

Funkcionalna verifikacija hardvera

Vežba 6

Rad sa sekvencama i razvoj drajvera

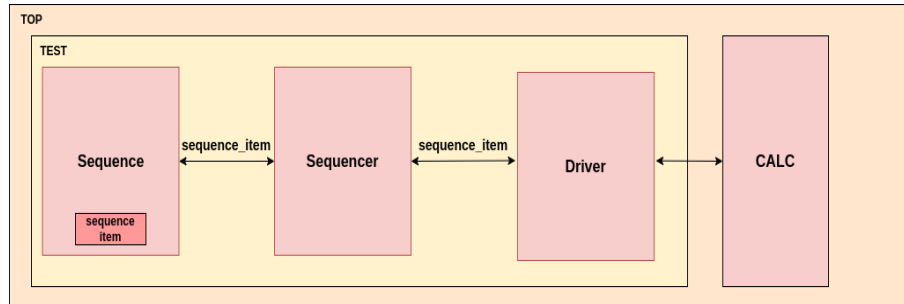
Sadržaj

1	Uvod	4
2	Data (sequence) item	5
3	Sekvence	8
3.1	Struktura	8
3.2	Generisanje stimulusa	8
3.2.1	<code>`uvm_do</code>	9
3.2.2	<code>`uvm_do_with</code>	9
3.2.3	Kontrola	10
3.3	Pokretanje sekvenci	10
4	Sekvencer	12
5	Drajver	13
5.1	API	14
5.1.1	<i>Unidirectional Non-Pipelined</i>	15
5.1.2	<i>Bidirectional Non-Pipelined</i>	16
6	Komunikacija	18
7	Mehanizam završetka testa	20
8	Organizacija fajlova koji čine verifikaciono okruženje	21
8.1	Kompajliranje u Vivadu	22
8.2	Kompajliranje u Cadence Xcelium simulatoru	23
9	Zadaci	25
10	Appendix	26

Ova vežba je posvećena opisu komunikacije između drajvera i sekvencera, opisu sekvenci i *sequence item*-a i opisu raznih mogućnosti UVM-a u pogledu pisanja i korišćenja sekvenci. Dat je i opis mehanizma završetka testa.

1 Uvod

U materijalu za vežbu 5 započeli smo pravljenje UVM verifikacionog okruženja pomoću koga će se verifikovati *calc* digitalni sistem. Prva komponenta koju smo napravili bila je Test klasa i u metodi *main_phase* smo postavljali određene vrednosti na ulaze *calc* digitalnog sistema kako bismo testirali njegovu funkcionalnost, pri čemu smo pristup ulazima digitalnog sistema dobili preko virtuelnog interfejsa. Verifikacija je u potpunosti mogla biti izvršena samo u test klasi, ali to ne bi poštovalo principe UVM metodologije koji nalažu partitionisanje koda na smislene celine(na više komponenti). Iz tog razloga u ovom materijalu uvodimo dodatne UVM klase: *sequence item*, *sequence*, *sequencer* *driver*. Sa ovim promenama verifikaciono okruženje bi trebalo da izgleda kao na slici 1:



Slika 1: Verifikaciono okruženje, vežba 6

Sa prethodne slike se vidi da Sekvenca (*eng. Sequence*) i drajver *eng. Driver* međusobno komuniciraju preko sekvencer komponente (*eng. Sequencer*), pri čemu se preko sekvencera šalje objekat *sequence item* klase u kome se nalaze informacije neophodne drajveru i sekvenci. U nastavku je svaka od klasa detaljnije opisana.

2 Data (sequence) item

Data (sequence) item predstavlja transakciju koja se koristi kao stimulus. Koristi se na mnogim mestima u verifikacionom okruženju uključujući slanje podataka od sekvencera ka drajveru. Slanjem više ovih transakcija ka drajveru, kreira se stimulus na ulazima DUT-a. Sama transakcija treba da sadrži sve informacije potrebne drajveru da ispravno generiše signale na interfejsu, ali može i sadržati neke dodatne podatke koji služe za kontrolu.

Ove transakcije će se implementirati kao klase koje nasleđuju (direktno ili indirektno) `uvm_sequence_item` klasu. Polja u klasi zavise od potreba drajvera, nivoa apstrakcije, ali i potreba ostatka okruženja. Uglavnom se ti podaci mogu podeliti u nekoliko grupa:

- Kontrolna polja - npr. tip transfera, veličina, ...
- Payload polja - sami podaci koji se šalju, u slučaju *calc* dizajna, to bi bili podaci nad kojima treba izvršiti operaciju i tip operacije.
- Konfiguraciona polja - npr. ponašanje prilikom grešaka
- Polja za analizu - pomoćna polja koja olakšavaju analizu, npr. vreme

Sam princip generisanja stimulusa u constrained-random verifikaciji se zasniva na randomizaciji ovih transakcija kako bi se pokrili interesantni scenariji. Samim tim, veliki broj polja je često deklarisan kao `rand`, uz određena ograničenja kako bi se generisale željene transakcije.

Sadržaj transakcija je opisan na primeru transakcije za APB i UART protokol. U ovom primeru se vidi da transakcija može sadržati željeni broj polja, ali i metode koje olakšavaju njeno korišćenje (u primeru UART *seq item*-a postoji metoda za računanje parnosti).

```

`ifndef APB_SEQ_ITEM_SV
`define APB_SEQ_ITEM_SV

parameter ADDR_WIDTH = 32;
parameter RDATA_WIDTH = 32;
parameter WDATA_WIDTH = 32;
typedef enum {
    APB_READ = 0,
    APB_WRITE = 1
} apb_direction_enum;

class apb_seq_item extends uvm_sequence_item;

    // polja
    rand bit [ADDR_WIDTH - 1 : 0] addr;
    rand apb_direction_enum dir;
    rand bit [RDATA_WIDTH - 1 : 0] rdata;
    rand bit [WDATA_WIDTH - 1 : 0] wdata;
    rand int unsigned delay = 0;
    bit error;

    // ograničenja
    constraint c_delay { delay <= 10 ; }

    // UVM factory registracija
    `uvm_object_utils_begin(apb_seq_item)
        `uvm_field_int(addr, UVM_DEFAULT)
        `uvm_field_enum(apb_direction_enum, dir, UVM_DEFAULT)
        `uvm_field_int(rdata, UVM_DEFAULT)
        `uvm_field_int(wdata, UVM_DEFAULT)
        `uvm_field_int(delay, UVM_DEFAULT)
        `uvm_field_int(error, UVM_DEFAULT)
    `uvm_object_utils_end

```

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

endclass

`endif
```

Kod 1: Primer APB transakcije

```
`ifndef UART_SEQ_ITEM_SV
`define UART_SEQ_ITEM_SV

typedef enum bit {GOOD_PARITY, BAD_PARITY} parity_e;
parameter PAYLOAD_WIDTH = 8;
parameter STOP_WIDTH = 8;
parameter ERR_WIDTH = 8;

class uart_seq_item extends uvm_sequence_item;

    rand bit start_bit;
    rand bit [PAYLOAD_WIDTH-1:0] payload;
    bit parity;
    rand bit [STOP_WIDTH-1:0] stop_bits;
    rand bit [ERR_WIDTH-1:0] error_bits;
    rand parity_e parity_type;
    rand int unsigned delay;

    // ogranichenja
    constraint c_delay {delay < 20;}
    constraint c_start_bit {start_bit == 0;}
    constraint c_stop_bits {stop_bits == '1;}
    constraint c_parity_type {parity_type == GOOD_PARITY;}
    constraint c_error_bits {error_bits == 0;}

    // UVM factory registracija
    `uvm_object_utils_begin(uart_seq_item)
        `uvm_field_int(start_bit, UVM_DEFAULT)
        `uvm_field_int(payload, UVM_DEFAULT)
        `uvm_field_int(parity, UVM_DEFAULT)
        `uvm_field_int(stop_bits, UVM_DEFAULT)
        `uvm_field_int(error_bits, UVM_DEFAULT)
        `uvm_field_enum(parity_e, parity_type, UVM_DEFAULT)
        `uvm_field_int(delay, UVM_DEFAULT)
    `uvm_object_utils_end

    // konstruktor
    function new(string name = "uart_seq_item");
        super.new(name);
    endfunction

    // metoda za racunanje parnosti
    function bit calc_parity(int unsigned num_of_data_bits = 8, bit[1:0] parity_mode = 0);
        bit parity;

        if (num_of_data_bits == 6)
            parity = ^payload[5:0];
        else if (num_of_data_bits == 7)
            parity = ^payload[6:0];
        else
            parity = ^payload;

        case(parity_mode[0])
            0: parity = ~parity;
            1: parity = parity;
        endcase
    endfunction

`endif
```

```

    case(parity_mode[1])
      0: parity = parity;
      1: parity = ~parity_mode[0];
    endcase

    if (parity_type == BAD_PARITY)
      return ~parity;
    else
      return parity;
    endfunction

    // parnost se racuna posle randomizacije, na osnovu dodeljenih vrednosti
    function void post_randomize();
      parity = calc_parity();
    endfunction : post_randomize
endclass

`endif

```

Kod 2: Primer UART transakcije

Nasledivanjem iz *uvm_sequence_item* klase i korišćenjem raznih UVM field makroa, omogućava se korišćenje mnogobrojnih metoda za rad sa ovom klasom uključujući *print*, *copy*, *compare* itd. Primer ispisa *print* metode je takođe dat ispod. Vidi se da korišćenje ovih metoda može biti veoma korisno i umnogome olakšati *debug*.

```

# -----
# Name           Type           Size   Value
# -----
# uart_frame     uart_frame    -      @366
# start_bit      integral      1      'h0
# payload        integral      8      'hcc
# parity         integral      1      'h1
# stop_bits      integral      2      'h3
# error_bit      integral      4      'h0
# parity_type     parity_e      1      GOOD_PARITY
# delay          integral      32     'd13
# -----

```

3 Sekvence

Sekvence su objekti (ne komponente) koje sadrže logiku generisanja stimulusa. Unutar njih se kreira objekat klase *Sequence item* koji se preko sekvencera prosleđuje drajveru. Kao i većina UVM okruženja, i kod sekvenci postoji preporučeni način upotrebe, kako u pogledu strukture same sekvence, tako i za sam način generisanja stimulusa. U ovom poglavlju je opisan mehanizam generisanja stimulusa, dat je pregled strukture sekvenci i objašnjene često korišćene funkcije.

3.1 Struktura

Sve sekvence u okruženju treba da, direktno ili indirektno, nasleđuju *uvm_sequence* klasu. Pošto sekvence nisu komponente one ne sadrže UVM faze, međutim ipak postoji sličan mehanizam koji garantuje ispravan rad i ispravnu komunikaciju sa drajverom. Tri najčešće korišćene metode su *pre_body()*, *body()* i *post_body()*. Glavna logika treba da se nalazi u *body()* tasku. *Pre_body()* i *post_body()* su taskovi koji se pozivaju pre, odnosno posle *body()* taska. Kao i kod pisanja testova, dobra je praksa da postoji bazna klasa u kojoj se deklarise sekvencer, vrši eventualno podizanje/spuštanje prigovora (*raise / drop objection*, opisano u narednom poglavlju), kao i sve ostale radnje koje će se ponavljati u svim sekvencama.

```
class calc_seq extends uvm_sequence#(calc_seq_item);

  `uvm_object_utils(calc_seq)
  `uvm_declare_p_sequencer(calc_sequencer)

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

  virtual task pre_body();
    // ...
  endtask : pre_body

  // transaction generating logic in body
  virtual task body();
    // calls to uvm_do or uvm_do with macro
    // or start / finish item
    // ...
  endtask : body

  virtual task post_body();
    // ...
  endtask : post_body

endclass : calc_seq
```

sekvence je potrebno parametrizovati tipom *sequence_item*-a koji će se generisati. Ovo, između ostalog, omogućava korišćenje podrazumevanog objekta(*req*) u sekvencama. Pored *factory* registracije (``uvm_object_utils(calc_seq)` u primeru), potrebno je i deklarirati sekvencer koji će se koristiti. Svaka sekvenca se pri pokretanju veže za odgovarajući sekvencer kome se može pristupiti preko pokazivača. Ovaj korak je moguće i izostaviti, ili deklarirati sekvencere na drugi način, međutim najjednostavniji način za naše potrebe je da se deklarise takozvani *p sekvencer* koristeći odgovarajući makro (``uvm_declare_p_sequencer(calc_sequencer)`, gde *calc_sequencer* predstavlja ime korišćenog sekvencera).

3.2 Generisanje stimulusa

Za generisanje transakcije potrebno ispratiti nekoliko koraka:

1. Kreiranje - veoma je bitno kreirati paket koristeći *factory* kako bi se olakšala ponovna upotreba odnosno kako bi se dozvolio *override* metod ukoliko je to potrebno (objašnjeno u narednim vežbama)

2. *Start_item()* - blokirajući poziv koji čeka da se uspostavi veza sa drajverom. Sekvenceru šalje pokazivač na sledeći objekat
3. Randomizacija paketa (polja unutar *sequence item-a*) ukoliko je to potrebno
4. *Finish_item()* - blokirajući poziv koji čeka da drajver završi sa paketom (napomena: *start_item()* i *finish_item()* čekaju na drajver kako bi se osigurao korektan transfer paketa, ali oni ne troše simulaciono vreme)
5. *Get_response()* - opcioni poziv ako je potrebno da drajver pošalje odgovor sekvenceru

Prva četiri koraka su obavezna, dok je peti opcioni i koristi se samo ukoliko za dato okruženje postoji potreba prikupljanja odgovora od drajvera. Logika generisanja stimulusa treba da se nalazi u *body()* tasku, npr:

```
virtual task body();

    calc_seq_item calc_it;

    // prvi korak kreiranje transakcije
    calc_it = calc_seq_item::type_id::create("calc_it");

    // drugi korak — start
    start_item(calc_it);

    // treci korak priprema
    // po potrebi moguće proširiti sa npr. inline ograničenjima
    assert(calc_it.randomize());

    // cetvrti korak — finish
    finish_item(calc_it);

endtask: body
```

Pošto se ova četiri koraka često ponavljaju tj. standardni su za interakciju drajvera i sekvencera, razvijeni su odgovarajući UVM makroi kako bi se izbeglo nepotrebno ponavljanje koda i povećala čitljivost. Iako veoma korisni, makroi ipak uvode neka ograničenja pri korišćenju, tako da je nekad ipak potrebno ručno odraditi ove korake. “UVM Cookbook” ne preporučuje upotrebu makroa, ali su oni česti u praksi.

3.2.1 ``uvm_do`

``uvm_do` makro kao argument prima *sequence_item*. Kada se nasledi i parametrizuje *uvm_sequence* klasa, nasleđuje se i instanca objekta pod nazivom *req* čiji je tip upravo parametar koji smo prosledili. Što znači da u svakoj sekvenci već postoji objekat *req* koji se može koristiti za komunikaciju. Slično postoji i objekat *rsp* koji treba koristiti za odgovor ukoliko je isti potreban. Upotreba ``uvm_do` makroa je:

```
virtual task body();
    `uvm_do(req)
endtask: body
```

Ovim se postiže generisanje i slanje jedne transakcije drajveru, koja će se zatim generisati na interfejsu.

3.2.2 ``uvm_do_with`

Ukoliko je potrebna veća kontrola prilikom slanja transakcije, moguće je koristiti ``uvm_do_with` makro, koji osim *sequence_item* argumenta prima i bilo koje validno inline ograničenje. Npr:

```
virtual task body();
    `uvm_do_with(req, { req.data == 16'h5A5A; } )
endtask: body
```

U ovom primeru se šalje jedna transakcija pri čemu je vrednost *data* polja ograničena na 16'h5A5A.

3.2.3 Kontrola

U prethodnim primerima smo slali samo jednu transakciju. Često je neophodno omogućiti veću kontrolu nad izvršavanjem sekvenci - omogućiti kontrolisanu randomizaciju, slanje željenog broja transakcija itd. Prilikom pisanja sekvenci moguće je koristiti sve mogućnosti koje SystemVerilog pruža. U nastavku je dato par konstrukcija koje se često sreću i koje daju uvid u ove mogućnosti.

Kontrola prilikom randomizacije je često vrši pomoću *rand* polja u sekvenci, kako bi se omogućila laka kontrola direktno iz testa koji startuje sekvencu. Sledeći primer ilustruje slanje više transakcija.

```
class calc_simple_seq extends calc_base_seq;

  `uvm_object_utils (calc_simple_seq)

  rand int unsigned num_of_tr;

  constraint num_of_tr_con { num_of_tr inside { [1:100] }; }

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

  virtual task body();
    repeat(num_of_tr) begin
      `uvm_do(req);
    end
  endtask : body

endclass : calc_simple_seq
```

Takođe je moguće pozivati jednu sekvencu iz druge, pri čemu je moguće koristiti prethodno opisane makroe i nad sekvencama (ili koristiti metodu *start* opisanu u narednom poglavlju).

```
class calc_simple_seq extends calc_base_seq;

  `uvm_object_utils (calc_simple_seq)

  calc_sub_seq sub_seq;

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

  virtual task body();
    `uvm_do(sub_seq);
  endtask : body

endclass : calc_simple_seq
```

3.3 Pokretanje sekvenci

Sekvence se pokreću ili iz testa ili iz drugih sekvenci. Postoje dva glavna načina pokretanja sekvenci - eksplicitno koristeći *start* metodu ili implicitno kao podrazumevanu sekvencu.

Ukoliko je potrebna veća kontrola pokretanja sekvenci (npr. podešavanje parametara, trenutak startovanja, ...) preporučuje se eksplicitno pokretanje sekvence korišćenjem *start* metode, npr:

```
`ifndef TEST_SIMPLE_SV
`define TEST_SIMPLE_SV

class test_simple extends test_base;
```

```

`uvm_component_utils(test_simple)

calc_simple_seq simple_seq;

function new(string name = "test_simple", uvm_component parent = null);
    super.new(name,parent);
endfunction : new

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    simple_seq = calc_simple_seq::type_id::create("simple_seq");
endfunction : build_phase

task main_phase(uvm_phase phase);
    phase.raise_objection(this);
    simple_seq.start(seqr);
    #100ms; // pre nego sto završimo simulaciju dajemo vremena DUT da izvrši proračun
    phase.drop_objection(this);
endtask : main_phase

endclass

`endif

```

Kod 3: *Start*_metoda

Prilikom poziva *start* metode jedini obavezan argument je sekvencer na kome će pokrenuti sekvencu. Što se tiče podrazumevanih sekvenci, one se uglavnom navode u *build* fazi testa, npr:

```

`ifndef TEST_SIMPLE_2_SV
`define TEST_SIMPLE_2_SV

class test_simple_2 extends test_base;

    `uvm_component_utils(test_simple_2)

    function new(string name = "test_simple_2", uvm_component parent = null);
        super.new(name,parent);
    endfunction : new

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

        uvm_config_db#(uvm_object_wrapper)::set(this,
                                                    "seqr.main_phase",
                                                    "default_sequence",
                                                    calc_simple_seq::type_id::get());

    endfunction : build_phase

endclass

`endif

```

Kod 4: Podrazumevane_sekvence

Za navođenje podrazumevane sekvence se koristi UVM *configuration database* koji omogućava postavljanje konfiguracionih parametara u okruženju, u našem slučaju podrazumevane sekvence (*uvm_config_db* će biti detaljnije obrađen u narednim vežbama). U prethodnom primeru je za *main* fazu prethodno instanciranog *seqr* sekvencera podešena *calc_simple_seq* kao podrazumevana sekvencu.

Pokretanje podrazumevanih sekvenci je u UVM-u nasleđeno iz OVM-a i zadržano je radi lakšeg prelaza sa jedne metodologije na drugu. Iako i dalje često korišćeno, preporuka je koristiti *start* metodu radi bolje kontrole.

4 Sekvencer

Sekvencer koristi sekvence kako bi poslao podatke drajveru, ali i prima odgovor od drajvera ukoliko je to potrebno čime se omogućava kreiranje korisnih stimulusa. Sekvencer služi kao arbiter čiji je zadatak kontrola slanja transakcija iz jedne ili više sekvenci.

U praksi najčešće nije potrebno modifikovati sekvencer jer su funkcionalnosti koje se nalaze u *uvm_sequencer* klasi obično dovoljne. Kao i drajver, i sekvencer je moguće parametrizovati sa tipom transakcije, npr:

```
class calc_sequencer extends uvm_sequencer#(calc_seq_item);  
  // ...  
endclass : calc_sequencer
```

Ukoliko, osim parametrizacije, nije potrebno ubacivati dodatni sadržaj u sekvencer, često se samo koristi *typedef* naredba kako bi se izbegao nepotrební kod, npr:

```
typedef uvm_sequencer#(calc_seq_item) calc_sequencer;
```

5 Drajver

Drajver je aktivna komponenta koja emulira signale koji se šalju dizajnu. Na osnovu podataka iz *sequence item*-a i smplovanja signala na interfejsu, postavlja signale na ulaze dizajna koji se verifikuje. Transakcije koje prima su obično na višem nivou apstrakcije, pa se vrši konverzija na pin-nivo apstrakcije kako bi se ispravno generisali signali. Drajver je preko TLM porta povezan na sekvencer kako bi vršio komunikaciju i sadrži virtuelni interfejs kako bi generisao signale. U UVM-u se drajver implementira nasleđujući *uvm_driver* $\#(REQ, RSP)$ klasu, koja je parametrizovana tipom transakcija koju će drajver zahtevati i odgovarati (engl. *request, response*). Ukoliko se navede samo REQ i RSP će biti istog tipa kao i REQ. Npr:

```
class calc_driver extends uvm_driver#(calc_seq_item);
// ...
endclass : calc_driver
```

Napomena: u SystemVerilog-u se parametrizacija vrši koristeći “#” znak. Nalik C++-su ovim mehanizmom se omogućava pisanje generičkih klasi kojima će se stvarna vrednost parametra proslediti tek prilikom korišćenja. Primer deklarisanja parametrizovane klase:

```
class param_example #(type T = int, type G = bit, int A = 10);
```

Kao i sve komponente unutar UVM verifikacionog okruženja, i drajver se implementira tako što se nasledi *uvm_driver* klasa, parametrizovana tipom *sequence item*-a koji će se koristiti za zahtev odnosno odgovor. Pored ovog drajver treba da sadrži kod za *factory* registraciju, konstruktor, kao i virtuelni interfejs preko koga će generisati signale. Kostur većine drajvera je dat ispod.

```
class example_driver extends uvm_driver#(example_seq_item);

  `uvm_component_utils(example_driver)

  virtual interface example_if vif;

  function new(string name = "example_driver", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    if (!uvm_config_db#(virtual example_if)::get(this, "*", "example_if", vif))
      `uvm_fatal("NO_IF", {"virtual interface must be set for:", get_full_name(), ".vif"})
  endfunction : connect_phase

  task main_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      drive_tr();
      seq_item_port.item_done();
    end
  endtask : main_phase

  task drive_tr();
    // do actual driving here
    // ...
  endtask : drive_tr

endclass : example_driver
```

Nasledivanjem *uvm_driver* klase, nasleđuje se i TLM port za komunikaciju sa sekvencerom (*seq_item_port*), ali i *req* i *rsp* objekti koji se mogu koristiti za generisanje stimulusa. Ovi objekti će poprimiti tip kojim je parametrizovan drajver (*example_seq_item* u primeru).

5.1 API

API za komunikaciju sa sekvencerom je:

- *get_next_item* - blokirajuća metoda koja čeka da REQ transakcija postane dostupna i vraća pokazivač na taj objekat; nakon izvršavanja ove metode obavezno je pozvati *item_done* pre sledećeg poziva *get_next_item*
- *try_next_item* - neblokirajuća metoda koja proverava da li postoji transakcija. Vraća *null* ukoliko nema dostupne transakcije u sekvenceru
- *item_done* - neblokirajuća metoda koja finalizira *handshake* sa sekvencerom; treba je pozvati nakon *get_next_item* ili uspešnog *try_next_item*; moguće joj je proslediti RSP transakciju kao argument
- *peek* - blokirajuća metoda koja čeka na dostupnu REQ transakciju od sekvencera i vraća pokazivač na taj objekat; naredni pozivi ka *peek* pre *get* ili *item_done* poziva će vratiti pokazivač na isti REQ objekat
- *get* - blokirajuća metoda koja čeka na dostupnu REQ transakciju od sekvencera; kada primi transakciju kompletira *handshake* i vraća pokazivač na REQ objekat
- *put* - neblokirajuća metoda koja vraća RSP transakciju kao odgovor sekvenceru; može se pozvati u bilo kom trenutku i nije deo *handshake*-a između drajvera i sekvencera

Na osnovu ovih metoda, postoje dva načina da se ostvari komunikacija sa sekvencerom - koristeći *get_next_item* / *item_done* ili *get* / *put* mehanizam.

Ispravno generisanje stimulusa u UVM-u zavisi od dobre veze između sekvenci i drajvera. Sekvence i drajver se moraju usaglasiti kako ne bi došlo do *deadlock* situacija kada se jedna strana zauvek blokira čekajući na odgovor. Kako bi se izbegle ovakve situacije i omogućila laka upotreba, ako i ponovna upotreba drajvera potrebno je dobro izmodelovati i dokumentovati funkcionalnost drajvera. Postoji veliki broj modela za generisanje stimulusa, međutim neki od najčešće korišćenjih modela za drajvere su:

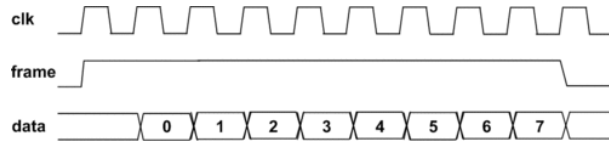
- *Unidirectional Non-Pipelined* - zahtevi se šalju drajveru, ali se ne primaju odgovori od drajvera; drajver može koristiti neki vid rukovanja, ali je proces slanja podataka uvek u jednom smeru; ovaj model se koristi za jednostrane komunikacione kanale nalik PCM interfejsu
- *Bidirectional Non-Pipelined* - podaci se šalju u oba smera, sekvencer šalje transakcije drajveru, dok drajver šalje odgovor; uvek je samo jedna transakcija aktivna, odgovor se uvek šalje pre slanja sledeće transakcije; ovaj model se koristi za jednostavnije protokole npr. APB
- *Pipelined* - podaci se šalju u oba smera, ali se faza slanja zahteva preklapa sa fazom slanja odgovora na prethodni zahtev; protočna obrada omogućava bolje performanse, ali komplikuje sam kod jer se zahtevi i odgovori moraju posebno obrađivati; primer upotrebe bio bi za AHB protokol
- *Out Of Order Pipelined* - u nekim primenama odgovori se ne šalju u redosledu u kom su primeljni zahtevi; ovo rezultuje u dosta komplikovanijem modelu gde se uglavnom koriste redovi za praćenje odgovora; još jedna varijacija bila bi *burst* transferi za više transakcija istovremeno; primer upotebe npr. AXI

U nastavku je su data dva primera modela drajvera. Za kompletan opis svih modela kao i alternativne implementacije datih modela pogledati "UVM Cookbook", poglavlje "*Driver/Use Models*".

5.1.1 Unidirectional Non-Pipelined

Kod ovog modela drajver kontroliše flow koristeći *get_next_item* poziv kako bi primio sledeću transakciju, a poziv *item_done* metode se vrši tek kada je završeno sa procesiranjem transakcije.

U nastavku je dat korišćenja ovog modela i odgovarajuće sekvence. U primeru se šalju ADPCM paketi koristeći PCM protokol.



Slika 2: Protokol

```
class unidir_driver extends uvm_driver #(unidir_seq_item);

  `uvm_component_utils(unidir_driver)

  virtual unidir_if vif;

  function new(string name = "unidir_driver", uvm_component parent = null);
    super.new(name, parent);
  endfunction

  task main_phase(uvm_phase phase);
    int top_idx = 0;
    // pocetno stanje
    vif.frame <= 0;
    vif.data <= 0;

    forever begin
      seq_item_port.get_next_item(req); // zahtev za transakcijom
      repeat (req.delay) begin // kasnjenje izmedju paketa
        @(posedge vif.clk);
      end

      vif.frame <= 1; // pocetak slanja
      for (int i = 0; i < 8; i++) begin // slanje podataka
        @(posedge vif.clk);
        vif.data <= req.data[3:0];
        req.data = req.data >> 4;
      end

      vif.frame <= 0; // kraj slanja
      seq_item_port.item_done(); // transakcija je završena
    end
  endtask; run
endclass: unidir_driver

// generisanje transakcija u petlji
class unidir_tx_seq extends uvm_sequence #(unidir_seq_item);

  `uvm_object_utils(unidir_tx_seq)

  // unidir sequence_item
  unidir_seq_item req;

  // kontrola za broj transakcija
  rand int no_reqs = 10;
```

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

task body();
    req = unidir_seq_item::type_id::create("req");
    repeat(no_reqs) begin
        start_item(req);
        assert(req.randomize());
        finish_item(req);
    end
endtask: body

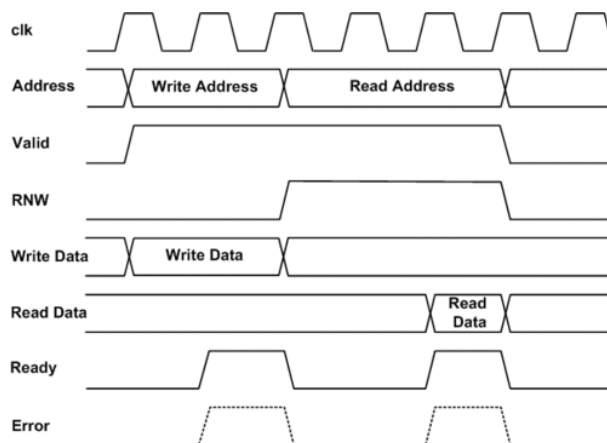
endclass: unidir_tx_seq
```

Kod 5: Primer *use* modela

5.1.2 Bidirectional Non-Pipelined

Najčešće korišćen model je upravo bidirekcionni. Sekvencer šalje zahteve drajveru koji ih izvršava, a zatim šalje odgovor nazad. Nova transakcija ne može biti započeta do god se ne primi odgovor i ne završi prethodna. Ovi modeli se koriste za jednostavnije protokole kao što je npr. AMPA APB.

Drajver prima transakciju, po protokolu koji se implementira pošalje podatke uz eventualno čekanje i zatim postavi polja u transakciji koja služe za odgovor i završi rukovanje sa sekvencerom. Pošto se i u drajveru i u sekvenci koristi pokazivač na isti objekat, nakon što se sekvenca odblokira (posle *finish_item*) može koristiti objekat sa novim vrednostima i vršiti dalju analizu. Primer je dat ispod:



Slika 3: Protokol

```
class bidir_driver extends uvm_driver #(bidir_seq_item);

    `uvm_component_utils(bidir_driver)

    virtual bidir_if vif;

    function new(string name = "bidir_driver", uvm_component parent = null);
        super.new(name, parent);
    endfunction

    task main_phase(uvm_phase phase);
        // pocetno stanje
        vif.valid <= 0;
    endtask
endclass
```

```

    vif.rnw <= 1;
    // cekanje da se reset završi
    @(posedge vif.resetn);

    forever begin
        seq_item_port.get_next_item(req); // zahtev za transakcijom

        repeat(req.delay) begin
            @(posedge vif.clk);
        end
        vif.valid <= 1;
        vif.addr <= req.addr;
        vif.rnw <= req.read_not_write;
        if (req.read_not_write == 0) begin
            vif.write_data <= req.write_data;
        end
        while(vif.ready != 1) begin
            @(posedge vif.clk);
        end

        // na kraju transakcije podesiti polja za odgovor
        if (req.read_not_write == 1) begin
            req.read_data = vif.read_data; // vratiti procitani podatak, ukoliko je u pitanju citanje
        end
        req.error = vif.error; // podesiti status
        vif.valid <= 0; // kraj slanja

        seq_item_port.item_done(); // transakcija je završena
    end
endtask

endclass

class bidir_seq extends uvm_sequence #(bidir_seq_item);

    `uvm_object_utils(bidir_seq)

    bidir_seq_item req;

    rand int limit = 40; // broj iteracija

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

    task body();
        req = bidir_seq_item::type_id::create("req");

        repeat(limit) begin
            start_item(req);
            // adresa je ogranicena na validan opseg
            assert (req.randomize() with {addr inside {[16'h0100:16'h1000]}});
            finish_item(req);
            // req pokazuje na objekat sa novim vrednostima podesenim u drajveru
            uvm_report_info("seq_body", req.convert2string());
        end
    endtask

endclass

```

Kod 6: Primer *use* modela

6 Komunikacija

UVM komponente komuniciraju preko standardnih TLM (engl. *transaction level modeling*) interfejsa koji olakšava povezivanje i ponovno korišćenje. Komponenta može komunicirati preko svog interfejsa sa bilo kojom drugom komponentom koja implementira taj interfejs. U praksi je ovaj način komunikacije veoma čest za drajver - sekvencer interakciju i monitor - *scoreboard* komunikaciju. U ovoj i narednoj vežbi je fokus na komunikaciji između drajvera i sekvencera, dok je detaljan opis rada TLM konekcija ostavljen za vežbu o monitoru.

U *uvm_driver* i *uvm_sequencer* klasama postoji deklaracija odgovarajućih portova i ugrađena logika oko komunikacije, tako da je upotreba od strane korisnika veoma jednostavna. Potrebno je samo povezati drajver i sekvencer, koristeći metodu *connect*. Povezivanje se tipično vrši u *connect* fazi, gde je, na primeru ispod, *driver* instanca prethodno kreiranog drajvera, a *sequencer* instanca prethodno kreiranog sekvencera. *seq_item_port* i *seq_item_export* su imena nasleđenih TLM portova.

```
function void connect_phase(uvm_phase phase);
    driver.seq_item_port.connect(sequencer.seq_item_export);
endfunction : connect_phase
```

Sam UVM ne pruža samo mogućnost korišćenja biblioteke, već i definiše način upotrebe kako bi se kreirale univerzalne verifikacione komponente koje će pratiti datu metodologiju. S tim u vezi, postoji jasno definisana komunikacija između drajvera i sekvencera, koristeći određene UVM funkcije, čiji je opis dat u nastavku.

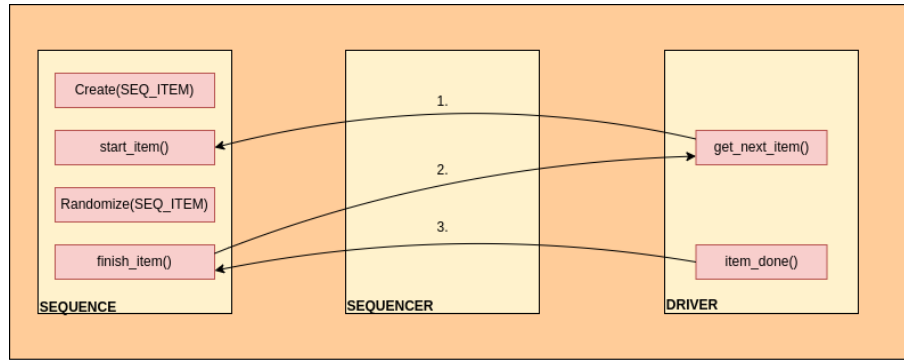
UVM drajver sadrži mnoge metode za komunikaciju sa sekvencerom. Najčešće korišćene su:

- *get_next_item()* - blokirajuća metoda koja traži novi objekat; vraća pokazivač na novi REQ objekat
- *try_next_item()* - neblokirajuća verzija koja vraća *null* pokazivač ukoliko nema dostupnog objekta
- *item_done()* - neblokirajuća metoda koja završava proces rukovanja između drajvera i sekvencera; poziva se nakon *get_next_item()* ili uspešnog *try_next_item()*

Sam sekvencer služi za arbitraciju, dok se *sequence item* objekti kreiraju i kontrolišu iz sekvenci. Da bi se paket poslao drajveru, on mora proći kroz nekoliko koraka:

1. Kreiranje - veoma je bitno kreirati paket koristeći *factory* kako bi se olakšala ponovna upotreba odnosno kako bi se dozvolio *override* metod ukoliko je to potrebno (objašnjeno u narednim vežbama)
2. *Start_item()* - blokirajući poziv koji čeka da se uspostavi veza sa drajverom. Sekvenceru šalje pokazivač na sledeći objekat
3. Randomizacija paketa ukoliko je to potrebno
4. *Finish_item()* - blokirajući poziv koji čeka da drajver završi sa paketom (napomena: *start_item()* i *finish_item()* čekaju na drajver kako bi se osigurao korektan transfer paketa, ali oni ne troše simulaciono vreme)
5. *Get_response()* - opcioni poziv ako je potrebno da drajver pošalje odgovor sekvenceru

Preporučeni *flow* je prikazan na slici 4. Drajver traži transakciju od sekvencera koristeći *get_next_item()*, obrađuje istu i zatim kompletira *handshake* koristeći *item_done()* čime signalizira sekvenceru da je objekat obrađen. Sa druge strane, u sekvenci se kreiraju transakcije prateći prethodne korake. Kada drajver pozove *item_done()*, odblokiraće se *finish_item()* metoda u sekvenci i može se nastaviti sa



Slika 4: Drajver-sekvencer *flow*

izvršavanjem.

Pored ovog načina, moguće je ostvariti komunikaciju koristeći *peek()*, *put()* i *get()* metode u drajveru koje su opisane u narednoj vežbi.

7 Mehanizam završetka testa

Svaki testbenč koji koristi standardne UVM faze se sastoji od nekoliko faza za kreiranje i povezivanje koje se izvršavaju u nultom vremenu, zatim određen broj faza koje troše vreme, kao i nekoliko faza za završetak testa koje se izvršavaju u nultom vremenu. Do kraja testa će doći kada se završe sve faze koje troše vreme, a svaka faza se završava kada nema prigovora za tu fazu što znači da se kraj testa kontroliše podizanjem i spuštanjem prigovora. Komponenta ili sekvenca će podići prigovor na početku aktivnosti koja se mora završiti u datoj fazi, a spustiti prigovor po završetku te aktivnosti. Tek kada se svi podignuti prigovori spuste, moguće je završiti datu fazu. U praksi se često dešava da je faza samo trenutno bez prigovora tj. da će se uskoro podići još neki prigovor i da ne treba odmah završiti sa trenutnom fazom. U tom slučaju se može postaviti vreme čekanja (engl. *drain time*) koje unosi kašnjenje za završetak faze. Ako u tom vremenskom intervalu podigne prigovor, faza se neće završiti i opet će se čekati na spuštanje svih prigovora.

Kontrola prigovora se uglavnom vrši ili iz samog testa ili iz sekvenci. Kontrola iz testa se često sreće kada se sekvence eksplicitno pokreću. Jedan primer je prikazan ispod.

```
task main_phase(uvm_phase phase);
    phase.raise_objection(); //raising objection
    example_seq.start(example_agent.sequencer);
    phase.drop_objection(); //dropping objection
endtask
```

Što se tiče kontrole iz sekvenci, pošto se čitava logika generisanja stimulusa nalazi u *body()* tasku, prigovor je potrebno podići pre odnosno posle izvršavanja *body()* taska i ovo se najčešće radi u *pre_body()* odnosno *post_body()* taskovima. Ovaj kod se najčešće nalazi u baznoj sekvenci kako bi se prigovori generisali prilikom pokretanja svake sekvence. Test se tada neće završiti do god postoje sekvence u okruženju koje nisu završile sa generisanjem stimulusa.

```
// objections are raised in pre_body
virtual task pre_body();
    uvm_phase phase = get_starting_phase();
    if (phase != null)
        phase.raise_objection(this, {"Running sequence '", get_full_name(), "'});
    uvm_test_done.set_drain_time(this, 200ns)
endtask : pre_body

// objections are dropped in post_body
virtual task post_body();
    uvm_phase phase = get_starting_phase();
    if (phase != null)
        phase.drop_objection(this, {"Completed sequence '", get_full_name(), "'});
endtask : post_body
```

Jedan način podizanja / spuštanja prigovora je prikazan kodom iznad. Koriste se *raise_objection/drop_objection* metode. *starting_phase* je član bazne klase koji će biti ispravno postavljen jedino ukoliko je sekvenca pokrenuta kao podrazumevana (*default*) za neku fazu.

Pomoću metode *set_drain_time* se postavlja takozvano *drain* vreme koje u datom primeru unosi toleranciju od 200 ns, odnosno nakon spuštanja poslednjeg prigovora test se neće odmah završiti nego će se sačekati još 200 ns zbog mogućnosti da se u tom periodu ponovo podigne neki prigovor.

Napomena: pošto je sekvenca ručno pokrenuta, *starting_phase* polje iz prethodnog kodnog fragmenta bi imalo vrednost *null* tako da se korišćenjem oba kodna fragmenta istovremeno ne bi podigla dva prigovora za istu sekvencu.

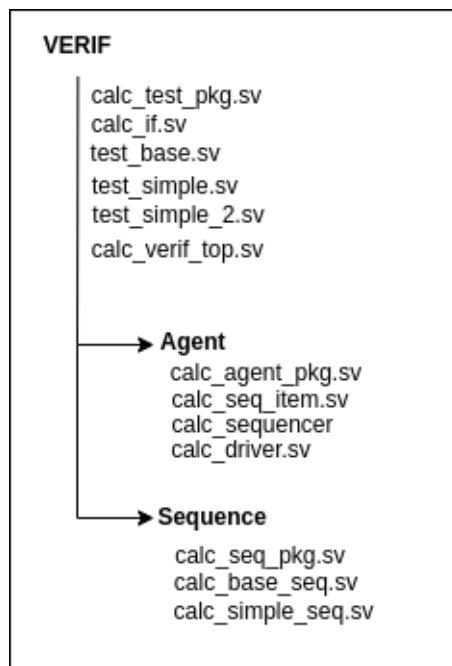
Napomena: podizanje / spuštanje prigovora u sekvenci se često koristi uz UVM 1.1 biblioteku. Za UVM 1.2 preporučen način je podizanje / spuštanje prigovora iz testa.

8 Organizacija fajlova koji čine verifikaciono okruženje

Broj fajlova koji čine verifikaciono okruženje može da postane dosta veliki i stoga određeni alati(kao Vivado) mogu da imaju problem da shvate kojim redosledom treba da kompajliraju fajlove. Iz tog razloga alatu treba malo pomoći.

U svakom direktorijumu koji čini verifikaciono okruženje preporuka je da postoji Systemverilog paket koji uključuje(*eng. include*) sve fajlove iz tog direktorijuma (sem *top* fajla), i koji takođe uključuje(*eng. import*) fajlove iz drugih direktorijuma ukoliko ih koriste fajlovi iz trenutnog direktorijuma. Fajlove treba uvlačiti poštujući hijerarhiju verifikacionog okruženja, fajlovi najniži u hijerarhiji se navode prvi(ukoliko se uvuče test pre sekvence, kompajliranje će pući jer test instancira sekvencu koja nije kompajlirana).

U priloženom materijalu za ove vežbe nalazi se verif direktorijum sa svim fajlovima koji čine skelet verifikacionog okruženja za verifikaciju calc dizajna:



Slika 5: Hijerarhija direktorijuma, vežba 6

Taj direktorijum unutar sebe poseduje *calc_test_pkg.sv* fajl unutar koga su uključeni svi fajlovi iz verif direktorijuma koji čine verifikaciono okruženje, što ilustruje sledeći kodni listing:

```

`ifndef CALC_TEST_PKG_SV
`define CALC_TEST_PKG_SV

package calc_test_pkg;

    import uvm_pkg::*;    // import the UVM library
    `include "uvm_macros.svh" // Include the UVM macros

    import calc_agent_pkg::*;
    import calc_seq_pkg::*;

    `include "v6_test_base.sv"
    `include "v6_test_simple.sv"
    `include "v6_test_simple_2.sv"

endpackage : calc_test_pkg
  
```

```
`include "calc_if.sv"

`endif
```

Kod 7: v6_calc_test_pkg

Svaki pod-direktorijum unutar verif direktorijuma takođe poseduje Systemverilog paket koji unutar sebe uključuje sve fajlove iz tog direktorijuma. Iz prethodnog kodnog listinga se može videti da calc_test_pkg pored fajlova iz datog direktorijuma uvlači fajlove iz drugih direktorijuma:

```
import calc_agent_pkg::*;
import calc_seq_pkg::*;
```

Razlog za ovo je taj što unutar klase *test* instanciramo objekte klase *driver*, *sequence* i *sequencer* koji su definisani u drugim direktorijumima. Naredba `import calc_agent_pkg::*` importuje sve fajlove koji su uvučeni unutar calc_agent_pkg.sv fajla. Naravno, mogli smo da uvučemo i samo određenu klasu iz pomenutog pod-direktorijuma na sledeći način:

```
import calc_agent_pkg::calc_driver;
```

Obratite pažnju na redosled uvlačenja fajlova u prethodnom kodnom listingu. Pošto klasa test_base unutar sebe instancira drajver i sekvencer, pre nego što se uvuče fajl u kome je definisana test_base klasa moraju se importovati fajlovi u kojima su definisani drajver i sekvencer. Ukoliko bi se uradilo sledeće, kompajliranje ne bi prošlo:

```
`ifndef CALC_TEST_PKG_SV
`define CALC_TEST_PKG_SV

package calc_test_pkg;

    import uvm_pkg::*; // import the UVM library
    `include "uvm_macros.svh" // Include the UVM macros

    `include "test_base.sv" // greska, moraju prvo da se importuju fajlovi iz agent paketa i sequence paketa !!!!
    import calc_agent_pkg::*;
    import calc_seq_pkg::*;

    `include "test_simple.sv"
    `include "test_simple_2.sv"

endpackage : calc_test_pkg

`include "calc_if.sv"

`endif
```

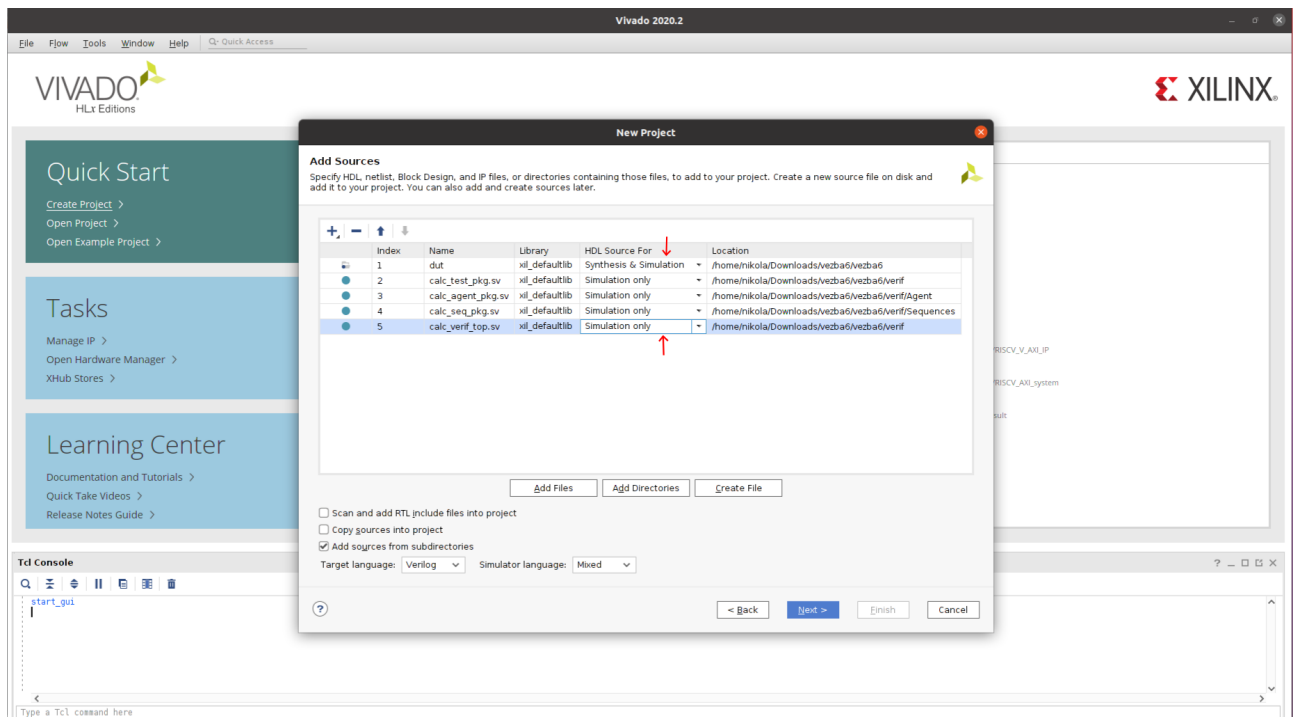
Kod 8: v6_calc_test_pkg sa greškom

Napomena: U test_pkg.sv fajlu može se primetiti da je calc_if.sv fajl uvučen nakon definicije paketa. To je urađeno jer Systemverilog ima ograničenje po kome *interface* ne sme biti definisan unutar paketa.

8.1 Kompajliranje u Vivadu

Ukoliko se koristi Vivado za kompajliranje fajlova, prilikom kreiranja projekta potrebno je da se ubace dizajn fajlovi, paketi u kojima su uključeni svi fajlovi koji čine verifikacioni okruženje i verifikacioni top fajl. Na sledećoj slici prikazan je primer dodavanja dizajn fajlova koji čine calc digitalni sistem kao i verifikacionih fajlova koji se koriste za verifikaciju:

Crvenom strelicom je naznačeno da svi dizajn fajlovi treba da se koriste za sintezu i simulaciju, dok se fajlovi koji čine verifikaciono okruženje koriste samo pri simulaciji.



Slika 6: Vivado, dodavanje fajlova pri kreiranju projekta

8.2 Kompajliranje u Cadence Xcelium simulatoru

Ukoliko se za simulaciju koristi Xcelium simulator, onda je prilikom pisanja .f fajla neophodno uključiti samo dizajn fajlove, pakete i top fajl. Sledeći primer ilustruje .f fajl pomoću koga se kompajliraju fajlovi iz priloženog materijala za vežbu 6:

```

--uvmhome "/eda/cadence/2019-20/RHELx86/XCELIUM_19.03.013/tools/methodology/UVM/CDNS-1.2/"
--uvm +UVM_TESTNAME=test_simple
--sv +incdir+./verif
--sv +incdir+./verif/Agent
--sv +incdir+./verif/Sequences

./dut/alu_input_stage.v
./dut/alu_output_stage.v
./dut/exdbin_mac.v
./dut/holdreg.v
./dut/mux_out.v
./dut/shifter.v
./dut/priority.v
./dut/calc_top.v

--sv ./verif/Agent/calc_agent_pkg.sv
--sv ./verif/Sequences/calc_seq_pkg.sv
--sv ./verif/calc_test_pkg.sv

--sv ./verif/calc_verif_top.sv

##--LINEDEBUG
--access +rwc
    
```

```
—disable _sem2009  
—nowarn "MEMODR"  
—timescale 1ns/10ps
```

Kod 9: v6_run.f

9 Zadaci

Zadatak Napisati sekvence koje će se koristiti za verifikaciju “Calc1” DUT-a, uključujući:

- Sekvenca za generisanje jedne transakcije na nasumično izabranom portu, sa nasumičnim podacima i komandom
- Sekvenca za generisanje više nasumičnih transakcija na istom portu
- Sekvenca za generisanje više nasumičnih transakcija, pri čemu se broj transakcija nalazi u opsegu (1, 10), a svaka naredna transakcija se nalazi na portu različitom od prethodnog
- Sekvenca za generisanje više nasumičnih transakcija pri čemu su komande uvek za jednu ALU, bez kašnjenja između generisanja transakcija
- smisliti još nekoliko primera sekvenci koji bi mogle biti korisne za verifikaciju

Zadatak Napisati test (uz eventualne pomoćne testove ili sekvence) u kome će se poslati tačno 8 transakcija, koristeći neku od prethodno razvijenih sekvenci. Na koje je načine ovo moguće odraditi?

U sekciji *Appendix* i u priloženom materijalu nalazi se skelet verifikacionog okruženja za proveru ispravnosti rada calc dizajna. Proširiti verifikaciono okruženje tako da se provere prethodno navedeni scenariji.

10 Appendix

```

`ifndef CALC_IF_SV
`define CALC_IF_SV

interface calc_if (input clk, logic [6 : 0] rst);

    parameter DATA_WIDTH = 32;
    parameter RESP_WIDTH = 2;
    parameter CMD_WIDTH = 4;

    logic [DATA_WIDTH - 1 : 0] out_data1;
    logic [DATA_WIDTH - 1 : 0] out_data2;
    logic [DATA_WIDTH - 1 : 0] out_data3;
    logic [DATA_WIDTH - 1 : 0] out_data4;
    logic [RESP_WIDTH - 1 : 0] out_resp1;
    logic [RESP_WIDTH - 1 : 0] out_resp2;
    logic [RESP_WIDTH - 1 : 0] out_resp3;
    logic [RESP_WIDTH - 1 : 0] out_resp4;
    logic [CMD_WIDTH - 1 : 0] req1_cmd_in;
    logic [DATA_WIDTH - 1 : 0] req1_data_in;
    logic [CMD_WIDTH - 1 : 0] req2_cmd_in;
    logic [DATA_WIDTH - 1 : 0] req2_data_in;
    logic [CMD_WIDTH - 1 : 0] req3_cmd_in;
    logic [DATA_WIDTH - 1 : 0] req3_data_in;
    logic [CMD_WIDTH - 1 : 0] req4_cmd_in;
    logic [DATA_WIDTH - 1 : 0] req4_data_in;

endinterface : calc_if

`endif

```

Kod 10: calc_if

```

`ifndef CALC_SEQUENCER_SV
`define CALC_SEQUENCER_SV

class calc_sequencer extends uvm_sequencer#(calc_seq_item);

    `uvm_component_utils(calc_sequencer)

    function new(string name = "calc_sequencer", uvm_component parent = null);
        super.new(name, parent);
    endfunction

endclass : calc_sequencer

`endif

```

Kod 11: v6_calc_sequencer

```

`ifndef CALC_DRIVER_SV
`define CALC_DRIVER_SV
class calc_driver extends uvm_driver#(calc_seq_item);

    `uvm_component_utils(calc_driver)
    virtual interface calc_if vif;
    function new(string name = "calc_driver", uvm_component parent = null);
        super.new(name, parent);
    endfunction // new

    function void build_phase(uvm_phase phase);
        if (!uvm_config_db#(virtual calc_if)::get(null, "*", "calc_if", vif))
            `uvm_fatal("NOVIF", {"virtual interface must be set:", get_full_name(), ".vif"})
        endfunction // build_phase

```

```

task main_phase(uvm_phase phase);
    forever begin
        @(posedge vif.clk);
        if (!vif.rst)
            begin
                seq_item_port.get_next_item(req);

                `uvm_info(get_type_name(),
                $sformatf("Driver sending ...\n%s", req.sprint()),
                UVM_HIGH)
                vif.req1_data_in = req.operand1;
                vif.req1_cmd_in = req.cmd;
                @(posedge vif.clk);
                vif.req1_data_in = req.operand2;
                vif.req1_cmd_in = 0;

                seq_item_port.item_done();
            end
        end
    endtask : main_phase
endclass : calc_driver
`endif

```

Kod 12: v6_calc_driver

```

`ifndef CALC_SEQ_ITEM_SV
`define CALC_SEQ_ITEM_SV

class calc_seq_item extends uvm_sequence_item;

    /* TODO add fields and methods here */

    `uvm_object_utils_begin(calc_seq_item)
    `uvm_object_utils_end

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

endclass : calc_seq_item
`endif

```

Kod 13: v6_calc_seq_item

```

`ifndef CALC_AGENT_PKG
`define CALC_AGENT_PKG

package calc_agent_pkg;

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

    //////////////////////////////////////////
    // include Agent components : driver,monitor,sequencer
    //////////////////////////////////////////
    `include "v6_calc_seq_item.sv"
    `include "v6_calc_sequencer.sv"
    `include "v6_calc_driver.sv"

endpackage

```

```
`endif
```

Kod 14: v6_calc_agent_pkg

```
`ifndef TEST_BASE_SV
`define TEST_BASE_SV

class test_base extends uvm_test;

    `uvm_component_utils(test_base)

    calc_driver drv;
    calc_sequencer seqr;

    function new(string name = "test_base", uvm_component parent = null);
        super.new(name,parent);
    endfunction : new

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        drv = calc_driver::type_id::create("drv", this);
        seqr = calc_sequencer::type_id::create("seqr", this);
    endfunction : build_phase

    function void connect_phase(uvm_phase phase);
        drv.seq_item_port.connect(seqr.seq_item_export);
    endfunction : connect_phase

endclass : test_base

`endif
```

Kod 15: v6_test_base

```
`ifndef TEST_SIMPLE_SV
`define TEST_SIMPLE_SV

class test_simple extends test_base;

    `uvm_component_utils(test_simple)

    calc_simple_seq simple_seq;

    function new(string name = "test_simple", uvm_component parent = null);
        super.new(name,parent);
    endfunction : new

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        simple_seq = calc_simple_seq::type_id::create("simple_seq");
    endfunction : build_phase

    task main_phase(uvm_phase phase);
        phase.raise_objection(this);
        simple_seq.start(seqr);
        #100ms; // pre nego sto završimo simulaciju dajemo vremena DUT da izvrši proračun
        phase.drop_objection(this);
    endtask : main_phase

endclass

`endif
```

Kod 16: v6_test_simple

```
`ifndef TEST_SIMPLE_2_SV
```

```

`define TEST_SIMPLE_2_SV

class test_simple_2 extends test_base;

    `uvm_component_utils(test_simple_2)

    function new(string name = "test_simple_2", uvm_component parent = null);
        super.new(name,parent);
    endfunction : new

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

        uvm_config_db#(uvm_object_wrapper)::set(this,
                                                    "seqr.main_phase",
                                                    "default_sequence",
                                                    calc_simple_seq::type_id::get());

    endfunction : build_phase
endclass

`endif

```

Kod 17: v6_test_simple_2

```

`ifndef CALC_TEST_PKG_SV
`define CALC_TEST_PKG_SV

package calc_test_pkg;

    import uvm_pkg::*;    // import the UVM library
    `include "uvm_macros.svh" // Include the UVM macros

    import calc_agent_pkg::*;
    import calc_seq_pkg::*;

    `include "v6_test_base.sv"
    `include "v6_test_simple.sv"
    `include "v6_test_simple_2.sv"

endpackage : calc_test_pkg

    `include "calc_if.sv"

`endif

```

Kod 18: v6_calc_test_pkg

```

`ifndef CALC_BASE_SEQ_SV
`define CALC_BASE_SEQ_SV

class calc_base_seq extends uvm_sequence#(calc_seq_item);

    `uvm_object_utils(calc_base_seq)
    `uvm_declare_p_sequencer(calc_sequencer)

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

    // objections are raised in pre_body
    virtual task pre_body();
        uvm_phase phase = get_starting_phase();
        if (phase != null)
            phase.raise_objection(this, {"Running sequence '", get_full_name(), "'"});
    endtask : pre_body

```

```

    // objections are dropped in post_body
    virtual task post_body();
        uvm_phase phase = get_starting_phase();
        if (phase != null)
            phase.drop_objection(this, {"Completed sequence '", get_full_name(), "'"});
    endtask : post_body
endclass : calc_base_seq
`endif

```

Kod 19: v6_calc_base_seq

```

`ifndef CALC_SIMPLE_SEQ_SV
`define CALC_SIMPLE_SEQ_SV

class calc_simple_seq extends calc_base_seq;

    `uvm_object_utils (calc_simple_seq)

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

    virtual task body();
        // simple example — just send one item
        `uvm_do(req);
    endtask : body

endclass : calc_simple_seq
`endif

```

Kod 20: v6_calc_simple_seq

```

`ifndef CALC_SEQ_PKG_SV
`define CALC_SEQ_PKG_SV
package calc_seq_pkg;
    import uvm_pkg::*; // import the UVM library
    `include "uvm_macros.svh" // Include the UVM macros
    import calc_agent_pkg::calc_seq_item;
    import calc_agent_pkg::calc_sequencer;
    `include "v6_calc_base_seq.sv"
    `include "v6_calc_simple_seq.sv"
endpackage
`endif

```

Kod 21: v6_calc_seq_pkg

```

`ifndef CALC_TEST_PKG_SV
`define CALC_TEST_PKG_SV
package calc_test_pkg;

    import uvm_pkg::*; // import the UVM library
    `include "uvm_macros.svh" // Include the UVM macros

    import calc_agent_pkg::*;
    import calc_seq_pkg::*;

    `include "test_base.sv"
    `include "test_simple.sv"
    `include "test_simple_2.sv"

```

```
endpackage : calc_test_pkg

`include "calc_if.sv"

`endif
```

Kod 22: v6_calc_test_pkg

```
module calc_verif_top;

    import uvm_pkg::*; // import the UVM library
    `include "uvm_macros.svh" // Include the UVM macros

    import calc_test_pkg::*;

    logic clk;
    logic [6 : 0] rst;

    // interface
    calc_if calc_vif(clk, rst);

    // DUT
    calc_top DUT(
        .c_clk      ( clk ),
        .reset      ( rst ),
        .out_data1   ( calc_vif.out_data1 ),
        .out_data2   ( calc_vif.out_data2 ),
        .out_data3   ( calc_vif.out_data3 ),
        .out_data4   ( calc_vif.out_data4 ),
        .out_resp1   ( calc_vif.out_resp1 ),
        .out_resp2   ( calc_vif.out_resp2 ),
        .out_resp3   ( calc_vif.out_resp3 ),
        .out_resp4   ( calc_vif.out_resp4 ),
        .req1_cmd_in ( calc_vif.req1_cmd_in ),
        .req1_data_in ( calc_vif.req1_data_in ),
        .req2_cmd_in ( calc_vif.req2_cmd_in ),
        .req2_data_in ( calc_vif.req2_data_in ),
        .req3_cmd_in ( calc_vif.req3_cmd_in ),
        .req3_data_in ( calc_vif.req3_data_in ),
        .req4_cmd_in ( calc_vif.req4_cmd_in ),
        .req4_data_in ( calc_vif.req4_data_in )
    );

    // run test
    initial begin
        uvm_config_db#(virtual calc_if)::set(null, "*", "calc_if", calc_vif);
        run_test();
    end

    // clock and reset init .
    initial begin
        clk <= 0;
        rst <= 1;
        #50 rst <= 0;
    end

    // clock generation
    always #50 clk = ~clk;

endmodule : calc_verif_top
```

Kod 23: v6_calc_verif_top