Funkcionalna verifikacija hardvera

Vežba 1
Uvod u SystemVerilog

Sadržaj

SystemVerilog jezik je nastao kao nadogradnja na Verilog 2001 jezik kako bi se poboljšala produktivnost, čitljivost i ponovna upotreba koda. Međutim, za razliku od Verilog-a i VHDL-a koji su jezici za opis hardvera i neadekvatni za verifikaciju zbog svojih ograničenih osobina u pogledu raznih programskih tehnika, SystemVerilog je prvi jezik u industriji za opis i verifikaciju hardvera (engl. *Hardware and Verification Language*, HDVL). Kombinuje osobine VHDL i Verilog-a, sa osobinama C i C++ jezika. Neke od karakteristika SystemVerilog-a su uvođenje objektno-orijentisanog mehanizma, velika mogućnost randomizacije, podrška za funkcionalnu pokrivenost (*coverage*), ...

U ovoj vežbi su objašnjene osnove SystemVerilog jezika, dat je pregled osnovnih pojmova, tipova podataka kao i poređenje sa već poznatim konstrukcijama iz VHDL jezika.

1 VHDL → SystemVerilog

U nastavku je dat pregled osnovnih pojmova u SystemVerilog-u i izvršeno poređenje sa VHDL jezikom.

Kao što se vidi na primeru 4-bitnog brojača, iako implementiraju isti model, kod pisan u VHDL-u (kod 1) i SystemVerilog-u (kod 2) se dosta razlikuje. Sintaksa SystemVerilog-a umnogome podseća na sintaksu C jezika, što daje velik kontrast VHDL-u koji po sintaksi podseća više na Pascal ili Ada jezik.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is
  port (
    clk   : in  std_logic;
    rst   : in  std_logic;
    ce_i  : in  std_logic;
    up_i  : in  std_logic;
    q_o   : out std_logic_vector(3 downto 0)
  );
end counter;

architecture rtl of counter is
  signal count_s: std_logic_vector(3 downto 0);
begin

  counter_p: process (clk)
  begin
    if clk = '1' and clk'event then
      if rst = '1' then
        count_s <= (others => '0');
      elsif ce_i = '1' then
        if up_i = '1' then
          count_s <= count_s + 1;
        else
          count_s <= count_s - 1;
        end if;
      end if;
    end if;
  end process;

  q_o <= count_s;

end rtl;
```

Kod 1: VHDL brojac

```
module counter
  (input clk,
   input rst,
   input ce_i,
   input up_i,
   output logic [3:0] q_o);

  logic [3:0] count;

  always_ff @(posedge clk) begin
    if (rst) begin
      count <= 4'b0000;
    end
    else if(ce_i) begin
      if(up_i) begin
        count <= count + 1'b1;
      end
```

```
      else begin
        count <= count - 1'b1;
      end
    end
  end

  assign q_o = count;

endmodule : counter
```

Kod 2: SystemVerilog brojac

1.1 Modul

Entity-architecture par koji se koristi u VHDL-u je objedinjen je u Verilog/SystemVerilog jeziku uvođenjem koncepta modula. Modul objedinjuje i opis interfejsa i opis funkcionalnosti. Sintaksa modula prikazana je ispod:

```
module module_name (port_list);
  input  [msb:lsb] input_port_list;
  output [msb:lsb] output_port_list;
  inout  [msb:lsb] inout_port_list;
  // ... statements ...
endmodule
```

Kao i u VHDL-u, portovi mogu biti ulazni, izlazni i ulazno-izlazni. Višebitni portovi se opisuju navođenjem opsega bita u uglastim zagradama.

1.2 *Continuous assignments*

Concurrent assignments u VHDL-u se obično nazivaju *continuous assignments* u SystemVerilog-u. Sintaksa je sledeća:

```
assign wire_variable = expression;
```

Kao i u VHDL-u, redosled ovih naredbi nije bitan. Takođe, se može uslovno dodeljivati vrednost.

```
data_o <= a when (sel = '1') else b;
```

```
assign data_o = sel ? a : b;
```

Na ovom jednostavnom primeru uslovne dodele se vrlo lako lako uočava velika sličnost sa C jezikom.

1.3 Proceduralni blokovi

Procese u VHDL-u zamenjuju *always* blokovi. Postoje četiri vrste *always* blokova.

- *always*: generalna procedura. Najčešće se koristi u verifikaciji za generisanje takta.

  ```
  always #10 clk =~clk;
  ```

- *always_ff*: za modelovanje sekvencijalne logike. Moguće limitirati reakciju bloka na određenu ivicu koristeći ključne reči *posedge* i *negedge*, za rastuću i opadajuću ivicu, respektivno.

  ```
  always_ff @(posedge clock iff reset == 0 or posedge reset) begin
    r1 <= reset ? 0 : r2 + 1;
    // ...
  end
  ```

- *always_comb*: za modelovanje kombinacione logike. U ovom bloku je lista osetljivosti implicitna i sadrži sve signale koji se koriste u bloku.

```
always_comb
  a = b & c;
```

- *always_latch*: koristi se za modelovanje lečeva.

```
always_latch
  if (load) q <= d;
```

Pored *always*, postoje još dve vrste proceduralnih blokova. To su *initial* i *final* blokovi. *Initial* blokovi se izvršavaju samo jednom, na početku simulacije. U VHDL-u bi to bio proces sa *wait* naredbom na kraju. Slično je i sa *final* blokom koji se izvršava na kraju simulacije.

1.4 Kašnjenje

Wait naredba iz VHDL se može koristiti za razne vrste kašnjenja (vremensko kašnjenje, čekanje na signal itd.). U SystemVerilogu se kašnjenje moze podeliti na tri tipa:

- Vremensko kašnjenje: specificira se # operatorom (npr. #5ns – čeka 5 ns pre izvršavanja sledeće naredbe). Moguće jedinice su fs, ps, ns, us, ms, s.
- Čekanje na događaj (engl. *event*): specificira se sa @ (npr. @(clk) – blokira izvršavanje dok se ne desi promena na clk signalu). Korišćenje *event*-ova će biti detaljno objašnjeno u vežbi o inter-proces komunikaciji.
- Čekanje na uslov: specificira se sa *wait* naredbom koja je ekvivalentna VHDL-ovoj *wait until* naredbi (npr. wait(x == 5) čeka da signal x poprimi vrednost 5)

1.5 Instanciranje

Sintaksa za instancioniranje modula je:

```
module_name instance_name (port_connecton_list)
```

Kao i u VHDL-u, prilikom instanciranja modula neophodno je navesti listu portova. Instancirani portovi se moraju poklapati sa portovima definisanim u modulu. Ovo se može postići na dva načina ili pozicioniranjem odnosno praćenjem redosleda definisanja portova u modulu ili eksplicitinim navođenjem željenog porta (pri čemu se ne mora voditi računa o redosledu). Primer instancioniranja brojača je dat ispod.

```
counter cnt_instance_1 (clk, rst, data);
counter cnt_instance_2 (.cnt_o(data), .clk(clk), .rst(rst));
```

2 Leksičke konvencije

U ovom poglavlju je dat pregled osnovnih jezičkih konvencija SystemVerilog jezika. One su veoma slične pravilima programskog jezika C.

SystemVerilog je *case-sensitive* jezik, odnosno razlikuje mala i velika slova. Npr. identifikatori "data" i "DaTa" se ne odnose na isti podatak.

2.1 Prazna mesta

Prazna mesta obuhvataju prazan karakter, tabulator ("\t") i novu liniju ("\n").

2.2 Komentari

Komentari mogu biti jednolinijski ili višelinijski. Jednolinijski komentar počinje znakom "//", dok su višelinijski komentari obuhvanjeni simbolima "/*" i "*/".

```
int x; // ovo je jednolinijski komentar
/* ovo je
 viselinijski
komentar */
```

2.3 Predstavljanje brojeva

Sintaksa predsavljanja brojeva je sledeća:

```
<velicina>'<brojni_sistem><vrednost_broja>
```

Veličina predstavlja broj bita pomoću kojih se zapisuje dati broj. Ovaj deo je opcion i ukoliko se ne navede koristi se podrazumevana veličina koja je po standardu "ne manja od 32 bita". SystemVerilog podržava predstavljanje brojeva u binarnom ('b), decimalnom ('d), heksadecimalnom ('h) ili oktalnom ('o) brojnom sistemu. Navođenje brojnog sistema je takođe opciono, a podrazumevan je decimalni brojni sistem. Jedini obavezni deo pri predstavljanju brojeva je sama vrednost broja koja se označava simbolima 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Takođe je dozvoljeno korišćenje dva specijalna simbola za nepoznato stanje ("x") i za stanje visoke impedanse ("z"). Više o korišćenju ovih vrednosti se može naći u poglavlju o tipovima podataka.

U nastavku je dato nekoliko primera predstavljanja brojeva.

```
3'b101 // trobitni binarni broj
4'hA   // cetvorobitni heksadecimalni broj
'o7    // 32-bitni oktalni broj
64     // 32-bitni decimalni broj
2'b1X  // dvobitni binarni broj, gde je donji bit nepoznat
```

2.4 Preprocesor

Kompajlerske naredbe počinju znakom " ` " (obratiti pažnju da ovo nije apostrof " ' "). Upotreba je slična C-u. Par primera je dato ispod:

```
`include file_name              // ukljuciti kod iz drugog fajla
`define macro_name macro_code // zamenjuje macro_code za macro_name
`ifdef macro_name1              // ukljucuje <source lines1> ako je macro_name1 definisan
  source lines1
`elsif macro_name2              // proizvoljan broj elsif naredbi
  source lines2
`else
  source lines3
`endif
```

```
`ifndef macro_name          // obrnuta logika od ifdef
`timescale 1ns/1ns          // specificiranje vremena
```

2.5 Sistemski pozivi

SystemVerilog sadrži veliki broj sistemskih funkcija i taskova. Prepoznaju se po karakteru "$" sa kojim počinju. Koriste se za generisanje ulaza i izlaza tokom simulacije. Nekoliko najčešće korišćenih taskova je navedeno ispod:

- $display(), $write(): služi sa ispis poruke. Nalik C-ovskoj printf funkciji, specijalni karakter "%" se koristi za prosleđivanje vrednosti. Neki od često korišćenih arugmenata su "%d" za decimalni format, "%h" za heksadecimalni, "%s" za string i "%t" za vremenski format. Dodavanjem 0 u argument ("%0d", "%0h"...) se izbegava prikaz vodećih nula. Npr.

  ```
  $display ("Primer ispisa vrednosti %d", x);
  $display ("Trenutno vreme je: %0t", $time);
  ```

- $warning(), $error(), $fatal(): taskovi za prikazivanje pronađenih problema
- $time, $realtime, $stime: vraća trenutno simulaciono vreme, kao 64-bitni integer, realni broj ili 32-bitni integer respektivno
- $finish(): služi sa završetak simulacije
- $random, $urandom, $urandom_range: funkcije za generisanje random brojeva. Biće detaljno objašnjeni u vežbi o randomizaciji
- $fopen, $fdisplay, $fstrobe, $fmonitor i $fwrite: funkcije za rad sa fajlovima

3 Tipovi podataka

Tipovi podataka u SystemVerilog-u predstavljaju kobinaciju Verolog-ovih i C-ovskih tipova. U nastavku je dat pregled najčešće korišćenih tipova podataka. Za potpunu listu konsultovati SystemVerilog LRM.

3.1 Celobrojni tipovi

SystemVerilog podržava 4 logičke vrednosti. To su logička nula (0), logička jedinica (1), nepoznato stanje (X) i stanje visoke impedanse (Z).

Celobrojni tipovi podataka se mogu podeliti u dve grupe: sa 2 stanja ili sa 4 stanja. Tipovi sa dva stanja mogu imati samo dve vrednosti: 0 ili 1, dok tipovi sa 4 stanja podržavaju sve logičke vrednosti. Jedan od najbitnijih razloga za korišćenje tipova sa 2 stanja je što oni zauzimaju 50% manje memorije i čine simulaciju mnogo bržom u odnosu na tipove sa 4 stanja. Zbog toga se, gde god je to moguće, koriste tipovi sa 2 stanja.

Napomena: Treba obratiti pažnju pri kombinovanju ove dve grupe. Ukoliko se promenljivoj sa dva stanja dodeli X ili Z vrednost, biće konvertovani u nulu što može dovesti do neočekivanih grešaka. Npr.

```
bit a = 1'bX; // a ce dobiti vrednost 0
```

Takođe, nalik C-ovskim tipovima, celobrojni tipovi mogu biti *signed* ili *unsigned*. Podrazumevani znak promenljive se može promeniti eksplicitnim navođenjem željenog znaka. Na primer:

```
int data;  // default signed
int unsigned data; // declared as unsigned
```

U tabeli 1 je dat pregled celobrojnih tipova.

Naziv tipa	**Broj stanja**	**Veličina**	**Znak**
bit	2	definiše korisnik	*unsigned*
byte	2	8 bita	*signed*
shortint	2	16 bita	*signed*
int	2	32 bita	*signed*
longint	2	64 bita	*signed*
logic	4	definiše korisnik	*unsigned*
reg	4	definiše korisnik	*unsigned*
integer	4	32 bita	*signed*
time	4	64 bita	*unsigned*

Tabela 1: Celobrojni tipovi podataka

Logic i *reg* su veoma slični tipovi podataka. *Logic* tip je uveden u SystemVerilog zbog čestih zabuna sa korišćenjem *reg* tipa u Verilogu (zbog imena se često pretpostavlja da ovaj tip predstavlja registar, iako to nije uvek slučaj). Preporuka je koristiti *logic* umesto *reg* tipa.

Veličinu pojedinih tipova može da definiše korisnik, navođenjem opsega u uglastim zagradama, npr:

```
bit  [7:0] x; // 8 bita
logic  [15:0] y; // 16 bita
```

3.2 Realni tipovi

SystemVerilog podržava nekoliko realnih tipova, po uzoru na C programski jezik:

- *shortreal*: nalik C-ovskom *float* tipu
- *real*: *double* u C-u
- *realtime* : identično *real* tipu

3.3 String

SystemVerilog sadrži i *string* tip koji predstavlja dinamički alociran niz bajtova promenljive veličine. Npr.

```
string test_name = "sample name";
```

Postoji veliki broj ugrađenih metoda za manipulaciju stringovima među kojima su i *len()*, *to_upper()*, *to_lower()*, *compare()*, *substract()* i mnoge druge. Pristupa im se preko "." operatora, npr:

```
string new_test_name = test_name.to_upper(); // new_test_name dobija vrednost "SAMPLE NAME"
```

3.4 Definisanje novih tipova

Kao i u C jeziku, i u SystemVerilog-u se definisanje novih tipova vrši naredbom *typedef*.

3.5 Enumeracije

Enumeracije omogućuju da se imena dodeljuju numeričkim vrednostima. Prilikom deklaracije enumeracije moguće je izostaviti tip i tada se koristi podrazumevana vrednost int. Takođe je moguće, ali nije obavezno, navesti željenu numeričku vrednost. Korisna je metoda *name()* koja vraća naziv umesto brojne vrednosti.

```
typedef enum {IDLE, SEND, RECEIVE} state_e;
// ...
enum bit {READ = 1, WRITE = 0} op;
state_e state;
```

4 Operatori

Operatori u SystemVerilogu preuzimaju operatore iz Veriloga i nadograđuju ih C-ovskim operatorima. U nastavku je dat pregled operatora.

```
assignment_operator ::=
    = | += | -= | *= | /= | %= | &= | |= | ^= | <<= | >>= | <<<= | >>>=

conditional_expression ::=
    cond_predicate ? { attribute_instance } expression : expression

unary_operator ::=
    + | - | ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~

binary_operator ::=
    + | - | * | / | % | == | != | === | !== | =?= | !?= | && | || | **
    | < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | << | >>> | <<<

inc_or_dec_operator ::= ++ | --

unary_module_path_operator ::=
    ! | ~ | & | ~& | | | ~| | ^ | ~^ | ^~

binary_module_path_operator ::=
    == | != | && | || | & | | | ^ | ^~ | ~^
```

Pri korišćenju binarnih operatora gde je jedan od operanda tip sa 2 stanja, a drugi sa 4 (npr. *bit* i *logic* ili *int* i *integer*) uvek "pobeđuje" tip sa 4 stanja, odnosno rezultat binarnog operatora gde je jedan operator tipa *logic*, a drugi tipa *bit* je tipa *logic*, a ukoliko je jedan operand tipa *integer*, a drugi *int* rezultat je *integer*.

Postoje dve vrste operatora jednakosti: *logical equality* i *case equality*. Operatori logičke jednakosti/nejednakosti "!=" i "==" vraćaju X ako bilo koji operand sadrži X ili Z (bez obzira da li je jednakost/nejednakost tačna). Međutim, ukoliko se rezultat konvertuje u tip *bit* (npr. u *if* uslovu), X se konvertuje u 0, što može dovesti do neočekivanih rezultata. Sa druge strane operatori "!==" i "===" uključuju i X i Z vrednosti u poređenje i uvek vraćaju 0 ili 1. Primeri korišćenja ovih operatora su dati ispod.

```
bit x;
logic y;
x = (3'bZ01 == 3'bZ01); // x dobija vrednost 0
y = (3'bZ01 == 3'bZ01); // y dobija vrednost X
x = (3'bZ01 === 3'bZ01); // x dobija vrednost 1
y = (3'bZX1 === 3'bZ01); // y dobija vrednost 0
if (3'bZ01 == 3'bZ01) begin // uslov nije ispunjen
 ... // telo se nece izvriti
end
if (3'bZ01 === 3'bZ01) begin // uslov je ispunjen
 ... // telo ce se izvriti
end
```

4.1 Konkatenacija

Konkatenacija je izražena u vitičastim zagradama, a izrazi unutar zagrada su odvojeni zarezima. Npr. $\{4'b1001, 4'b10x1\}$ je 100110x1, odnosno $\{$"*primer*", "_", "*konkatenacije*" je "primer_konkatenacije".

4.2 Replikacija

Operator replikacije ima sledeću sintaksu: $\{n\{m\}\}$, gde se vrednost m replicira n puta. Npr. $\{4\{4'b1001\}\}$ je 1001100110011001.

4.3 Uslovni operator

Kao i u C-u, uslovni operator ima sledeću sintaksu:

```
uslov ? tacan_iskaz : netacan_iskaz
```

Ovaj operator se često sreće *assign* naredbama. Npr.

```
assign data_o = sel ? a : b; // data_o dobija vrednost a ukoliko je sel jednak
                             // jedinici , b u suprotnom
```

5 Proceduralne naredbe i kontrola toka

U ovom poglavlju je dat pregled proceduralnih naredbi u SystemVerilog jeziku, objašnjene su osnovne naredbe selekcije i petlje.

Proceduralne naredbe u SystemVerilog-u su:

- *initial*: naredba se izvršava jednom na početku simulacije
- *final*: naredba se izvšava jednom na kraju simulacije
- *always*: naredba se uvek izvršava
- *functions, tasks*: naredbe se izvršavaju prilikom poziva funkcije ili taska

5.1 Blokirajuća i neblokirajuća dodela

SystemVerilog podržava dva načina dodele vrednosti: blokirajući i neblokirajući. Blokiraćuje naredbe se moraju izvršiti pre sledeće naredbe u sekvencijalnom bloku odnosno dodele su trenutne. Neblokirajuće naredbe se izvršavaju u isto vreme kada se evaluiraju sve računice sa desne strane naredbe. Redosled kod neblokirajućih naredbi nije bitan. I ovde se može povući paralela sa VHDL jezikom, odnosno dodelama vrednosti signalima i promenljivima. Dozvoljeno je i dodavanje kašnjenja ovim naredbama. Sintaksa je:

```
<l_vrednost> = <kasnjenje> <iskaz>  // blokirajuca dodela
<l_vrednost> <= <kasnjenje> <iskaz> // neblokirajuca dodela
```

Primer:

```
module assignment;

  logic a, b, c, d, e, f;

  initial begin
    a = #10 1'b1; // a postaje 1 u vremenu 10
    b = #20 1'b0; // b postaje 0 u vremenu 30
    c = #40 1'b1; // c postaje 1 u vremenu 70
  end

  initial begin
    d <= #10 1'b1; // d postaje 1 u vremenu 10
    e <= #20 1'b0; // e postaje 0 u vremenu 20
    f <= #40 1'b1; // f postaje 1 u vremenu 40
  end

  always @(a, b, c, d, e, f) begin
    $display("Vreme %0t:", $time);
    $write("\ta = %0d,", a);
    $write("\tb = %0d,", b);
    $write("\tc = %0d,", c);
    $write("\td = %0d,", d);
    $write("\te = %0d,", e);
    $write("\tf = %0d\n", f);
  end

  // Rezultat izvrsavanja:
  //
  // Vreme 10:
  //    a = 1,   b = x,   c = x,   d = x,   e = x,   f = x
  // Vreme 10:
  //    a = 1,   b = x,   c = x,   d = 1,   e = x,   f = x
  // Vreme 20:
  //    a = 1,   b = x,   c = x,   d = 1,   e = 0,   f = x
  // Vreme 30:
```

```
    //    a = 1,  b = 0,  c = x,  d = 1,  e = 0,  f = x
    // Vreme 40:
    //    a = 1,  b = 0,  c = x,  d = 1,  e = 0,  f = 1
    // Vreme 70:
    //    a = 1,  b = 0,  c = 1,  d = 1,  e = 0,  f = 1

endmodule : assignment
```

Kod 3: Primer dodele

5.2 Naredbe selekcije

Naredbe selekcije su *if-else* i *case* naredbe. Iste su kao naredbe u C jeziku. Vitičaste zagrade u C-u zamenjuju ključne reči *begin..end* (primetiti da je, kao i u C-u, moguće izostaviti *begin..end* ukoliko blok sadrži samo jednu liniju). Primer je dat ispod.

```
if (x == 0)
  y = 1;
else begin
  y = 5;
  x = 2;
end

// ...

case(x)
  0, 1, 2 : y = 4;
  3 : y = 1;
  default : y = 0;
endcase
```

5.3 Petlje

Naredbe petlji su *while, do while, for, foreach, repeat* i *forever*.

Upotreba *while, do while* i *for* petlje je identična C-ovksim naredbama. *Foreach* naredba služi za iteraciju članova niza i biće objašnjena u poglavlju o nizovima.

Repeat petlja služi za ponavljanje naredbi tačno definisan broj puta.

Forever petlja služi za beskonačno ponavljanje naredbi. Ekvivalentna je *while*(1) petlji.

Sintaksa i par primera petlji je dato ispod:

```
// SystemVerilog podrzava deklarisanje
// promenljive unutar petlje
for (int i = 0; i < 15; i++) begin
  // ...
end

do begin
  // ...
end
while (x > 5);

while (x > 8) begin
  // ...
end

repeat(3) begin
  // ...
end
```

```
forever begin
  // ...
end
```

5.4 Naredbe skoka

Naredbe skoka (engl. *jump*) su *return*, *break* i *continue*. Upotreba je ista kao i u C jeziku.

5.5 Iff

Ključna reč iff omogućava detaljniju kontrolu događaja. Događaj (*event*) će se trigerovati jedino ako je uslov ispunjen. Npr.:

```
@ (posedge clk iff x == 0); // okida se na svaku ivicu takta ukoliko je x nula
```

5.6 Taskovi i funkcije

Taskovi i funkcije, kao i u većini jezika, pružaju mogućnost grupisanja dugih ili često korišćenih delova koda u jedan blok koji se lako poziva iz bilo kog dela koda, olakšava debagovanje i čini kod čitljivijim. I taskovima i funkcijama se mogu prosleđivati *input*, *output*, *inout* ili *ref* argumenti. Razlika je sledeća:

- *input* – kopira prosleđenu vrednost na početku rutune i nadalje koristi tu vrednost
- *output* – kopira vrednost na kraju rutine, obično rezultat izvršavanja rutine
- *inout* – kopira *in* na početku i *out* na kraju
- *ref* – prosleđivanje po referenci

Navođenje smera argumenta je opciono. Ukoliko se ne navede, koristi se podrazumevana vrednost *input*. Međutim, ukoliko se jednom navede smer, svi naredni argumenti će podrazumevati taj smer (pogledati example_task1 ispod). Takođe je navođenje tipa opciono. Podrazumevana vrednost je *logic*. Moguće je navesti podrazumevanu vrednost argumenta koja će se koristiti ukoliko se prilikom poziva taska ili funkcije ne navede vrednost.

Primeri taskova i funkcija:

```
module task_func;

  task example_task (input logic x, inout int y);
    $display("example_task: x = %0d, y = %0d", x, y);
    y += x;
    #5;
    // ...
  endtask : example_task

  task example_task1 (a, b, output integer c, d);
    // a i b su input logic
    // c i d su output integer
    // ...
    $display("example_task1: a = %0d, b = %0d, c = %0d, d = %0d", a, b, c, d);
    c = a;
    d = b;
  endtask

  function void example_func (int x, int y = 5);
    $display("example_func: x = %0d, y = %0d", x, y);
    // #5ns; nije moguce u funkciji
```

```
    // ...
  endfunction

  function int example_func1 ();
    // ...
    return 4;
  endfunction : example_func1

  initial begin
    int x0, x1;
    integer y0, y1;

    // primeri poziva
    example_task(5, x0);
    $display("nakon poziva example_task (%0t): x0 = %0d", $time, x0);

    example_task1(.a(4), .c(y0), .b(x1), .d(y1));
    $display("nakon poziva example_task1 (%0t): y0 = %0d, y1 = %0d", $time, y0, y1);

    example_func(3);
    x0 = example_func1();
    void'(example_func1()); // ignorisati povratnu vrednost
  end

  // Rezultat izvrsavanja:
  //
  // example_task: x = 1, y = 0
  // nakon poziva example_task (5): x0 = 1
  // example_task1: a = 0, b = 0, c = x, d = x
  // nakon poziva example_task1 (5): y0 = 0, y1 = 0
  // example_func: x = 3, y = 5

endmodule : task_func
```

Kod 4: Primeri taskova i funkcija

Iako je uloga taskova i funkcija ista, postoji nekoliko bitnih razlika:

- Funkcije se moraju izvršiti u jednom simulacionom trenutku. Taskovi ne. Ovo znači da funkcije ne mogu sadržati naredbe koje konzumiraju simulaciono vreme (npr. naredbe kašnjenja ili čekanja na događaj)
- Funkcija ne može pozivati task, ali task može pozivati funkciju kao i ostale taskove.
- Funkcija vraća vrednost, task ne. U funkciji je moguće definisati povratni tip i zatim *return* naredbom vratiti vrednost. Dozvoljen je i tip *void* ukoliko funkcija ne vraća vrednost. Taskovi smeju da sadrže naredbu *return* (za rano prekidanje taska), ali ne mogu vraćati vrednost kao funkcije. Međutim, ovo se lako prevazilazi korišćenjem *output* argumenata.
- Taskovi se moraju pozivati u posebnoj naredbi. Ne mogu biti deo komplikovanijih izraza, dok funkcije mogu (npr. if(example_func1() == 5) ...)

5.7 Vreme

Timeunit i *timeprecision* (ili obuhvaćene u *timescale*) naredbe omogućavaju specificiranje vremena. Kao što smo već rekli, kašnjenje se navodi korišćenjem znaka "#" i moguće je eksplicitno zadati željeno kašnjenje (npr. #5ns, #2s, #4ms). Međutim, moguće je i navesti kašnjenje bez jedinice (npr. #1) pri čemu se koristi jedinica specificirana pomoću *timeunit* naredbe. Npr. ukoliko je *timeunit* 100ps, onda #1 postaje kašnjenje od 100ps. Sa druge strane, *timeprecision* određuje najmanje kašnjenje sa kojim se može konfigurisati dato vreme, odnosno koliko je decimalnih mesta moguće koristiti u odnosu na specificirani *timeunit*. Ukoliko je npr. *timeunit* 100ps, a *timeprecison* 10ps, #1 predstavlja kašnjenje od 100ps, dok je #0.1 najmanje kašnjenje koje se može specificirati. Ukoliko se navede kašnjenje od npr. #15.39, ono će biti zaokruženo na #15.4. Lako se može

zaključiti da *timeunit* ne sme biti manji od *timeprecison*. *Timescale* naredba obuhvata *timeunit* i *timeprecision* naredbe u jednu i specificira se u formatu: "timescale unit/precision". U nastavku je dato još nekoliko primera:

```
`timescale 10ps/1fs // #1 je kanjenje od 10ps, a #0.0001 je najmanje merljivo kasnjenje
`timescale 1ns/1ps // #1 je kanjenje od 1ns, a #0.001 je najmanje merljivo kasnjenje
```

6 Primer jednostavnog testbenča

U nastavku je dat primer jednostavnog testbenča, koji instancira brojač sa početka vežbe. Testbenč sadrži dva *initial* bloka, jedan za dodelu vrednosti signalima, a drugi za završetak simulacije i ispis poruka. Primetiti da je redosled navođenja ovih blokova nebitan. U *always* bloku se generise takt, dok se u *final* bloku ispisuje poruka pri završetku simulacije. Funkcija *compare_values* poredi dva prosleđena argumenta.

```
module simple_tb;

  logic clk;
  logic rst;
  logic ce;
  logic up;
  logic [3 : 0] data;

  counter cnt_inst (clk, rst, ce, up, data);

  function void compare_values(logic [3:0] expected, logic [3 : 0] received);
    if (expected !== received) begin
      $error("Error in comparison: expected %0h, received %0h\n", expected, received);
    end
    else begin
      $display("Successful comparison at time %0t with value %0h", $time, expected);
    end
  endfunction : compare_values

  initial begin
    $display("Starting simulation ... ");
    #500ns;
    $finish;
  end

  initial begin
    repeat(3) @(posedge clk iff !rst);
    compare_values('he, data);
  end

  initial begin
    clk <= 0;
    rst <= 1;
    up <= 0;
    ce <= 1;
    #50ns rst <= 0;
  end

  always begin
    #5ns clk <= ~clk;
  end

  final begin
    $display("Ending simulation at time %0t", $time);
  end

endmodule : simple_tb
```

Kod 5: Primeri jednostavnog testbenča

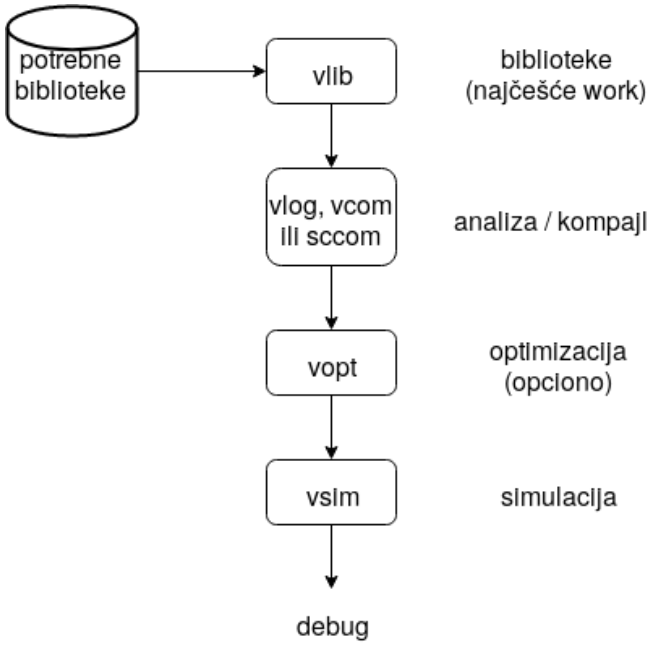
7 Pokretanje simulacije

Za simulaciju će se koristiti alat QuestaSim, razvijen od strane Mentor Graphics-a. QuestaSim je poslednji alat od Mentor Graphics-a za funkcionalnu verifikaciju. Podržava simulaciju najnovijih standarda za SystemVerilog, SystemC, Verilog 2001 i VHDL. Predstavlja nadogradnju ModelSim-a i pruža podršku za napredne verifikacione koncepte koje ModelSim ne podržava. Npr. verifikacija vođena sakupljanjem pokrivenosti (engl. *coverage driven verification*), rad sa *assertion*-ima i puno korišćenje mogućnosti SystemVerilog-a u pogledu randomizacije i ograničenja.

U QuestaSim-u se osnovna simulacija vrši prateći *flow* prikazan na slici 1. Potrebni koraci su:

1. mapiranje biblioteka
2. kompajliranje
3. opciona optimizacija
4. učitavanje dizajna u simulator
5. puštanje simulacije
6. *debug*

Napomena: optimizacija u trećem koraku može znatno ograničiti mogućnosti *debug*-a, ali ubrzati simulaciju i uštedeti na memorijskom prostoru, pa se zato mora pažljivo koristiti.



Slika 1: QuestaSim flow

Za sve navedene korake postoje komande u Questasim-u (*vlib, vlog, vsim, ...*). Najčešće se komande grupišu u skriptu radi lakše upotrebe, ali je moguće unositi ih direktno u terminal ili koristiti GUI. U nastavku je dat opis komandi potrebnih za današnje vežbe.

7.1 Biblioteke

Pre kompajliranja sors koda, mora se kreirati biblioteka u kojoj će se čuvati svi kompajlirani fajlovi. Obično se biblioteka zove *work* i ukoliko se eksplicitno ne navede naziv podrazumeva se *work*.

- GUI: File → New → Library
- CLI: vlib <lib_name>

7.2 Kompajliranje

Za kompajliranje dizajna postoje različite komande na osnovu jezika u kome je implementiran.

- GUI: Compile → Compile
- CLI: vlog (Verilog), vcom (VHDL), sscom (SystemC) praćeno listom fajlova

Za kompajliranje SystemVerilog fajlova se takođe koristi naredba *vlog*, ali uz -sv argument. Npr.:

```
vlog -sv counter.sv simple_tb.sv
```

Korisni argumenti su i -work <ime_biblioteke>kako bi se rezultati smeštali u željenu biblioteku, +incdir+<ime_direktorijuma>za uključivanje dodatnog foldera i +acc kako bi se omogućio pristup svim signalima u dizajnu.

Prilikom kompajliranja QuestaSim vrši ispis svih kompajliranih modula, ali i navodi ime top modula. Taj top modul ćemo koristiti za simulaciju u narednom koraku. Na primer:

```
# QuestaSim vlog ...
# -- Compiling module counter
# -- Compiling module simple_tb
#
# Top level modules:
#        simple_tb
```

7.3 Simulacija

Učitavanje u simulator:

- GUI: Simulate → Start simulation
- CLI: vsim <top_name>

Simulatoru se prosleđuje ime top modula koji želimo da simuliramo.

Kada se dizajn učita u simulator, simulaciono vreme je nula i čeka se na dalje komande za pokretanje simulacije. Ovo se vrši naredbom *run*. Moguće je pustiti simulaciju do kraja ili na određeni vremenski period.

```
run -all
run 10us
```

7.4 Naredba do

Korisna naredba u QuestaSim-u je *do* koja služi za izvršavanje naredbi iz fajla. Da bi se ubrzao proces kompilacije i simulacije, obično se ove komande grupišu u skriptu koju je tada lako pozivati iz alata (komentari započinju simbolom "#"). Za prethodni primer testbenča:

```
# Kreiranje biblioteke
vlib work

# Kompajl
vlog +acc -sv v1_counter.sv v1_simple_tb.sv

# Pokretanje simulacije
vsim simple_tb
```

Kod 6: do fajl

Kako bi smo izvršili ove naredbe, potrebno je u Questi pozvati

```
do <ime_fajla>
```

7.5 Debug

QuestaSim pruža veliki broj alata za analizu i debug dizajna. Oni uključuju gledanje *waveform*-a, postavljanje *breakpoint*-a, analizu povezanosti, praćenje pokrivenosti, ... Većina će biti pokazana u narednim vežbama, dok ćemo se ovde, zbog jednostavnosti primera, zadržati na gledanju *waveform*-a.

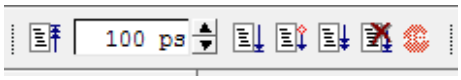
Waveform-i se prikazuju u tzv. *wave* prozoru koji se aktivira sa naredbom:

- GUI: View → Wave
- CLI: view wave

Dodavanje signala se može izvršiti na nekoliko načina:

- iz glavnog prozora, desnim klikom na željeni objekat i odabrati Add → To Wave → .... Poslednji korak su opcije koje označavaju opseg signala koje želimo da dodamo
- dodavanjem pojedinačnih signala iz *Objects* prozora na isti način
- *drag and drop*
- komandom "add wave <putanja>" (podržava *wildcard* "*" za dodavanje više signala)

Nakon učitavanja željenih signala, iz *toolbar*-a je lako puštati simulaciju. Simulacija se može puštati do kraja ili određeni vremenski period, može se prekidati, nastavljati, restartovati itd. (slika 2).



Slika 2: QuestaSim run toolbar

Analizu signala olakšava zumiranje, praćenje ivica, praćenje kursora koji se takođe lako kontorlišu iz *toolbar*-a (slika 3)

Slika 3: QuestaSim zoom toolbar

Nakon ubacivanja signala u *wave* prozor i svih dodatnih podešavanja, moguće je sačuvati sva podešavanja u *do* fajl. Podrazumevano ime je "wave.do" i izvršava se kao i svi *do* fajlovi:

```
do wave.do
```

QuestaSim pruža velike mogućnosti rada sa *waveform*-ima, uključujući i poređenje signala, čuvanje podešavanja za buduću upotrebu, merenje vremena, ... Za detaljan opis svih mogućnosti konsultovati *User Manual*, koji se može naći u QuestaSim-u (Help → Questa Documentation).

8 Zadaci

Zadatak Koristeći primer jednostavnog testbenča za brojač, pokrenuti simulaciju i analizirati rezultate. Uočiti razlike između *initial*, *always* i *final* blokova. Pogledati *waveform*-e. Pogledati kako menjanje *timescale*-a utiče na simulaciju.

Zadatak Napisati blok za generisanje stimulusa odnosno generisanje signala *ce_i* i *up_i*.

Zadatak Napisati monitor blok za proveru vrednosti brojača. Vrednosti porediti funkcijom *compare_values*.