

VEŽBA 17

U okviru ove vežbe upoznaćemo se sa internom komunikacijom između niti (Interthread Communication). Vežba je podeljena u tri dela, u prvom ćemo proći kroz bazičnu realizaciju prenosa informacija između dva System Verilog modula koji komuniciraju preko svojih IO portova, u sledećem se upoznajemo sa blokirajućom tehnikom komunikacije i na kraju radimo ne blokirajuću komunikaciju.

U direktorijumima:

17_Interthread_Communication\01_Modules,

fajl: **modules.sv** realizuje ovu klasičnu postavku.

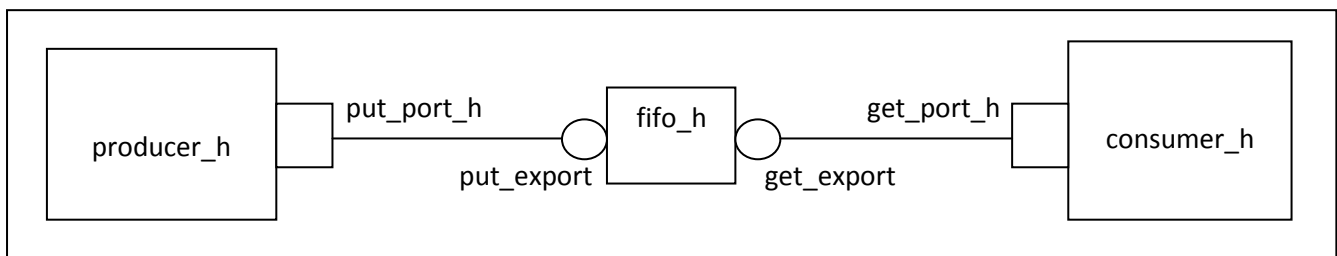
17_Interthread_Communication\02_Blocking,

fajlovi: **communication_test.svh, consumer.svh, example_pkg.sv, producer.svh, top.sv** realizuju prenos informacija u blokirajućem režimu.

Finalno u direktorijumu:

17_Interthread_Communication\03_NonBlocking

fajlovi: **communication_test.svh, consumer.svh, example_pkg.sv, producer.svh, top.sv** realizuju ne blokirajući prenos informacija.



Slika 1

U iskustvu rada sa Verilogom ili VHDL-om zapravo uvek radimo sa nitima (threads), jer svaki modul sa svojim initial blokom zapravo jeste nit za sebe. Kao što vidimo na *slici 1* želimo da imamo **producer** nit za koju smo sigurni da radi pouzdano, ili je **uvm_component**, i **consumer** nit sa kojom želimo da ostvarimo komunikaciju. Ako pogledamo naš inicijalni kod **modules.sv**,

```
module producer(output byte shared, input bit put_it, output bit get_it);
```

```
initial
```

```
repeat(3) begin
```

```
$display("Sent %0d", ++shared);
```

```
get_it = ~get_it;
```

```
@(put_it);
```

```
end
```

```
endmodule : producer
```

```

module consumer(input byte shared, output bit put_it, input bit get_it);
  initial
    forever begin
      @(get_it);
      $display("Received: %0d", shared);
      put_it = ~put_it;
    end
endmodule : consumer

module top;
  byte shared;
  producer p (shared, put_it, get_it);
  consumer c (shared, put_it, get_it);
endmodule : top

```

Imamo ovde **producer** modul i **consumer** modul, svaki od njih ima svoj initial blok, oni se pokreću u sopstvenim nitima, i svaki od njih ima signal **put_it** i **get_it** da komuniciraju kroz njihove modul portove. U modulu top smo instancirali **producer** i **consumer**. **Consumer** dolazi dole čekajući **get_it** za promenu. **Producer** generiše novi **shared** podatak, rotira **get_it** bit (promena na toj liniji) i čeka **put_it** bit, odnosno biva blokiran sve dok prijemnik ne potvrdi preuzimanje podatka sa **shared** bus-a. Rotirani **get_it** bit (promena na toj liniji) omogućava **consumer**-u da pročita **shared bus** i potom rotira **put_it** bit (promena na toj liniji) i time javlja **producer**-u da može da se deblokira, odnosno da okonča aktuelni ciklus petlje. Slanje sledećeg **shared** podatka koristi isti mehanizam. Bez ovih blokada ne bi smo imali komunikaciju zato što bi se jedna nit izvršavala sve vreme. **Blokade** predstavljaju kritičnu tačku komunikacije između niti.

Pogledajmo sada kako smo ovo izveli upotrebom klase, fajl **communication_test.svh** u folderu **02_Blocking**

```

class communication_test extends uvm_test;
  `uvm_component_utils(communication_test)

  producer producer_h;
  consumer consumer_h;
  uvm_tlm_fifo #(int) fifo_h;

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

  function void build_phase(uvm_phase phase);
    producer_h = new("producer_h", this);
    consumer_h = new("consumer_h", this);
    fifo_h = new("fifo_h", this);
  endfunction : build_phase

  function void connect_phase(uvm_phase phase);
    producer_h.put_port_h.connect(fifo_h.put_export);
    consumer_h.get_port_h.connect(fifo_h.get_export);
  endfunction : connect_phase

endclass : communication_test

```

U ovom **uvm_test** –u imamo **producer** komponentu, **consumer** komponentu i namensku UVM FIFO klasu koja se koristi pri prenosu transakcija između nezavisnih procesa, a to je **uvm_tlm_fifo**. Vidimo da je deklaracija i instanciranje ova tri bloka urađena u skladu sa dosadašnjim UVM iskustvom, konstruktor samog testa je klasičan, u **build** fazi generišemo sva tri bloka i vezujemo ih na sopstvene hendlere, dok u **connect_phase** povezujemo ulaznu i izlaznu stranu FIFO bloka na izvor odnosno prijemnik informacija.

Ako pogledamo **producer.svh**

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

  int shared;
  uvm_put_port #(int) put_port_h;

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

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

  task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    repeat (3) begin
      put_port_h.put(++shared);
      $display("Sent %0d", shared);
    end
    phase.drop_objection(this);
  endtask : run_phase
endclass : producer
```

Ovde primećujemo objekat nazvan **uvm_put_port**, tipa **int**, deklariramo promenljivu **put_port_h**. U **build_phase** ga kreiramo i potom u našoj **run_phase** koja se pokreće u sopstvenoj niti podižemo objection, zatim tri puta prosleđujemo vrednost metodom **put** iz ove klase **put_port_h.put(++shared)**. Šta se zapravo dešava? Pri prvom prolazu podatak ide pravo u **fifo**, u drugom prolazu kada se detektuje da je fifo pun nit ostaje blokirana, sve dok prijemna strana ne preuzme podatak, tek tada se naredni podatak upisuje u FIFO.

Bitan je detalj da TLM FIFO objekat može da skladišti samo jedan element, stoga je jasno da upotrebom ovako skromnog FIFO resursa naš izvor i prijemnik informacija biće blokirani slično kao u prvom delu vežbe, uz prednost da je blokiranje i deblokiranje prepušteno FIFO bloku, a ne izvoru i prijemniku direktno. Ako pogledamo naš **consumer** iz fajla **consumer.svh**:

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

  uvm_get_port #(int) get_port_h;
  int shared;
```

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

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

```
task run_phase(uvm_phase phase);
  forever begin
    get_port_h.get(shared);
    $display("Received: %0d", shared);
  end
endtask : run_phase
endclass : consumer
```

vidimo da slično kao u **producer** bloku, ovde deklariramo **uvm_get_port** objekat, u **build** fazi ga kreiramo, dok u **run_phase** pozivamo njegovu metodu **get** (**get_port_h.get(shared)**). Šta se ovde dešava; prilikom pozivanja metode **get** iz ovog **fifo** nit ostaje blokirana jer je FIFO inicijalno bio prazan, ubrzo potom **producer** upisuje prvi podatak u FIFO, prijemniku se prosleđuje podatak, prijemna nit se odblokira... Sve dok je FIFO popunjen sa jednim podatkom, i sam **producer** ostaje blokirana. Naravno ovom tehnikom preveniramo gubljenje podataka, pojednostavljujemo rukovanje, a jedina cena koju plaćamo je nešto niža brzina.

Samo povezivanje **producer** bloka, **consumer** bloka i sprežnog FIFO modula realizuje se u UVM-u unutar **connect_phase** modula u kome se obavlja povezivanje, u našem slučaju to je **communication_test.svh**

```
function void connect_phase(uvm_phase phase);
  producer_h.put_port_h.connect(fifo_h.put_export);
  consumer_h.get_port_h.connect(fifo_h.get_export);
endfunction : connect_phase
```

Pristupamo **put_port_h** i **get_port_h** objektima iz **producer** i **consumer** blokova i primećujemo da svaki od njih ima metodu **connect**. **fifo_h** iz **build_phase** ima metode članice **put_export** i **get_export** kojima se promeću podaci.

Kada se prođe kroz **connect** fazu, naši **put_port**, i **get_port** povezuju se kroz **put_export**, odnosno **get_export**, i konačno su **producer** i **consumer** spregnuti preko TLM FIFO objekta i mogu da komuniciraju u bloking režimu.

U prethodna dva dela ove vežbe obradili smo blokirajuću komunikaciju između niti. Ta tehnika radi dobro sve dok ne moramo da brinemo oko taktova ili vremena unutar simulacije. Ukoliko su izvor i prijemnik informacija apstraktni, dakle nisu referencirani na realno vreme unutar simulacije blokirajuća komunikacija predstavlja zadovoljavajuće rešenje.

Ukoliko u sistemu imamo realne sintetizibilne module koji generišu informacije na ivicu takta, to znači da naši izvori informacija nemogu da podnesu dodatne blokirajuće uslove pri prometu tih informacija – tada bi proces prenosa i analize informacija opstruirao sam izvor, što nema smisla. Stoga u trećem delu vežbe menjamo producer modul tako da mu dodeljujemo vremensku referencu, videti deo koda unutar **run_phase** niže:

```
task run_phase(uvm_phase phase);
  phase.raise_objection(this);
  repeat (3) begin
    #17; // ubačeno vremensko kašnjenje.
    put_port_h.put(++shared); // blokirajuće punjenje FIFO
```

```

    $display("%0tns Sent %0d", $time, shared);
end
#17; // ubačeno vremensko kašnjenje
phase.drop_objection(this);
endtask : run_phase

```

Praktično jedina njegova izmena je dodavanje vremenskog kašnjenja pre postavljanja podatka u FIFO i nakon toga. Ovim blokom emuliramo sintetizibilni izvor podataka.

Prijemni blok sada prilagođavamo ovakvom izvoru:

```

class consumer extends uvm_component;
  `uvm_component_utils(consumer);

  uvm_get_port #(int) get_port_h;
  virtual clk_bfm clk_bfm_i; // uvodimo varijablu tipa bfm clk_bfm_i
  int shared;

  function void build_phase(uvm_phase phase);
    get_port_h = new("get_port_h", this);
    clk_bfm_i = example_pkg::clk_bfm_i; // povezujemo se na bfm
  endfunction : build_phase

  task run_phase(uvm_phase phase);
    forever begin
      @(posedge clk_bfm_i.clk); // na svaku rastuću ivicu takta iz našeg bfm-a
      if(get_port_h.try_get(shared)) // pokušavamo da preuzmemo podatak iz FIFO
        $display("%0tns Received: %0d", $time, shared); // ako uspemo ispisujemo ga, u protivnom idemo dalje
      end
    end
  endtask : run_phase

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

endclass : consumer

```

Jasno nam je da u ovom konceptu, gde **producer** blokirajući puni FIFO, moramo obezbediti neblokirajuće, brzo čitanje FIFO, kako se ne bi desila kolizija. To realizujemo tako što **consumer** na svaku rastuću ivicu takt signala sa **bfm**-a pokušava da očita FIFO sadržaj. Reč brzo u prethodnoj rečenici podrazumeva veću učestanost pokušaja čitanja iz FIFO od učestanosti upisa koju realizuje **producer**. To je obezbeđeno time što je perioda upisa u FIFO 17 ns, dok je perioda takta na bfm-u 14ns.

Ovde skrećemo pažnju na primitivno instanciranje **bfm**-a suprotno objektno orijentisanoj tehnici koja je uvedena u vežbi 11 kada smo uveli **config_db** i uradili postavljanje našeg bfm-a u konfiguracioni database, a zatim ga po potrebi predavali testovima koji se izvršavaju (imali smo testove **random_test** i **add_test**)., iz prostog razlog što se fokusiramo na baratanje sa FIFO mehanizmom, naravno da nam nije cilj da odustanemo od OO pristupa. Ovde u top modulu imamo i deklaraciju i instanciranje **clk_bfm** –a, dok je u paketu **example_pkg** isti bfm deklarisan kao virtuelan. (ovo treba preraditi na pristup sa **config_db** iz pedagoških razloga!)