Funkcionalna verifikacija hardvera

Vežba 3
Thread-ovi u SystemVerilog jeziku

Sadržaj

Treća vežba je posvećena korišćenju niti (engl. *threads*) u SystemVerilog jeziku, kao i komunikaciji između njih. Pored opisa samih niti i različitih načina korišćenja, opisani su događaji (engl. *events*), semafori i *mailbox* koji omugućuju komunikaciju i sinhronizaciju između njih.

Napomena: iako postoji dosta razlika između niti (*threads*) i procesa, u većini literature o SystemVerilog-u se ova dva termina podednako koriste (prvenstveno zbog naizmeničnog korišćenja u SystemVerilog LRM-u). Pa se i događaji i semafori često nazivaju objektima za inter-procesnu komunikaciju.

1 Niti

U ovom poglavlju je dat pregled niti. Opisani su različini načini kreiranja, sinhronizacija i dati primeri upotrebe.

1.1 Uvod

Testbenč je često komplesna struktura, sačinjena od više komponenti koje moraju raditi u paraleli (npr. drajver će generisati podatke na interfejs, monitor će ih prikupljati, *scoreboard* proveravati, itd.). Kako bi se omogućio nesmetan rad svih blokova, potrebno je omogućiti da se svaki zadatak izvršava nezavisno od ostatka okruženja, a to se najlakše postiže korišćenjem niti. Niti su nezavisne sekvence instrukcija koje se izvšavaju u paraleli. Olakšavaju pisanje kompleksnih verifikacionih okruženja i čine kod čitljivijim i efikasnijim.

1.2 *Fork..join*

Niti se kreiraju koristeći *fork..join* konstrukciju. Svaka naredba unutar ove konstrukcije predstavlja zasebnu nit koja će se izvršavati u paraleli sa ostatkom naredbi. U nastavku je dat primer upotrebe i rezultat izvršavanja datog koda.

```
module fork_join_example;
    initial begin
        $display("Before fork .. join at %0t", $time);
        fork

            #15 $display("1st example at %0t", $time); // posebna nit

            #5 $display("2nd example at %0t", $time); // posebna nit

            // jedna posebna nit u begin..end bloku
            begin
                $display("3rd example at %0t", $time);
                #10
                  $display("4th example at %0t", $time);
            end

        join
        // nastavak izvrsavanja
        $display("After fork .. join at %0t", $time);
    end

    // Rezultat izvrsavanja:
    //
    // Before fork .. join at 0
    // 3rd example at 0
    // 2nd example at 5
    // 4th example at 10
    // 1st example at 15
    // After fork .. join at 15

endmodule : fork_join_example
```

Kod 1: fork..join

Prethodni kodni fragment će kreirati tri niti. Primetiti da *begin..end* blok i sve naredbe koje su u njemu objedinjene kreiraju jednu nit. Takođe, primetiti da se sa nastavkom izvršavanja koda (nakon *join* naredbe) kreće tek nakon što se završe sve niti unutar *fork..join* konstrukcije.

1.3 *join_any, join_none*

Ponekad nije zgodno čekati da se sve niti završe kako bi se nastavilo sa ostatkom koda. Možda se neke niti vrte u beskonačnoj petlji, možda neke čekaju na događaje koji se nikada ne dese i sl.

Zbog toga se, osim *fork..join* konstrukcije obezbeđuju još dva načina manipulacije koda - koristeći *fork..join_any* i *fork..join_none*.

Kao i *fork..join*, niti će se kreirati na potpuno isti način. Jedina razlika je u načinu nastavka izvršavanja koda nakon *join* naredbe. *Join_any* čeka da se završi bar jedna nit, dok *join_none* uopšte ne čeka na niti i odmah kreće sa izvršavanjem ostalih naredbi. Primeri su dati ispod.

```
module fork_join_any_example;

    initial  begin
       $display("Before fork join_any at %0t", $time);
       fork
          #15 $display("1st example at %0t", $time);
          #5 $display("2nd example at %0t", $time);
          begin
             $display("3rd example at %0t", $time);
             #10;
             $display("4th example at %0t", $time);
          end
       join_any
       $display("After fork join_any at %0t", $time);
    end

    // Rezultat izvrsavanja:
    //
    // Before fork join_any at 0
    // 3rd example at 0
    // 2nd example at 5
    // After fork join_any at 5
    // 4th example at 10
    // 1st example at 15

endmodule : fork_join_any_example
```

Kod 2: join any

```
module fork_join_none_example;

    initial  begin
       $display("Before fork join_none at %0t", $time);
       fork
          #15 $display("1st example at %0t", $time);
          #5 $display("2nd example at %0t", $time);
          begin
             $display("3rd example at %0t", $time);
             #10;
             $display("4th example at %0t", $time);
          end
       join_none
       $display("After fork join_none at %0t", $time);
    end

    // Rezultat izvrsavanja:
    //
    // Before fork join_none at 0
    // After fork join_none at 0
    // 3rd example at 0
    // 2nd example at 5
    // 4th example at 10
    // 1st example at 15

endmodule : fork_join_none_example
```

Kod 3: join none

1.4 Kontrola

Kontrola nad kreiranim nitima se može vršiti čekanjem na nit ili uništavanjem niti koristeći *wait fork* i *disable fork* naredbe.

1.4.1 *Wait fork*

Wait fork naredba se koristi kako bi bili sigurni da su se sve *child* niti završile (sve niti kreirane od strane niti koja poziva *wait fork*). U primeru ispod, u *initial bloku* se prvo kreira jedna nit koristeći *fork..join_none* konstrukciju, a zatim se kreiraju još dve niti u *fork..join_any* bloku. Ove tri niti predstavljaju *child* niti *initial* bloka i *wait fork* naredba čeka na završetak sve tri niti pre nego što se nastavi izvršavanje ostalih naredbi.

```
module wait_fork_example;

    initial  begin
       fork
          #15 $display("1st example at %0t", $time);
       join_none
       $display("After fork join_none at %0t", $time);

       fork
          #5 $display("2nd example at %0t", $time);
          begin
             $display("3rd example at %0t", $time);
             #10;
             $display("4th example at %0t", $time);
          end
       join_any
       $display("After fork join_any at %0t", $time);

       wait fork;
       $display("After wait fork at %0t", $time);
    end

    // Rezultat izvrsavanja:
    //
    // After fork join_none at 0
    // 3rd example at 0
    // 2nd example at 5
    // After fork join_any at 5
    // 4th example at 10
    // 1st example at 15
    // After wait fork at 15

endmodule : wait_fork_example
```

Kod 4: wait fork

1.4.2 *Disable fork*

Disable fork naredba terminira sve *child* niti. Ukoliko te *child* niti imaju svoje *child* niti i one će biti terminirane. Primetiti da se u datom primeru naredbe koje ispisuju "1st example" i "4th example" neće izvršiti jer su niti terminirane pre nego što su stigle do ispisa ovih poruka.

```
module disable_fork_example;

    initial  begin
       fork
          #15 $display("1st example at %0t", $time);
       join_none
       $display("After fork join_none at %0t", $time);

       fork
```

```
            #5 $display("2nd example at %0t", $time);
            begin
                $display("3rd example at %0t", $time);
                #10;
                $display("4th example at %0t", $time);
            end
        join_any
        $display("After fork join_any at %0t", $time);

        disable fork;
        $display("After disable fork at %0t", $time);
    end

    // Rezultat izvrsavanja:
    //
    // After fork join_none at 0
    // 3rd example at 0
    // 2nd example at 5
    // After fork join_any at 5
    // After disable fork at 5

endmodule : disable_fork_example
```

Kod 5: disable fork

2 Inter-procesna komunikacija

Zbog velikog broja niti u većini verifikacionih okruženja i česte potrebe za međusobnom komunikacijom i sinhronizacijom, potrebno je obezbediti bezbedan i jednostavan način za pisanje *thread-safe* okruženja. Ovo se postiže korišćenjem događaja (*events*), semafora i *mailbox*-ova.

2.1 Događaji

SystemVerilog podržava *event* tip koji olakšava sinhronizaciju između niti, uz pomoć operatora "@" i "→" za čekanje na događaj i trigerovanje događaja. Događaji se deklarišu koristeći ključnu reč *event* praćenu sa imenom događaja. U sledećem primeru se u prvom *initial* bloku trigeruje *event* e1, na koji se čeka u drugom *initial* bloku.

```
module simple_event_example;

  event e1;

  initial begin
    #50;
    $display("Triggering event at %0t", $time);
    -> e1;
  end

  initial begin
    $display("Waiting for event at %0t", $time);
    @(e1);
    $display("Event observed at %0t", $time);
  end

  // Rezultat izvrsavanja:
  //
  // Waiting for event at 0
  // Triggering event at 50
  // Event observed at 50

endmodule : simple_event_example
```

Kod 6: Primer događaja

"@" operator je *edge-sensitive*, što može prouzrokovati probleme i neželjene situacije ukoliko se ne koristi pravilno. Zbog toga SystemVerilog omogućuje i korišćenje *level-sensitive* operatora *wait* i *triggered* osobine. U narednom primeru se vidi razlika između korišćenja ova dva operatora:

```
module edge_sensitive_example;

  event e1, e2;

  initial begin
    $display("Thread 1 before trigger");
    -> e1;
    @e2;
    $display("Thread 1 after trigger");
  end

  initial begin
    $display("Thread 2 before trigger");
    -> e2;
    @e1;
    $display("Thread 2 after trigger");
  end

  // Rezultat izvrsavanja:
  //
  // Thread 1 before trigger
  // Thread 2 before trigger
```

```
    // Thread 1 after trigger

endmodule : edge_sensitive_example

module level_sensitive_example;

    event e1, e2;

    initial begin
        $display("Thread 1 before trigger");
        -> e1;
        wait(e2.triggered());
        $display("Thread 1 after trigger");
    end

    initial begin
        $display("Thread 2 before trigger");
        -> e2;
        wait(e1.triggered());
        $display("Thread 2 after trigger");
    end

    // Rezultat izvrsavanja:
    //
    // Thread 1 before trigger
    // Thread 2 before trigger
    // Thread 2 after trigger
    // Thread 1 after trigger

endmodule : level_sensitive_example
```

Kod 7: Događaji osetljivi na nivo i ivicu

U prvom primeru se druga nit nikad neće završiti pošto nije uhvatila *zero-delay* događaj. Dok će se u drugom primeru izvršiti zato što *level-sensitive* naredba hvata sve događaje koji su se izvršili u tom simulacionom vremenu.

2.2 Semafori

Semafor je sinhronizacioni objekat koji omogućava ekskluzivan pristup deljenom objektu. Npr. ukoliko imamo jednu promenljivu kojoj se pristupa iz više niti, može doći do neočekivanih rezultata ukoliko, u isto vreme, jedna nit pokušava da pročita vrednost promenljive, dok druga nit vrši upis. Semafori obezbeđuju *lock* mehanizam pristupa. Nit pre pristupa proverava da li je promenljiva "zaključana", ukoliko nije, uzima ključ i vrši pristup toj promenljivoj. Do god nit ne završi sa radom i ne vrati ključ, nijedna druga nit ne može pristupati toj promenljivoj. Semafor može sadržati i više od jednog ključa. Do god ima slobodnih ključeva niti ih mogu uzimati i nastavljati svoj rad, a ukoliko ih nema, moraju čekati da se ključevi oslobode.

Postoji nekoliko metoda za rad sa semaforima:

- *new(num_of_keys)* - kreira semafor sa željenim brojem ključeva. Podrazumevana vrednost za *num_of_keys* je 0.
- *get(num_of_keys)* - traži *num_of_keys* ključeva. Ukoliko su dostupni, uzima ključeve i nastavlja se sa radom, a ukoliko nisu, tred blokira dok ne postanu dostupni. Podrazumevana vrednost za *num_of_keys* je 1.
- *put(num_of_keys)* – vraća prethodno uzete ključeve u semafor. Podrazumevana vrednost za *num_of_keys* je 1.
- *try_get(num_of_keys)* – slično kao *get()*, ali ne blokira ukoliko ključevi nisu dostupni već samo vraća 0 da signalizira ovu situaciju. Podrazumevana vrednost za *num_of_keys* je 1.

Treba obratiti paznju pri radu sa više ključeva. Moguće je vratiti više ključeva nego što je uzeto, što može prouzrokovati probleme u daljem radu.

```
module semaphore_example;

   semaphore sem;

   task sem_example();
      sem.get();
      $display("Got sem at %0t", $time);
      #50;
      sem.put();
   endtask

    initial  begin
      sem = new(1);
      fork
         sem_example();
         sem_example();
         sem_example();
      join
   end

   // Rezultaat izvrsavanja:
   //
   // Got sem at 0
   // Got sem at 50
   // Got sem at 100

endmodule : semaphore_example
```

Kod 8: Semafori

2.3 *Mailbox*

Mailbox je mehanizam koji omogućava slanje poruka iz jednog procesa ka drugom. Ponašanje *mailbox*-ova podseća na poštansko sanduče od kojeg i dobija ime. Pismo (podatak) se sprema i ostavlja u sanduče odakle se kasnije preuzima. Ukoliko prilikom provere sandučeta nijedno pismo nije stiglo može se odabrati jedna od dve akcije: sačekati da stigne pismo ili proveriti kasnije. Takođe je moguće odabrati između dva tipa sandučeta: sa neograničenim ili ograničenim kapacitetom. Ukoliko se sanduče sa ograničenim kapacitetom napuni, proces koji šalje podatak će čekati dok se podaci ne preuzmu i oslobodi prostor u sandučetu.

U SystemVerilog-u postoji veliki broj metoda za rad sa *mailbox*-ovima. Njihovi prototipi su:

- function new(int bound = 0) - konstruktor. Kreira nov *mailbox*. Kao argument prima kapacitet sandučeta. Podrazumevana vrednost je 0 odnosno neograničeno sanduče.
- function int num() - vraća broj poruka u sandučetu.
- task put(podatak) - staviti podatak u *mailbox*. Ukoliko je sanduče puno, metoda će blokirati do god ne oslobodi prostor i završi slanje.
- function int try_put(podatak) - kao *put*, ali bez blokiranja. Funkcija vraća *int* koji signalizira da li je slanje uspešno.
- task get(ref poruka) - blokirajuća metoda za preuzimanje poruke.
- function int try_get(ref poruka) - neblokirajuća metoda za preuzimanje poruke.
- task peek(ref poruka) - blokirajuća metoda koja kopira poruku, ali je ne preuzima (poruka ostaje u sandučetu).

- function int try_peek(ref poruka) - neblokirajuća verzija *peek* metode.

Mailbox je podrazumevano netipiziran odnosno može da prima bilo koji tip podatka. Iako veoma korisna, ova osobina često uvodi greške prilikom pokušaja čitanja pogrešnog tipa (npr. ukoliko se tip podatka u *get* metodi ne poklapa sa podatkom dostupnim u sandučetu). Kako bi se izbegle ove greške, *mailbox*-ove je moguće parametrizovati.

U nastavku je dato nekoliko primera upotrebe *mailbox*-ova.

```
module mailbox_example;

  mailbox mbox = new();
  int  mssg, num;

   initial  begin
      for(int  i=0; i<5; i++) begin
         #10;
         $display("Sending message %0d at %0t", i, $time);
         mbox.put(i);
      end

      num = mbox.num();
      $display("%0d packets in the mailbox at %0t", num, $time);
  end

   initial  begin
      for(int  i=0; i<5; i++) begin
         #15;
         mbox.get(mssg);
         $display("Got message %0d at %0t", mssg, $time);
      end

      #20;
      $display("Sending message 6 at %0t", $time);
      mbox.put(6);
  end

   initial  begin
      #100;
      mbox.get(mssg);
      $display("Got message %0d at %0t", mssg, $time);
       $finish ;
  end

  // Rezultat izvrsavanja:
  //
  // Sending message 0 at 10
  // Got message 0 at 15
  // Sending message 1 at 20
  // Got message 1 at 30
  // Sending message 2 at 30
  // Sending message 3 at 40
  // Got message 2 at 45
  // Sending message 4 at 50
  // 2 packets in the mailbox at 50
  // Got message 3 at 60
  // Got message 4 at 75
  // Sending message 6 at 95
  // Got message 6 at 100

endmodule // mailbox_example
```

Kod 9: Mailbox

3 Zadaci

Zadatak Kod 10 sadrži primere upotrebe niti (fajl "fork_examples.sv" u pratećim materijalima). Analizirati konstrukcije. Koji će biti rezultat izvršavanja svakog primera? Zašto?

```
module fork_examples;

  bit a, b;

  initial begin
    a = 1'b0;
    b = 1'b0;
    repeat(5) begin
      #50 a = 1'b1;
      #50 b = 1'b1;
      #50 a = 1'b0;
      #50 a = 1'b1;
      #100 b = 1'b0;
    end
  end

  initial begin
    // example 1
    fork
      forever begin
        @(a);
        $display("Example 1: Observed a change in a, new value = %b at %0t", a, $time);
      end
      @(negedge b);
    join_any
    disable fork;
    $display("Exiting example 1 at %0t", $time);

    // example 2
    fork
      begin
        fork
          forever begin
            @(a);
            $display("Example 2: Observed a change in a, new value = %b at %0t", a, $time);
          end
          forever begin
            @(b);
            $display("Example 2: Observed a change in b, new value = %b at %0t", b, $time);
          end
        join_none
      end
      begin
        #300;
        $display("Example 2: After 300 at %0t", $time);
      end
    join
    $display("Exiting example 2 at %0t", $time);

    // example 3
    fork
      begin
        #100;
        $display("Example 3: After 100 at %0t", $time);
      end
      wait(a == b);
    join
    $display("Exiting example 3 at %0t", $time);

    // example 4 - which values of i will be displayed?
    for (int i = 0; i < 3; i++) begin
      fork
```

```
            $display("Example 4: i = %0d at %0t", i, $time);
        join_none
    end
    $display("Exiting example 4 at %0t", $time);

    #200; // some delay

    // example 5 - which threads will be killed?
    disable fork;
    $display("Exiting example 5 at %0t", $time);
  end

endmodule : fork_examples
```

Kod 10: Primer

Zadatak Za primer testbenča sa prošle vežbe, modifikovati drajver i monitor klase tako da budu osetljivi na reset signal (u obe klase neka se ispiše da je reset uočen i drajver ne sme slati transakcije do god je reset aktivan). Nasumično generisati reset signal iz top modula i proveriti da li komponente ispravno reaguju.