Funkcionalna verifikacija hardvera

Vežba 12
Regresija i proces debagovanja

Sadržaj

Vežba 12 je posvećena regresiji i *debug*-ovanju kako verifikacionog okruženja, tako i samog DUT-a. Opisane su funkcionalnosti UVM-a koje mogu olakšati debug i ubrzati proces verifikacije, dat je opis naprednih mogućnosti alata u pogledu sakupljanja pokrivenosti i spajanja tih podataka. Drugi deo vežbe je posvećen greškama prilikom razvoja okruženja koje se često prave (*gotchas*), bilo zbog specifičnosti jezika ili zbog nerazumevanja implementacije.

1 *Debug* verifikacionog okruženja

Iako simulatori pružaju dosta koristih funkcionalnosti u cilju *debug*-a, postoji i dosta osobina u UVM bibilioteci koje olakšavaju pronalaženje i ispravku problema u verifikacionom okruženju. U ovom poglavlju su opisane te osobine.

1.1 UVM *Command Line Processor*

Sa UVM procesorom komandne linije smo se već sreli na prethodnim vežbama. Ispod su navedene korisne opcije koje mogu olakšati i skratiti vreme pronalaženja problema.

- +UVM_CONFIG_DB_TRACE - omogućava praćenje rada sa *uvm_config_db* klasom. Kada je uključen, na izlazu će se ispisivati UVM_INFO poruke koje daju informacije o tome kada su postavljeni ili preuzeti podaci. Npr:

```
UVM_INFO ../uvm-1.1a/src/base/uvm_resource_db.svh(129) @ 0 ns: reporter [CFGDB/SET]
    Configuration 'uvm_test_top.mem_interface' (type virtual mem_interface) set by  = (virtual
    mem_interface) ?
UVM_INFO ../uvm-1.1a/src/base/uvm_resource_db.svh(129) @ 0 ns: reporter [CFGDB/GET]
    Configuration 'uvm_test_top.mem_interface' (type virtual mem_interface) read by uvm_test_top =
    (virtual mem_interface) ?
```

- +UVM_PHASE_TRACE - olakšava praćenje UVM faza. Kada je uključena ispisuju se informacije o početku i završetku UVM faza. Npr:

```
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1114) @ 0 ns: reporter [PH/TRC/STRT] Phase 'uvm
    .uvm_sched.reset' (id=184) Starting phase
# UVM_INFO ../uvm-1.1a/src/base/uvm_phase.svh(1342) @ 80 ns: reporter [PH/TRC/DONE] Phase '
    uvm.uvm_sched.reset' (id=184) Completed phase
```

- +UVM_OBJECTION_TRACE - na prethodnim vežbama je opisan mehanizam završetka testa uz podizanje/spuštanje prigovora. Ova opcija omogućava praćenje prigovora i ispisuje kada je neki prigovor podignut/spušten, kao i koliko ima aktivnih prigovora. Npr:

```
# UVM_INFO @ 0 ns: reset_objection [OBJTN_TRC] Object uvm_test_top raised 1 objection(s) (
    Raising Reset Objection): count=1  total=1
# UVM_INFO @ 0 ns: reset_objection [OBJTN_TRC] Object uvm_top added 1 objection(s) to its total (
    raised from source object uvm_test_top, Raising Reset Objection): count=0  total=1
# UVM_INFO @ 80 ns: reset_objection [OBJTN_TRC] Object uvm_test_top dropped 1 objection(s) (
    Dropping Reset Objection): count=0  total=0
```

- +UVM_VERBOSITY - pruža mogućnost promene *verbosity*-a u čitavom okruženju odnosno kontrolu ispisa poruka
- +UVM_MAX_QUIT_COUNT - pruža mogućnost završetka simulacije kada se dostigne dati broj grešaka u dizajnu. Podrazumevana vrednost je -1 odnosno simulacija se nikad ne završava na osnovu broja grešaka. Pod brojem grešaka se podrazumeva broj prijavljenih *uvm_error*-a.

1.2 Funkcije

Pored zadavanja opcija preko komandne linije, u UVM-u postoji i veliki broj funkcija koje se mogu koristiti za *debug*. Neke od njih su:

- *print_topology* - funkcija za ispis topologije komponenti u verifikacionom okruženju. Prototip funkcije je: *function void print_topology (uvm_printer printer = null)*, a uglavnom se poziva iz *end_of_elaboration* faze.

- *debug_ connected_ to* - funkcija za debug TLM portova. Ispisuje mapu svih komponenti koje su povezane na dati port. Prototip funkcije je: *function void debug_ connected_ to (int level = 0, int max_ level = -1)*, a uglavnom se poziva iz *end_ of_ elaboration* faze.
- *debug_ provided_ to* - funkcija za debug TLM *imp*-a. Ispisuje mapu svih komponenti koje su povezane na port za koji je povezan dati imp. Prototip funkcije je: *function void debug_ provided_ to (int level = 0, int max_ level = -1)*, a uglavnom se poziva iz *end_ of_ elaboration* faze.

2 Regresija

Nakon verifikovanja osnovnih funkcionalnosti tj. nakon što osnovni testovi prolaze bez grešaka, može se preći na sledeći korak u procesu verifikacije, a to je upravo regresija. Regresija se vrši na svakom nivou hijerarhije kako bi se uverili da nove osobine, ali i prethodno verifikovane osobine i dalje ispravno funkcionišu. U praksi regresija znači puštanje svih (ili ispravno odabranih) postojećih testova, sa nasumičnim *seed*-ovima random generatora. Puštanje testova sa nasumičnim seed-om ima za zadatak da postigne željenu funkcionalnu i strukturnu pokrivenost i otkrije greške u dizajnu, ukoliko postoje. Zbog ovoga je veoma bitno imati dobar model pokrivenosti koji omogućava praćenje stanja verifikacije i eventualne modifikacije prilikom regresije. Nakon svake pronađene greške i bilo kakve promene bilo dizajna bilo verifikacionog okruženja, potrebno je ponovo pustiti regresiju kako bi se uverili da te promene nisu izazvale nove greške.

Međutim, zbog kompleksnosti dizajna i okruženja regresija može biti veoma zahtevna po pitanju vremena i resursa. Efikasnost regresije se može povećati na dva načina - izborom optimalnih testova i puštanjem više testova u paraleli. Prilikom izbora testova treba voditi računa o vremenu trajanja svakog testa, ali i da li je moguće postići željenu pokrivenost sa manjim brojem testova. Ukoliko je moguće, treba odabrati najmanji broj testova koji i dalje zadovoljava pokrivenost.

Analizom izveštaja o pokrivenosti može se zaključiti da li je potrebno modifikovati testove - uvoditi dodatna ograničenja randomizacije ili pisati direktne testove. Cilj ovih modifikacija je pokriti "rupe" i dostići željenu pokrivenost.

2.1 Puštanje testova i spajanje podataka o pokrivenosti

Glavni princip regresije je puštanje testova sa nasumično izabranim vrednostima *seed*-a. Zbog broja testova i velike količine podataka koje treba proanalizirati, važno je dobro organizovati proces regresije. Zbog velikog broja često nije potrebno imati detaljan log fajl za svaki test. Bitne informacije su koja je bila vrednost *seed*-a random generatora i da li je test prijavio greške. Stim u vezi, postoji par podešavanja koja se često koriste prilikom regresije:

- UVM_VERBOSITY na nisku vrednost - fajlovi samo analiziraju u potrazi za greškama. Ukoliko je test javio grešku, pustiće se ponovo sa istim parametrima i tada detaljnije proanalizirati
- UVM_MAX_QUIT_COUNT - ukoliko test prijavi određeni broj grešaka često nije potrebno puštati test da se završi do kraja i time trošiti vreme

Kako bi se omogućilo puštanje testova sa nasumičnim vrednostima *seed*-a, potrebno je proslediti tu informaciju prilikom pokretanja simulacije. U QuestaSim alatu se ovo radi na sledeći način:

```
vsim top_module -sv_seed random <ostali_argumenti>
```

Pošto je potrebno puštati velik broj testova, nije nužno imati otvoren GUI već se u praksi praktikuje puštanje simulacija iz komandne linije. U QuestaSim alatu se ovo radi uz "-c" ili "-batch" argument:

```
vsim -c top_module <ostali_argumenti>
```

Što se tiče prikupljanja podataka o pokrivenosti, QuestaSim alat koristi UCDB fajlove (engl. *Unified Coverage DataBase*). U njima se čuvaju informacije o raznim vrstama podataka: različitim tipovima stukturne pokrivenosti, FSM pokrivenosti, SystemVerilog grupama, *assertion* tvrđenjima, ... Podaci se u ovim fajlovima mogu čuvati prilikom eksplicitnog zahteva ili prilikom završetka simulacije:

```
GUI: Tools -> Coverage Save
CLI: coverage save top.ucdb
     coverage save -onexit top.ucdb
```

Nakon puštanja testova potrebno je spojiti podatke o pokrivenosti iz različitih testova da bi se dobile konačne informacije o statistici.

```
GUI: Verification Management Browser window
CLI: vcover merge output inputA.ucdb inputB.ucdb
```

Prilikom korišćenja GUI: nakon otvaranja *Verification Management Browser* prozora, potrebno je selektovati sve .ucdb fajlove za spajanje, a zatim desnim klikom odabrati opciju *Merge*. Otvoriće se novi prozor u kome je moguće podesiti i dodatne opcije.

Prilikom korišćenja CLI: primer komande iznad će spojiti inputA.ucdb i inputB.ucdb fajlove i rezultat upisati u output. Takođe je moguće proslediti i dodatne argumente.

Za detaljniji opis svih mogućnosti alata pogledati odgovarajući *User's Manual* (za verziju alata koja se koristi na vežbama informacije se mogu pronaći u poglavlju 25 - *Coverage and Verification Management in the UCDB*).

3 Česte greške

U ovom poglavlju su opisane česte greške koje se prave prilikom razvoja verifikacionih okruženja. Tabela ispod sadrži opis problema i primere.

<table>
<tr><td colspan="3">Upotreba signed tipova podataka</td></tr>
<tr><td>Opis</td><td colspan="2">Kako bi se izbeglo korišćenje opširnih deklaracija promenljivih, mogu se koristiti neki predefinisani tipovi</td></tr>
<tr><td>Primer</td><td colspan="2">byte umesto bit [7:0]
byte je signed tip čiji je opseg od -128 do +127, a ne od 0 do 255</td></tr>
<tr><td colspan="3">Deklaracija nizova</td></tr>
<tr><td>Opis</td><td colspan="2">Deklarisanje nizova koristeći samo jednu vrednost, a ne opseg [high:low]</td></tr>
<tr><td>Primer</td><td colspan="2">deklaracija niza x od 256 elemenata
x [256] je ekvivalentno sa x[0 : 255], a ne sa x[255 : 0]</td></tr>
<tr><td colspan="3">Podrazumevani argumenti u tasku / funkciji</td></tr>
<tr><td>Opis</td><td colspan="2">Podrazumevani tip argumenta je isti kao i prethodno naveden tip</td></tr>
<tr><td>Primer</td><td colspan="2">task example (ref int array[7], int a, b); // a i b su ref
task example (ref int array[7], input int a, b); // eksplicitno navesti da su a i b input</td></tr>
<tr><td colspan="3">Inicijalizacija promenljivih</td></tr>
<tr><td>Opis</td><td colspan="2">Problem se može javiti ukoliko se lokalna promenljiva inicijalizuje prilikom deklaracije jer u tom slučaju promenljiva inicijalizuje prilikom starta simulacije</td></tr>
<tr><td>Primer</td><td>Problem:
int x = 5;</td><td>Rešenje:
int x;
x = 5;</td></tr>
<tr><td colspan="3">Više operatora u izrazu</td></tr>
<tr><td>Opis</td><td colspan="2">U izrazu treba da postoji maksimalno jedan operator npr. $<$, $\leq$, $==$, $\geq$, ili $>$ jer se izraz podeli u više binarnih izraza koji se evaluiraju s leva na desno</td></tr>
<tr><td>Primer</td><td>Problem:
<pre>class Example;
  rand bit [7 : 0] lo, med, hi;
  constraint ex_con {lo < med < hi;}
endclass</pre>Primer rezultata prethodnog koda: lo = 20, med = 244, hi = 164 (izraz lo <med <hi postaje ((lo <med) <high); prvo se evaluira lo <med što daje rezultat 0 ili 1 i ta vrednost se poredi sa hi)</td><td>Rešenje:
<pre>class Example;
  rand bit [7 : 0] lo, med, hi;
  constraint ex_con {lo < med; med < hi;}
endclass</pre></td></tr>
<tr><td colspan="3">Korišćenje događaja u petlji</td></tr>
<tr><td>Opis</td><td colspan="2">Ukoliko se u petlji čeka na level-sensitive događaj, potrebno je preći u sledeće vreme kako bi se izbegla zero-delay petlja. Wait blokira jednom u simulacionom trenutku.</td></tr>
<tr><td>Primer</td><td>Problem:
<pre>forever begin
  wait(e1.triggered);
  process_in_zero_time();
end</pre></td><td>Rešenje:
<pre>forever begin
  @(e1);
  process_in_zero_time();
end</pre></td></tr>
<tr><td colspan="3">Razlika između “or” i “||”</td></tr>
<tr><td>Opis</td><td colspan="2">“or” je separator, a ne operacija, za razliku od “||” što je logička operacija ILI</td></tr>
</table>

<table>
<tr><td>Primer</td><td>Problem:

<pre>always @(a || b) ...</pre>
Lista je osetljiva na rezultat a ili b, a ne na bilo koju promenu signala a i b. Npr. ukoliko je a=1, a b promeni vrednost sa 0 na 1 blok se neće trigerovati</td><td>Rešenje:

<pre>always @(a or b) ...</pre>
Lista je osetljiva na promene signala a i b. Trigerovaće se na bilo koju promenu signala a i b, bez obzira na vrednost</td></tr>
<tr><td colspan="3">*Handle*, a ne objekat</td></tr>
<tr><td>Opis</td><td colspan="2">Važno je uočiti razliku između handle-a (pokazivača) i samog objekta. Pokazivač se deklariše, a objekat konstruiše. Jedan pokazivač može pokazivati na više objekata u toku simulacije.</td></tr>
<tr><td>Primer</td><td colspan="2"><pre>Transaction t1, t2; // deklaracija dva pokazivaca
t1 = new(); // alociranje prvog objekta tipa Transaction
t2 = t1; // i t1 i t2 pokazuju na isti objekat
t1 = new(); // alociranje drugog objekta tipa Transaction</pre></td></tr>
<tr><td colspan="3">Granice u for petlji</td></tr>
<tr><td>Opis</td><td colspan="2">For petlje se izvšavaju do god iterator ne dostigne zadatu vrednost, ali je moguće zadati vrednost van opsega iteratora</td></tr>
<tr><td>Primer</td><td colspan="2"><pre>bit [3 : 0] i;
for (i = 0; i < 50; i++) begin
// ...
end</pre>Simulator neće javiti grešku. Najsigurnije je ipak koristiti int za tip kod iteratora petlje.</td></tr>
<tr><td colspan="3">Provera uspešnosti randomizacije</td></tr>
<tr><td>Opis</td><td colspan="2">Zbog velikog broja ograničenja u verifikacionom okruženju i mogućnosti dodavanja novih ograničenja tokom rada, moguće je da randomizacija neće uspeti</td></tr>
<tr><td>Primer</td><td colspan="2"><pre>assert(x.randomize());</pre></td></tr>
</table>

4 Zadaci

Zadatak U pretećim materijalima za vežbu se nalazi fajl "v12_gotchas_examples.sv" u kome su prikazane često pravljene greške. Uočiti i ispraviti sve greške u kodu kako bi se u simulaciji ispravno ispisivalo 10 nasumičnih transakcija (nasumično odabran vrednosti polja *addr* i *data*) prilikom svake promene vrednosti polja *e1* i *e2*, pri čemu *data* uvek ima vrednost između 5 i 10, a *addr* 2 ili 3.

Zadatak Pustiti regresiju za primer "Calc1" dizajna uz kreirano verifikaciono okruženje. Analizirati rezultate. Generisati podatke o pokrivenosti.

5 Appendix

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

// example transaction
class Transaction extends uvm_sequence_item;

    bit [1 : 0] addr;
    bit [7 : 0] data;

    `uvm_object_utils_begin(Transaction)
        `uvm_field_int(addr, UVM_DEFAULT)
        `uvm_field_int(data, UVM_DEFAULT)
    `uvm_object_utils_end

    constraint addr_data_constraint {
        addr == 5;
        5 < data < 10;
    }

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

endclass : Transaction

// test
class test extends uvm_test;

    `uvm_component_utils(test)

    bit [2 : 0] i;
    Transaction tr_queue[$];
    bit         e1, e2;

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

    // build phase - fill tr_queue with 10 random transactions
    function void build_phase(uvm_phase phase);
        Transaction tr;
        super.build_phase(phase);

        `uvm_info(get_type_name(), "Starting build phase ...", UVM_HIGH)
        tr = Transaction::type_id::create("tr");

        for(i = 0; i < 10; i++) begin
            tr.randomize();
            tr_queue.push_back(tr);
        end
    endfunction : build_phase

    task run_phase(uvm_phase phase);
        `uvm_info(get_type_name(), "Starting run phase ...", UVM_HIGH)
        phase.raise_objection(this);

        fork
            generate_random_events();
            display_tr();
        join_any;
        disable fork;

        phase.drop_objection(this);
    endtask : run_phase
```

```
  task generate_random_events();
    forever begin
      #($urandom_range(10,1)); // delay
      if ($urandom_range(1)) begin
        `uvm_info(get_type_name(), "e1 changed...", UVM_HIGH)
        e1 = !e1;
      end
      else begin
        `uvm_info(get_type_name(), "e2 changed...", UVM_HIGH)
        e2 = !e2;
      end
    end
  endtask : generate_random_events

  task display_tr();
    Transaction tr;
    while(tr_queue.size()) begin
      @(e1 || e2);
      tr = tr_queue[i];
      `uvm_info(get_type_name(),
                $sformatf("Transaction: \n%s", tr.sprint()),
                UVM_HIGH)
    end
  endtask : display_tr

endclass : test

// top module
module top();

  initial begin
    run_test("test");
  end

endmodule : top
```

Kod 1: v12_gotchas_examples