja za slanje svih transakcija
          forever begin
            // Čekanje na transakciju od sekvencera
            seq_item_port.get_next_item(req);

            //usaglasavanje sa clockom
            @(posedge msi.clk);

            //kasnjenje transakcije
            repeat(req.randomDelay) begin
              @(posedge msi.clk);
            end

            `uvm_info("Master-driver CLK","",UVM_LOW)
            //setovanje valid signala
            msi.valid = 1'b1;
            //slanje podatka pre drajvovanja, ukoliko je postavljeno u config
            if(cfg.uvc_test_mode) begin

              sampled_driver_item.data = req.data;
```

```
            ana_port.write(sampled_driver_item);
          end
          //slanje podataka na interfejs
          msi.data = req.data;

          while (!msi.ready) begin
            @(posedge msi.clk);
          end

          @(posedge msi.clk);
          //reset valid signala
          msi.valid = 1'b0;

          `uvm_info("MASTER_DRIVER", $sformatf("DELAY=%0d",req.randomDelay),
UVM_LOW)

          // Završetak transakcije
          `uvm_info("ZAVRSETAK","",UVM_LOW)
          seq_item_port.item_done();
        end
      endtask
    endclass
```

(vr_slave_driver.svh)

Klasu pod nazivom vr_slave_driver, koja nasleđuje uvm_driver parametarski zadatu sa vr_transaction. Ova klasa se koristi za upravljanje transakcijama u slave komponenti verifikacionog okruženja. Princip pisanja koda je isti, međutim ovde imamo logiku pisanja sa slave strane. U run_phase imamo opisan ovaj postupak.

```
    class vr_slave_driver extends uvm_driver #(vr_transaction);

      `uvm_component_utils(vr_slave_driver)

      // Virtualni interfejs za komunikaciju s DUT-om
      virtual interface master_slave_interface msi;

      // Instanca config fajla
       vr_config cfg;

      // Konstruktor
        function new(string name = "vr_slave_driver", uvm_component parent);
        super.new(name, parent);
```

```
        endfunction

        // Metoda za postavljanje virtualnog interfejsa
        function void build_phase(uvm_phase phase);
          super.build_phase(phase);
          if(!uvm_config_db #(virtual master_slave_interface)::get(this, "",
"vif", msi))
            `uvm_fatal("NOINTF", {"Interface not defined for: ",
get_full_name()})

          if(!uvm_config_db#(vr_config)::get(this, "", "cfg", cfg)) begin
            `uvm_fatal("SLAVE_DRIVER", "Could not retreive the configuration
handle from test.")
            end
        endfunction

        // Metoda za izvršavanje transakcije
         task run_phase(uvm_phase phase);

          super.run_phase(phase);
        // Inicijalizujemo ready na 0
          msi.ready <= 0;

          forever begin

            seq_item_port.get_next_item(req);

        // Proveravamo always_ready signal. Ako je postavljen u config, onda je
ready uvek jednak 1
            if(cfg.always_ready) begin
              wait (msi.valid==1);
              msi.ready <= 1;
          //  msi.data_out <= msi.data; //opciono da bi videli podatak na wave
            end

            else begin
            // Čekanje na signal valid od DUT-a
            wait (msi.valid==1);
             @(posedge msi.clk);
            // Slanje aktivnog ready signala
            msi.ready <= 1;
            // msi.data_out <= msi.data; //opciono da bi videli podatak na wave
```

```
        repeat(req.randomDelay) begin
        @(posedge msi.clk);
        end


        // Postavljanje ready signala na nulu
        msi.ready <= 0;

        // Završavanje transakcije

        end
        seq_item_port.item_done();
      end
    endtask
endclass
```

Monitori

(vr_master_slave_monitor.svh)

Klasa pod nazivom vr_master_slave_monitor, koja nasleđuje uvm_monitor. Ova klasa se koristi za praćenje i prikupljanje podataka sa master-slave interfejsa u verifikacionom okruženju. Unutar ove klase imamo atribute:

- *virtual interface master_slave_interface msi* - Virtualni interfejs za komunikaciju sa DUT-om.
- *uvm_analysis_port#(vr_transaction) analysis_port* - Analizni port za slanje transakcija na scoreboard.
- *vr_transaction sampled_item* - Promenljiva za čuvanje uzorkovane transakcije.

Konstruktor klase vr_master_slave_monitor poziva konstruktor nadklase uvm_monitor sa zadatim imenom i roditeljskom komponentom. Takođe, inicijalizuje analizni port analysis_port.

Run_phase je task koji sadrži glavni kod za praćenje i prikupljanje podataka, sampled_item se kreira kao novi objekat vr_transaction, forever petlja omogućava beskonačno prikupljanje podataka, @(negedge msi.clk) čeka na negativnu ivicu takta. Ako su msi.valid i msi.ready signali postavljeni na 1, prikuplja podatke sa interfejsa, sampled_item.data = msi.data postavlja uzorkovane podatke. Funkcija analysis_port.write(sampled_item) šalje uzorkovane podatke na scoreboard preko porta.

```
      class vr_master_slave_monitor extends uvm_monitor;

        // Virtualni interfejs za komunikaciju sa DUT-om
        virtual interface master_slave_interface msi;


          `uvm_component_utils(vr_master_slave_monitor)

        // Analysis port za slanje transakcija na scoreboard
          uvm_analysis_port#(vr_transaction) analysis_port;

          vr_transaction sampled_item;

        // Konstruktor
          function new(string name = "vr_master_slave_monitor", uvm_component
parent);
          super.new(name, parent);
          analysis_port = new("analysis_port", this);

        endfunction

        // Metoda za postavljanje virtualnog interfejsa
        function void build_phase(uvm_phase phase);
          super.build_phase(phase);
          if(!uvm_config_db  #(virtual  master_slave_interface)::get(this,  "",
"vif", msi))
            `uvm_fatal("MASTER MONITOR", {"Master-slave interface not defined
for: ", get_full_name()})
        endfunction

        // Metoda za pokretanje monitora
        task run_phase(uvm_phase phase);
```

```
    // Instanciranje transakcije za prikupljene podatke
    sampled_item = vr_transaction::type_id::create("sampled_item", this);

    // Beskonačna petlja za prikupljanje podataka
    forever begin
      // Čekanje na pozitivan brid takta
      @(negedge msi.clk);

      // Prikupljanje podataka iz DUT-a
      if (msi.valid == 1'b1 && msi.ready == 1'b1) begin
        // Prikupljanje podataka
        @(negedge msi.clk);
        sampled_item.data = msi.data;

        // Slanje prikupljenih podataka na scoreboard kroz analysis port
        analysis_port.write(sampled_item);
      end
    end
  endtask
endclass
```

Agenti

(vr_master_agent.svh)

Klasa pod nazivom vr_master_agent, koja nasleđuje uvm_agent. Ova klasa predstavlja agenta u UVM okruženju koji se koristi za generisanje, slanje i prikupljanje transakcija za master komponentu. Atribute koje ovde možemo primetiti su:

- *uvm_analysis_port #(vr_transaction) ap* - Analizni port agenta za slanje transakcija na scoreboard.
- *virtual interface master_slave_interface msi* - Virtualni interfejs za komunikaciju sa DUT-om.
- *vr_master_driver driver* - Drajver agenta.
- *uvm_sequencer #(vr_transaction) sequencer* - Sekvencer agenta.

U ovom kodu možemo primetiti dve faze. Build_phase je faza u kojoj se instanciraju sve potrebne komponente agenta, ona instancira drajver i sekvencer. Connect_phase je faza u kojoj se povezuju različite komponente agenta. U ovom slučaju povezujemo drajver i sekvencer, kao i analysis port drajvera sa analysis portom agenta.

```
class vr_master_agent extends uvm_agent;
  `uvm_component_utils(vr_master_agent)


  uvm_analysis_port #(vr_transaction) ap; // Analysis port agenta

  virtual interface master_slave_interface msi;

// Komponente agenta
  vr_master_driver driver;
  uvm_sequencer #(vr_transaction) sequencer;

  // Konstruktor
    function new(string name="vr_master_agent", uvm_component parent);
    super.new(name, parent);
  endfunction

  // Build faza - instanciranje komponenti
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    // Instanciranje drajvera i sekvencera
    driver = vr_master_driver::type_id::create("driver", this);
    sequencer                                                    =
uvm_sequencer#(vr_transaction)::type_id::create("sequencer", this);
    ap = new("ap", this); // Inicijalizacija analysis porta agenta

  endfunction

  // Connect faza - povezivanje sekvencera i drajvera
  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

    driver.seq_item_port.connect(sequencer.seq_item_export);
    driver.ana_port.connect(ap); //povezivanje portova
  endfunction
endclass
```

(vr_slave_agent.svh)

Klasa vr_slave_agent definiše UVM agenta koji sadrži drajver, sekvencer i monitor, kao i konfiguraciju agenta i virtualni interfejs za komunikaciju sa DUT-om. Ova klasa koristi UVM makroe za generisanje osnovnih funkcionalnosti, definiše konstruktor za inicijalizaciju agenta, instancira drajver, sekvencer i monitor tokom faze izgradnje (build_phase), povezuje sekvencer i drajver tokom faze povezivanja (connect_phase).

Unutar build_phase imamo if(!uvm_config_db#(vr_config)::get(this, "", "cfg", cfg)) begin ... end - Pokušavamo da dobijemo konfiguraciju cfg iz konfiguracione baze podataka. Ako to ne uspe, generišemo fatalnu grešku. Uslov if(cfg.is_active) begin ... end nam kaže da ukoliko imamo signal is_active na 1, to znači da je agent konfigurisan kao aktivan. Instanciranje monitora se radi uvek, bez obzira na to da li je agent aktivan ili pasivan.

```
class vr_slave_agent extends uvm_agent;
  `uvm_component_utils(vr_slave_agent)

  // Deklaracija komponenata agenta
  vr_slave_driver driver;
  vr_master_slave_monitor monitor;
  uvm_sequencer #(vr_transaction) sequencer;

  vr_config cfg;

  virtual interface master_slave_interface intf;

  // Konstruktor
    function new(string name="vr_slave_agent", uvm_component parent);
    super.new(name, parent);
  endfunction

  // Build phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

   // Config ucitavamo pre nego sto instanciramo komponente
    if(!uvm_config_db#(vr_config)::get(this, "", "cfg", cfg)) begin
      `uvm_fatal("SLAVE_AGENT", "Could not retreive the configuration
handle from test.")
      end
```

```
          // Drajver i sekvencer instanciramo samo ako je agent konfigurisan
kao aktivan
if(cfg.is_active) begin
driver = vr_slave_driver::type_id::create("driver", this);

sequencer=uvm_sequencer#(vr_transaction)::type_id::create("sequencer", this);

      end
          // Monitor instanciramo i kada je agent aktivan i kada je pasivan
monitor = vr_master_slave_monitor::type_id::create("monitor", this);

        endfunction

        // Connect phase
        function void connect_phase(uvm_phase phase);
          super.connect_phase(phase);

          // Povezivanje sekvencera i drajvera
          driver.seq_item_port.connect(sequencer.seq_item_export);
        endfunction
      endclass
```

Scoreboard

(vr_scoreboard.svh)

Klasa pod nazivom vr_scoreboard, koja nasleđuje uvm_component. Ova klasa predstavlja scoreboard komponentu u UVM okruženju, koja služi za proveru ispravnosti transakcija između master i slave komponenti. Ova klasa sadrži promenljive za čuvanje master1_trans (promenljiva za čuvanje transakcija primljenih od master komponente) i slave_trans (promenljiva za čuvanje transakcija primljenih od slave komponente). Za povezivanje sa monitorima imamo dva analysis porta, master1_trans_port - analysis port za povezivanje sa monitorom master komponente i slave_trans_port - analysis port za povezivanje sa monitorom slave komponente. Build_phase je faza u kojoj se mogu instancirati dodatne komponente ako je potrebno. U ovom slučaju, ova faza ne radi ništa osim poziva build_phase nadklase.

Funkcija function void write_mm(vr_transaction t_m1) se poziva kada master drajver pošalje transakciju na scoreboard pre nego što je drajvuje. Transakcija se čuva u master1_trans i zatim se poziva funkcija compare_trans za poređenje

transakcija. Sa druge strane, function void write_ss(vr_transaction t_s) se poziva kada monitor slave komponente pošalje transakciju na scoreboard. Transakcija se čuva u slave_trans i zatim se poziva funkcija compare_trans za poređenje transakcija.

Funkcija compare_trans poredi transakcije iz master drajvera i slejv monitora. Ako su obe transakcije dostupne (master1_trans != null && slave_trans != null), proverava se da li su podaci (data) iz transakcija jednaki. Ako jesu, generiše se informativna poruka. Ako nisu, generiše se greška sa detaljima o očekivanim i stvarnim podacima. Nakon poređenja, transakcije se resetuju (master1_trans = null, slave_trans = null).

```
`uvm_analysis_imp_decl(_mm)
`uvm_analysis_imp_decl(_ss)

class vr_scoreboard extends uvm_component;
  `uvm_component_utils(vr_scoreboard)

  // Promenljive za čuvanje transakcija
  vr_transaction master1_trans;
  vr_transaction slave_trans;

  // Analysis port za povezivanje sa monitorima
uvm_analysis_imp_mm #(vr_transaction, vr_scoreboard) master1_trans_port;
uvm_analysis_imp_ss #(vr_transaction, vr_scoreboard) slave_trans_port;

  // Konstruktor
  function new(string name="vr_scoreboard", uvm_component parent);
    super.new(name, parent);
    master1_trans_port = new("master1_trans_port", this);
    slave_trans_port = new("slave_trans_port", this);
  endfunction

  // Build phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);

  endfunction

  // Write funkcija za master 1 transakcije
```

```
        function void write_mm(vr_transaction t_m1);
          master1_trans = t_m1;
       `uvm_info("SCOREBOARD", $sformatf("Received master1 transaction: %0s",
t_m1.sprint()), UVM_LOW)
          compare_trans();
        endfunction

        // Write funkcija za slave transakcije
        function void write_ss(vr_transaction t_s);
          slave_trans = t_s;
      `uvm_info("SCOREBOARD", $sformatf("Received slave transaction: %0s",
t_s.sprint()), UVM_LOW)
          compare_trans();
        endfunction

       function void compare_trans();
          if (master1_trans != null && slave_trans != null) begin
            if (master1_trans.data == slave_trans.data) begin
        `uvm_info("SCOREBOARD", "Master and Slave transactions match", UVM_LOW)
            end else begin
        `uvm_error("DATA_MISMATCH", $sformatf("Expected: %0h, Actual: %0h",
master1_trans.data, slave_trans.data))
            end
            // Reset transakcije nakon poređenja
            master1_trans = null;
            slave_trans = null;
          end
        endfunction
      endclass
```

Test

(vr_test.svh)

Klasa vr_test, koja nasleđuje uvm_test. Ova klasa predstavlja osnovnu UVM test komponentu, koja služi za konfigurisanje i pokretanje test sekvenci. Unutar testa definišemo agente, sekvence, kao i konfiguracioni fajl koji ćemo podesiti u skladu sa zahtevima testiranja (env_master_slave - promenljiva koja predstavlja instancu okruženja koja sadrži master i slave agente, master_seq - promenljiva koja predstavlja sekvencu za master agenta, slave_seq - promenljiva

koja predstavlja sekvencu za slave agenta, vr_config cfg - promenljiva koja predstavlja konfiguracioni objekat za agente).

U build_phase fazi se vrši instanciranje okruženja i konfiguracionog objekta. Konfiguracioni objekat (cfg) se postavlja sa odgovarajućim vrednostima, a zatim se skladišti u UVM bazu podataka pomoću uvm_config_db.

U run_phase fazi se podiže objection (phase.raise_objection(this)), što sprečava završetak faze dok se objection ne spusti. Kreiraju se master i slave sekvence, koje se pokreću istovremeno unutar fork blokova. Sekvence se pokreću za svaki agent u okruženju (env_master_slave). Nakon pokretanja svih sekvenci, čekamo određeno vreme (#1000), zatim spuštamo objection (phase.drop_objection(this)), što omogućava završetak run_phase faze.

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


  // Pokazivac na instance
  vr_env env_master_slave;

  vr_sequence master_seq;
  vr_sequence slave_seq;

  vr_config cfg;

  // Konstruktor
  function new(string name = "vr_test", uvm_component parent);
    super.new(name, parent);
  endfunction

  // Build phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Kreiranje instanci
   env_master_slave = vr_env::type_id::create("env_master_slave", this);
    cfg = vr_config::type_id::create("cfg", this);

    // Konfigurisanje agenata - uvek se vrsi u testu
  cfg.is_active = 1;
```

```
        cfg.always_ready=0;
        cfg.uvc_test_mode=1;

          // Nakon konfigurisanja, cfg objekat saljemo u bazu podataka
          uvm_config_db#(vr_config)::set(this, "*", "cfg", cfg);

        endfunction

        // Run faza

       task run_phase(uvm_phase phase);
          super.run_phase(phase);

          master_seq = vr_sequence::type_id::create("master_seq");
          slave_seq = vr_sequence::type_id::create("slave_seq");

          phase.raise_objection(this);

          // Start master sekvenci

            for (int i = 0; i < 5; i++) begin
              fork
                  begin
          master_seq=vr_sequence::type_id::create($sformatf("master_seq_%0d",
i));
          master_seq.start(env_master_slave.masteragent.sequencer);
                  end

                  begin
          slave_seq  =  vr_sequence::type_id::create($sformatf("slave_seq_%0d",
i));
          slave_seq.start(env_master_slave.slaveagent.sequencer);
                  end
              join_any
              // Timeout da osiguramo da fork ne blokira zauvek
              #250;
          end
           #1000;
          phase.drop_objection(this);
        endtask

      endclass
```

Enviroment

(vr_env.svh)

Klasa vr_env, koja nasleđuje uvm_env. Vr_env predstavlja okruženje (environment) u UVM metodologiji, koje uključuje agente (master i slave), scoreboard i vrši njihovo povezivanje. U build_phase fazi se vrši instanciranje master i slave agenata, kao i scoreboard-a. Instance agenata se kreiraju koristeći create metodu sa odgovarajućim imenima ("masteragent" i "slaveagent") i prosleđuju se referenca na trenutno okruženje (this). Takođe, kreira se i scoreboard sa imenom "sb".

U connect_phase fazi se vrši povezivanje analysis portova agenata sa odgovarajućim implementacijama analysis porta u scoreboard-u masteragent.ap se povezuje sa sb.master1_trans_port, dok se slaveagent.monitor.analysis_port povezuje sa sb.slave_trans_port.

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

  vr_master_agent masteragent;
  vr_scoreboard sb;
  vr_slave_agent slaveagent;

  function new(string name="vr_env", uvm_component parent);
  super.new(name, parent);

  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // Kreiranje agentata
    masteragent = vr_master_agent::type_id::create("masteragent", this);
    slaveagent = vr_slave_agent::type_id::create("slaveagent", this);

    // Kreiranje scoreboard
    sb = vr_scoreboard::type_id::create("sb", this);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);

    // Povezivanje analysis portova na scoreboard
    masteragent.ap.connect(sb.master1_trans_port);
```

```
    slaveagent.monitor.analysis_port.connect(sb.slave_trans_port);
  endfunction
endclass
```

Konfiguracioni fajl

(vr_config.svh)

Klasa vr_config omogućava jednostavno konfigurisanje različitih parametara koji se koriste u UVM testbench-u, omogućavajući fleksibilnost i prilagodljivost u različitim scenarijima verifikacije. Svako polje predstavlja jedan aspekt ponašanja ili konfiguracije koji se može podešavati prema potrebama specifičnog verifikacionog testa.

- Polja (fields):
  - *is_active*: Bit koji označava da li je agent aktivan (1) ili pasivan (0).
  - *always_ready*: Bit koji označava da li je agent uvek spreman (1) ili nije (0).
  - *uvc_test_mode*: Bit koji označava da li je agent u UVC (Universal Verification Component) test režimu (1) ili nije (0).
- uvm_object_utils makro: uvm_object_utils_begin i uvm_object_utils_end: Ovi makroi se koriste da bi se automatski generisale neke od osnovnih funkcionalnosti u UVM metodologiji, kao što su copy, compare, print i pack funkcije. Ovi makroi takođe omogućavaju automatsko generisanje metoda za čitanje i pisanje (get i set) polja objekta.

```
class vr_config extends uvm_object;


  // Opcije za konfiguraciju:
  bit is_active; // Izbor izmedju aktivnog i pasivnog agenta
  bit always_ready;
  bit uvc_test_mode;


  `uvm_object_utils_begin(vr_config)
    `uvm_field_int(is_active, UVM_ALL_ON)
    `uvm_field_int(always_ready, UVM_ALL_ON)
```

```
    `uvm_field_int(uvc_test_mode, UVM_ALL_ON)
    `uvm_object_utils_end

    extern function new(string name = "vr_config");
  endclass

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

Testbench

(vr_testbench.sv)

Testbench demonstrira integraciju UVM metodologije sa SystemVerilog simulacijom. UVM komponente kao što su agenti, sekvenceri, drajveri, monitori, scoreboard, konfiguracija i test su definisani kao SystemVerilog klase i instancirani u okruženju (vr_env). Interfejs msi omogućava komunikaciju sa DUT-om preko clk i reset signala. Simulacija se pokreće, test se izvršava, i waveform se čuva za dalju analizu.

```
`include "uvm_macros.svh"

module tb;

  logic clk;
  logic reset;


  // Ubacivanje UVM biblioteke
  import uvm_pkg::*;

  // Ukljucivanje fajlova verifikacionog okruzenja
  `include "vr_transaction.svh"
  `include "vr_config.svh"
  `include "vr_sequence.svh"
  `include "vr_master_driver.svh"
  `include "vr_slave_driver.svh"
  `include "vr_master_slave_monitor.svh"
```

```
    `include "vr_master_agent.svh"
    `include "vr_slave_agent.svh"
    `include "vr_scoreboard.svh"
    `include "vr_env.svh"
    `include "vr_test.svh"

    master_slave_interface msi(clk,reset);

  //Generisanje CLK
    initial begin
      clk = 0;
      #60;
      forever begin
        clk = ~clk;
        #10;
      end
    end

    // Generisanje reseta
    initial begin
      reset = 1;
      #160;
      reset = 0;
    end

    // Ubacivanje pokazivaca na interfejs u bazu podataka
    initial begin

uvm_config_db#(virtual master_slave_interface)::set(null, "*","vif", msi);
    end

    // Pokretanje testa
    initial begin
      run_test("vr_test");
      end

      // Cuvanje waveform fajla
    initial begin
      $dumpfile("dump.vcd");
      $dumpvars;
      end
  endmodule
```

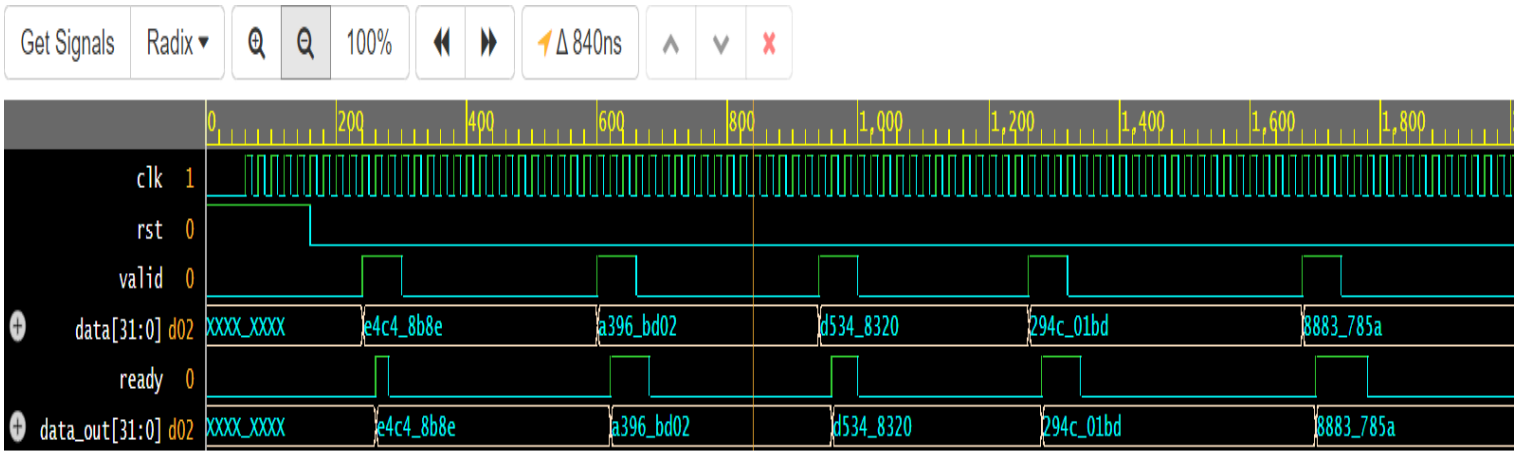
Interfejs

(master_slave_interface)

Interfejs master_slave_interface služi kao standardizovani način komunikacije između testbench-a i DUT-a. Omogućava definisanje zajedničkih signala i podataka koji se koriste za kontrolu i prenos informacija. Ovaj interfejs će biti povezan sa odgovarajućim komponentama (npr. drajverima i monitorima) unutar UVM okruženja kako bi se realizovala funkcionalna i strukturalna verifikacija dizajna.

```
interface master_slave_interface(input logic clk, input logic rst);
  logic valid;
  logic ready;
  logic [31:0] data;
  //logic [31:0] data_out; //dodat signal iz razloga da vidimo razmenu podataka. Ovaj signal treba ubaciti u slejv drajver da bi imali podatke na njemu. Potrebno je skinuti komentare na oba mesta u slejv drajveru
endinterface
```

Testiranje komunikacije mastera i slejva



Slika 4. Testiranje komunikacije