Opis standardnih sekvencijalnih mreža korišćenjem VHDL jezika – 3

Memorije

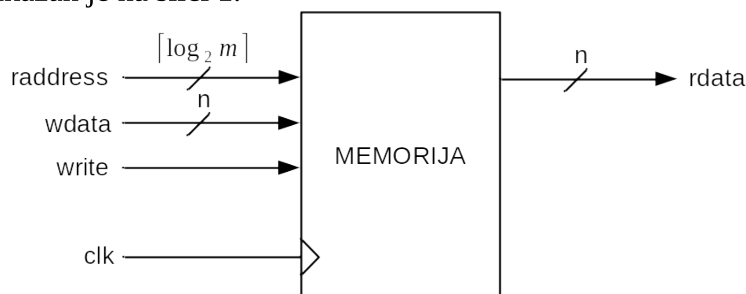
Memorija predstavlja veliki broj individualnih registara organizovanih u jednu celinu, kojima se pristupa preko zajedničkih portova. U tom smislu memorija se može posmatrati kao velika registarska banka. Ovo je samo donekle tačno, jer postoje bitne razlike između memorije i registarske banke:

- Kapacitet memorije je znatno veći od kapaciteta registarske banke. Kapacitet registarske banke iznosi nekoliko desetina registara (tipično je reč o 32 registra), dok se kapacitet memorije meri milionima, pa i milijardama registara.
- Registarska banka po pravilu ima veći broj pristupa za upis i čitanje podataka (tipično 2 pristupa za čitanje i 1 pristup za upis), dok memorija gotovo uvek ima samo 1 pristup koji se koristi i za čitanje i za upis. U jednom trenutku može se ili čitati sadržaj iz memorije ili vršiti upis novog sadržaja, ali nije moguć istovremeni upis i čitanje.
- Tehnološki, registarska banka je gotovo uvek „izgrađena“ od flip flopova. U slučaju memorija tehnologija izrade memorijskih ćelija može biti raznolika. Upravo zbog ovoga, individualni registri od kojih je memorija sastavljena se u slučaju memorije ne nazivaju registrima već *memorijskim lokacijama*.

Takođe, za razliku od registarskih banki u koje je uvek moguće i upisivati i čitati podatke, memorije se mogu podeliti u dve veliku grupe u zavisnosti od toga da li je upis podataka moguć ili nije:

- RAM memorije – memorije kod kojih je moguć upis podataka u memoriju tokom rada
- ROM memorije – memorije kod kojih je moguće samo čitati podatke

Razmotrimo prvo RAM memorije. Osnovni interfejs RAM memorije sastavljene od *m n*-bitnih lokacija prikazan je na slici 1.



Slika 1. Interfejs memorije kapaciteta m n-bitnih memorijskih lokacija sa jednim pristupom za čitanje ili upis podataka

RAM memorija u opštem slučaju poseduje sledeće portove:

- 1 pristupa za čitanje ili upis podataka upisanih u memorijske lokacije,

- *clk* ulaz za sinhronizaciju rada registarske banke

Opciono, RAM memorija može posedovati i sledeće portove:

- *reset* ulazni port za inicijalizaciju sadržaja memorijskih lokacija
- *clock enable* ulazni port za selekciju rastućih ivica *clk* porta na koje memorija treba da se aktivira
- *enable* ulazni port za dozvolu rada memorije

Pristup za čitanje ili upis podataka sastoji se iz sledećih portova:

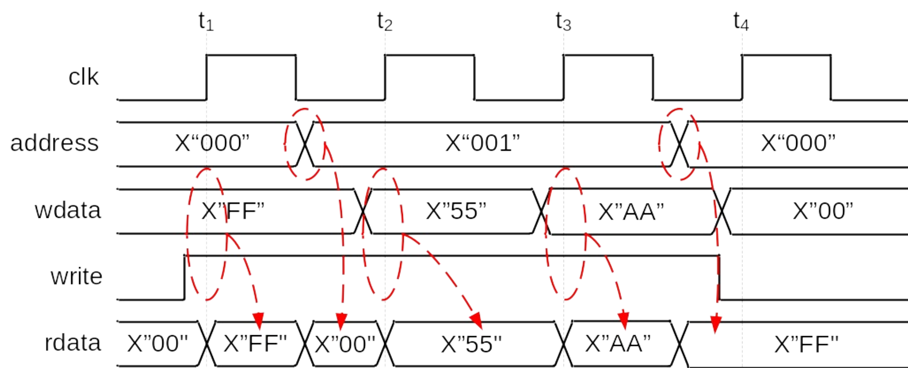
- *address*, ulazne adresne magistrale koja služi za adresiranje memorijske lokacije čiji sadržaj se želi pročitati ili se u nju želi upisati novi sadržaj; moguće vrednosti adresa su iz opsega [0 *m*-1] celih brojeva
- *rdata*, izlazne magistrale podataka preko koje se dobija trenutni sadržaj adresirane memorijske lokacije
- *wdata*, ulazne magistrale podataka preko koje se prosleđuje podatak koji je potrebno upisti u adresiranu memorijsku lokaciju
- *write*, ulaznog porta dozvole upisa novog podatka u adresiranu memorijsku lokaciju
- *read*, opcionog ulaznog porta koji inicira proces čitanja podatka iz adresirane memorijske lokacije

Kao i u slučaju registarskih banki, u zavisnosti od trenutka pojavljivanja sadržaja adresirane memorijske lokacije na *rdata* portu, razlikujemo dva načina čitanja podataka iz memorije:

- **sinhrono čitanje** - kod kojega se sadržaj adresirane memorijske lokacije pojavlje na *rdata* portu prilikom nailaska sledeće rastuće ivice *clk* signala,
- **asinhrono čitanje** - kod kojega se sadržaj adresirane memorijske lokacije pojavljuje odmah nakon stabilizacije nove adrese na *raddress* portu.

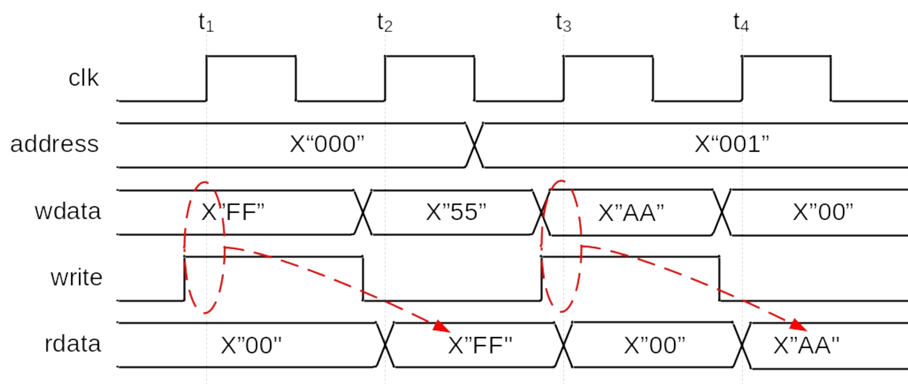
Upis podataka u memoriju vrši se preko pristupa za upis. Postavljanjem adrese lokacije u koju želimo upisati podatak na *address* ulazni port, adresira se željena memorijska lokacija. Podatak koji se želi upisati u adresiranu memorijsku lokaciju postavlja se na *wdata* ulazni port. Sam upis podatka je uvek sinhroni sa *clk* signalom, i aktivira se kada se *write* ulazni port postavi na vrednost 1.

Na primer, vremenski dijagram rada 8-bitne RAM memorije kapaciteta 1024 lokacije, sa asinhronim čitanjem, prikazan je na slici 2.



Slika 2. Vremenski dijagram rada 8-bitne RAM memorije kapaciteta 1024 lokacije, sa asinhronim čitanjem

Vremenski dijagram rada iste 8-bitne RAM memorije, ali ovaj put sa sinhronim čitanjem, prikazan je na slici 3.



Slika 3. Vremenski dijagram rada 8-bitne RAM memorije kapaciteta 1024 lokacije, sa sinhronim čitanjem

Kao i u slučaju registarskih banki, a što se sa slika 25 i 26 može i videti, razlika između memorija sa asinhronim i sinhronim čitanjem ogleda se u brzini odziva memorije prilikom izvršavanja operacije čitanja. U slučaju asinhronog čitanja, željeni podatak dostupan je u istom taktu kada se inicira proces čitanja, dok je kod sinhronog čitanja željeni podatak dostupan tek u sledećem taktu. Memorije sa sinhronim čitanjem unose kašnjenje prilikom operacije čitanja od 1 periode *clk* takt signala.

VHDL modeli RAM memorije

Da bi smo ilustrovali različite načine modelovanja RAM memorije, koristićemo dve verzije 32-bitne RAM memorije sastavljene od 1024 lokacije:

- verzija 1 – RAM memorija sa asinhronim čitanjem,
- verzija 2 – RAM memorija sa sinhronim čitanjem.

U oba slučaja e*ntity* deklaracija RAM memorije ima sledeći izgled

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

entity ram1024x32 is
   port (clk:        in std_logic;

         address: in std_logic_vector(9 downto 0);
         rdata:    out std_logic_vector(31 downto 0);
         wdata:   in std_logic_vector(31 downto 0);
         write:    in std_logic
         );
end entity ram1024x32;
```

Nakon što smo opisali interfejs memorije, potrebno je modelovati njenu funkcionalnost. Za ovo je potrebno napisati odgovarajuće arhitekturno telo.

Bihevioralni model

Varijanta 1: Model RAM memorije sa asinhronim čitanjem

```
architecture beh1 of ram1024x32 is
  type ram_type_t is array (0 to 1023) of std_logic_vector(31 downto 0);
  signal ram_s: ram_type_t;
begin
   -- proces koji modeluje upis u memoriju
   write_ram: process (clk) is
   begin
      if (clk’event and clk = '1') then
         if (write = ’1’) then
            ram_s(to_integer(unsigned(address))) <= wdata;
         end if;
      end if;
   end process;

   -- asinhrono citanje iz memorije
   rdata <= ram_s(to_integer(unsigned(address)));
end architecture beh1;
```

Varijanta 2: Model RAM memorije sa sinhronim čitanjem, „Read-First“ mod

```
architecture beh2 of ram1024x32 is
  type ram_type_t is array (0 to 1023) of std_logic_vector(31 downto 0);
  signal ram_s: ram_type_t;
begin
   ram: process (clk) is
   begin
      if (clk’event and clk = '1') then
         if (write = ’1’) then
```

```
            ram_s(to_integer(unsigned(address))) <= wdata;
         end if;
         rdata <= ram_s(to_integer(unsigned(address)));
      end if;
   end process;
end architecture beh2;
```

NAPOMENA: Ovaj tip RAM memorije prilikom upisa novog podatka u memorijsku lokaciju na izlazni port za podatke (*rdata*) postavlja staru vrednost memorijske lokacije.

Varijanta 3: Model RAM memorije sa sinhronim čitanjem, „Write-First“ mod

```
architecture beh3 of ram1024x32 is
  type ram_type_t is array (0 to 1023) of std_logic_vector(31 downto 0);
  signal ram_s: ram_type_t;
begin
   ram: process (clk) is
   begin
      if (clk’event and clk = '1') then
         if (write = ’1’) then
            ram_s(to_integer(unsigned(address)))  <= wdata;
            rdata <= wdata;
         else
            rdata <= ram_s(to_integer(unsigned(address)));
         end if;
      end if;
   end process;
end architecture beh3;
```

NAPOMENA: Ovaj tip RAM memorije prilikom upisa novog podatka u memorijsku lokaciju na izlazni port za podatke (*rdata*) postavlja novu vrednost memorijske lokacije.

Varijanta 4: Model RAM memorije sa sinhronim čitanjem, „No-Change“ mod

```
architecture beh4 of ram1024x32 is
  type ram_type_t is array (0 to 1023) of std_logic_vector(31 downto 0);
  signal ram_s: ram_type_t;
begin
   ram: process (clk) is
   begin
      if (clk’event and clk = '1') then
         if (write = ’1’) then
            ram_s(to_integer(unsigned(address)))  <= wdata;
         else
            rdata <= ram_s(to_integer(unsigned(address)));
         end if;
```

```
      end if;
    end process;
end architecture beh4;
```

NAPOMENA: Ovaj tip RAM memorije prilikom upisa novog podatka u memorijsku lokaciju ne menja stanje izlaznog porta za podatke (*rdata*).

Kombinovanjem entity deklaracije i odgovarajućeg arhitekturnog tela dobijamo kompletan model RAM memorije.

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

entity ram1024x32 is
   port (clk:        in std_logic;

         address:   in std_logic_vector(9 downto 0);
         rdata:      out std_logic_vector(31 downto 0);
         wdata:     in std_logic_vector(31 downto 0);
         write:      in std_logic
         );
end entity ram1024x32;

architecture beh1 of ram1024x32 is
  type ram_type_t is array (0 to 1023) of std_logic_vector(31 downto 0);
  signal ram_s: ram_type_t;
begin
   -- proces koji modeluje upis u memoriju
   write_ram: process (clk) is
   begin
      if (clk'event and clk = '1') then
         if (write = '1') then
            ram_s(to_integer(unsigned(address))) <= wdata;
         end if;
      end if;
   end process;

   -- asinhrono citanje iz memorije
   rdata <= ram_s(to_integer(unsigned(address)));
end architecture beh1;
```

Verifikaciono okruženje

Primer jednog mogućeg testbenča koji se može iskoristiti za verifikaciju RAM memorije prikazan je u nastavku.

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


entity ram1024x32_tb is
end entity ram1024x32_tb;

architecture beh of ram1024x32_tb is
   -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
   -- generatora sa ulazima DUV-a
   signal clk_s: std_logic;
   signal address_s: std_logic_vector(9 downto 0);
   signal rdata_s: std_logic_vector(31 downto 0);
   signal wdata_s: std_logic_vector(31 downto 0);
   signal write_s: std_logic;

begin
  -- Komponenta koja se verifikuje
  duv: entity work. ram1024x32(beh1)
     port map (
        clk =>       clk_s,
        address => address_s,
        rdata =>     rdata_s,
        wdata =>   wdata_s,
        write =>     write_s);

  -- Klok generator koji generise periodicni clk_s signal koji
  -- ce se koristiti za aktiviranje RAM memorije
  clk_gen: process
  begin
     clk_s <= ‘0’, ‘1’ after 100 ns;
     wait for 200 ns;
  end process;

  -- Stimulus generator koji generise potrebne vrednosti na
  -- ulaznim portovima DUV-a na osnovu kojih ce biti moguce
  -- proveriti da li DUV implementira potrebnu funkcionalnost
  stim_gen: process
  begin
     -- Inicijalizacija
     address_s <= (others => '0');
     wdata_s <= (others => '0');
     write_s <= '0';

     wait for 200 ns;
     -- Upisimo prvo podatke u RAM memoriju
     write1_s <= '1';
     for i in 0 to 1023 loop
        address_s <= std_logic_vector(to_unsigned(i, 10));
```

```
        wdata_s <= std_logic_vector(to_unsigned(2*i+1, 32));
        wait for 200 ns;
    end loop;

    write1_s <= '0';
    wait for 200 ns;

    -- Procitajmo podatke iz RAM memorije
    for i in 0 to 1023 loop
        address_s <= std_logic_vector(to_unsigned (i, 10));
        wait for 200 ns;
    end loop;

    wait;
  end process;
end architecture beh;
```

Prikazani testbenč instancionira komponentu RAM memorije 1024x32 i koristeći *clk_gen* i *stim_gen* procese generiše povorku ulaznih signala koja može da se iskoristi za proveru ispravnog rada napisanog modela RAM memorije.

Clk_gen proces generiše periodičnu povorku impulsa na signalu *clk_s* koji se dovodi na *clk* ulaz RAM memorije.

Stim_gen proces generiše vremenske oblike za adresnu magistralu RAM memorije (*address_s* signal). Takođe, proces generiše i potrebne vremenske oblike za ulaznu magistralu podataka (*wdata_s*), kao i za signal dozvole upisa, *write_s*.

VHDL modeli ROM memorije

Osnovna razlika između RAM i ROM memorije je da se iz ROM memorije mogu samo čitati podaci. Upis podataka nije moguć u ROM memoriju. U tom smislu, interfejs ROM memorije je gotovo identičan sa RAM memorijom sa dve razlike: ne postoji ulazni port za podatke, *wdata*, i ne postoji signal dozvole upisa u memoriju, *write*.

Da bi smo ilustrovali različite načine modelovanja ROM memorije, koristićemo dve verzije 20-bitne ROM memorije sastavljene od 32 lokacija:

- verzija 1 – ROM memorija sa asinhronim čitanjem,
- verzija 2 – ROM memorija sa sinhronim čitanjem.

U prvoj verziji e*ntity* deklaracija ROM memorije ima sledeći izgled

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

entity rom32x20 is
   port (address:   in std_logic_vector(4 downto 0);
```

```
        rdata:      out std_logic_vector(19 downto 0)
        );
end entity rom32x20;
```

U drugoj verziji e*ntity* deklaracija ROM memorije ima sledeći izgled

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

entity rom32x20 is
   port (clk:        in std_logic;
        address:  in std_logic_vector(4 downto 0);
        rdata:      out std_logic_vector(19 downto 0)
        );
end entity rom32x20;
```

Nakon što smo opisali interfejs memorije, potrebno je modelovati njenu funkcionalnost. Za ovo je potrebno napisati odgovarajuće arhitekturno telo.

Bihevioralni model

Varijanta 1: Model ROM memorije sa asinhronim čitanjem

```
architecture beh1 of rom32x20 is
     type rom_type_t is array (0 to 31) of std_logic_vector(19 downto 0);
     signal rom_s: rom_type_t :=(
                    X”00F00”, X”00F01”, X”00F02”, X”00F03”, X”00F04”,
                    X”00F05”, X”00F06”, X”00F07”, X”00F08”, X”00F09”,
                    X”00F0A”, X”00F0B”, X”00F0C”, X”00F0D”, X”00F0E”,
                    X”00F0F”, X”00F10”, X”00F11”, X”00F12”, X”00F13”,
                    X”00F14”, X”00F15”, X”00F16”, X”00F17”, X”00F18”,
                    X”00F19”, X”00F1A”, X”00F1B”, X”00F1C”, X”00F1D”,
                    X”0001E”, X”0001F”);
begin
     -- asinhrono citanje iz ROM memorije
     rdata <= rom_s(to_integer(unsigned(address)));
end architecture beh1;
```

Varijanta 2: Model ROM memorije sa sinhronim čitanjem

```
architecture beh2 of rom32x20 is
     type rom_type_t is array (0 to 31) of std_logic_vector(19 downto 0);
```

```
    signal rom_s: rom_type_t :=(
                X"00F00", X"00F01", X"00F02", X"00F03", X"00F04",
                X"00F05", X"00F06", X"00F07", X"00F08", X"00F09",
                X"00F0A", X"00F0B", X"00F0C", X"00F0D", X"00F0E",
                X"00F0F", X"00F10", X"00F11", X"00F12", X"00F13",
                X"00F14", X"00F15", X"00F16", X"00F17", X"00F18",
                X"00F19", X"00F1A", X"00F1B", X"00F1C", X"00F1D",
                X"0001E", X"0001F");
begin
  -- sinhrono citanje iz ROM memorije
  rom: process (clk) is
  begin
    if (clk'event and clk = '1') then
      rdata <= rom_s(to_integer(unsigned(address)));
    end if;
  end process;
end architecture beh2;
```

Kombinovanjem entity deklaracije i odgovarajućeg arhitekturnog tela dobijamo kompletan model RAM memorije. Na primer, u nastavku je prikazan kompletni model ROM memorije 32x20 sa sinhronim čitanjem podataka.

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

entity rom32x20 is
  port (clk:      in std_logic;
       address:  in std_logic_vector(4 downto 0);
       rdata:    out std_logic_vector(19 downto 0)
       );
end entity rom32x20;


architecture beh2 of rom32x20 is
    type rom_type_t is array (0 to 31) of std_logic_vector(19 downto 0);
    signal rom_s: rom_type_t :=(
                X"00F00", X"00F01", X"00F02", X"00F03", X"00F04",
                X"00F05", X"00F06", X"00F07", X"00F08", X"00F09",
                X"00F0A", X"00F0B", X"00F0C", X"00F0D", X"00F0E",
                X"00F0F", X"00F10", X"00F11", X"00F12", X"00F13",
                X"00F14", X"00F15", X"00F16", X"00F17", X"00F18",
                X"00F19", X"00F1A", X"00F1B", X"00F1C", X"00F1D",
                X"0001E", X"0001F");
begin
  -- sinhrono citanje iz ROM memorije
  rom: process (clk) is
  begin
    if (clk'event and clk = '1') then
```

```
            rdata <= rom_s(to_integer(unsigned(address)));
        end if;
    end process;
end architecture beh2;
```

Verifikaciono okruženje

Primer jednog mogućeg testbenča koji se može iskoristiti za verifikaciju ROM memorije prikazan je u nastavku.

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

entity rom32x20 _tb is
end entity rom32x20 _tb;


architecture beh of rom32x20 _tb is
    -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
    -- generatora sa ulazima DUV-a
    signal clk_s: std_logic;
    signal address_s: std_logic_vector(4 downto 0);
    signal rdata_s: std_logic_vector(19 downto 0);

begin
   -- Komponenta koja se verifikuje
   duv: entity work. rom32x20(beh2)
       port map (
        clk =>        clk_s,
        address => address_s,
        rdata =>     rdata_s);

   -- Klok generator koji generise periodicni clk_s signal koji
   -- ce se koristiti za aktiviranje ROM memorije
   clk_gen: process
   begin
       clk_s <= ‘0’, ‘1’ after 100 ns;
       wait for 200 ns;
   end process;

   -- Stimulus generator koji generise potrebne vrednosti na
   -- ulaznim portovima DUV-a na osnovu kojih ce biti moguce
   -- proveriti da li DUV implementira potrebnu funkcionalnost
   stim_gen: process
   begin
       -- Inicijalizacija
       address_s <= (others => '0');

       -- Procitajmo podatke iz ROM memorije
```

```
        for i in 0 to 31 loop
            address_s <= std_logic_vector(to_unsigned(i, 5));
            wait for 200 ns;
        end loop;

        wait;
    end process;
end architecture beh;
```

Prikazani testbenč instancionira komponentu ROM memorije 32x20 i koristeći *clk_gen* i *stim_gen* procese generiše povorku ulaznih signala koja može da se iskoristi za proveru ispravnog rada napisanog modela ROM memorije.

Clk_gen proces generiše periodičnu povorku impulsa na signalu *clk_s* koji se dovodi na *clk* ulaz ROM memorije.

Stim_gen proces generiše vremenske oblike za adresnu magistralu ROM memorije (*address_s* signal).

Zadaci za vežbu

Zadatak 1:

Napisati VHDL model memorije sa sledećim karakteristikama:

- kapacitet memorije, $m = 64$
- širine magistrala podataka, $n = 16$
- tip čitanja podataka iz registarske banke – sinhrono
- mod čitanja – „Write-First“
- sinhroni *reset* ulaz
- sinhroni *clock enable* ulaz
- sinhroni *enable* ulaz

Za razvijeni VHDL model napisati odgovarajući testbenč pomoću kojega će biti moguće izvršiti funkcionalnu verifikaciju modela.

Zadatak 2:

Napisati VHDL model dvopristupne memorije sa sledećim karakteristikama:

- kapacitet memorije, $m = 128$
- širine magistrala podataka, $n = 32$
- broj pristupa za upis, $w = 1$
- broj pristupa za čitanje, $r = 1$
- tip čitanja podataka iz registarske banke – sinhrono
- mod čitanja – „Read-First“
- sinhroni *reset* ulaz
- sinhroni *clock enable* ulaz
- sinhroni *enable* ulaz

Za razvijeni VHDL model napisati odgovarajući testbenč pomoću kojega će biti moguće izvršiti funkcionalnu verifikaciju modela.

Zadatak 3:

Preraditi VHDL model iz zadatka 2 tako da umesto zajedničkog kloka *clk*, svaki pristup ima svoj sinhronizacioni signal (*rclk* za pristup za čitanje, *wclk* za pristup za upis). Za razvijeni VHDL model napisati odgovarajući testbenč pomoću kojega će biti moguće izvršiti funkcionalnu verifikaciju modela.

Zadatak 4:

Napisati VHDL model konvertora BCD koda u 7-segmentni kod. Model treba da se bazira na korišćenju ROM memorije u kojoj je smeštena konverziona tabela između BCD i 7-segmentnog koda. Za razvijeni VHDL model napisati odgovarajući testbenč pomoću kojega će biti moguće izvršiti funkcionalnu verifikaciju modela.

Brojači

Brojači su standardne sekvencijalne mreže koje se sastoje od jednog *n*-bitnog registra (koji se obično naziva brojački registar) i odgovarajućih aritmetičkih kombinacionih mreža koje služe za izračunavanje nove vrednosti brojačkog registra. Iako postoji veliki broj različitih brojača, osnovni tip predstavljaju brojači koji u svakoj periodi klok signala uvećavaju ili umanjuju trenutni sadržaj brojačkog registra za jedan.

Brojači su vrlo korisne komponente koje se često koriste unutar složenih digitalnih sistema. Brojači se, na primer, koriste za brojanje dešavanja odgovarajućih događaja, merenje periodičnih vremenskih intervala koji se mogu iskoristiti za kontrolu različitih zadataka u sistemu, merenju proteklog vremena između specifičnih događaja, itd.

U zavisnosti od toga kojim signalom se okidaju, brojači se mogu podeliti u dve grupe:

- asinhrone (redne) brojače
- sinhrone (paralelne) brojače

Kod asinhronih brojača okidanje flip flopova izvodi se redno, obično se izlaz prethodnog flip flopa vodi na *clk* ulaz narednog. Zbog ovakvog načina povezivanja asinhroni brojači imaju veće propagaciono kašnjenje jer da bi se stabilizovao izlaz tekućeg flip flopa neophodno je da se prvo stabilizuje stanje prethodnog flip flopa u nizu. Što je asinhroni brojač veći, to je ovo propagaciono kašnjenje duže. Zbog ove osobine, asinhroni brojači se ne koriste često unutar složenih digitalnih sistema.

Za razliku od asinhronih brojača, flip flopovi unutar sinhronih brojača okidaju se jednim, zajedničkim klok signalom. Zbog ovakvog načina povezivanja stanja svih flip flopova unutar brojačkog registra se menjaju istovremeno, tako da veličina brojača nema uticaja na njegovu brzinu. Većina brojača koji se koriste unutar složenih digitalnih sistema pripadaju ovoj grupi.

U zavisnosti od smera brojanja svi brojači se mogu podeliti u tri grupe:

- brojači na gore,
- brojači na dole
- obostrani brojači

U zavisnosti opsega brojanja, brojači se dele u dve grupe:

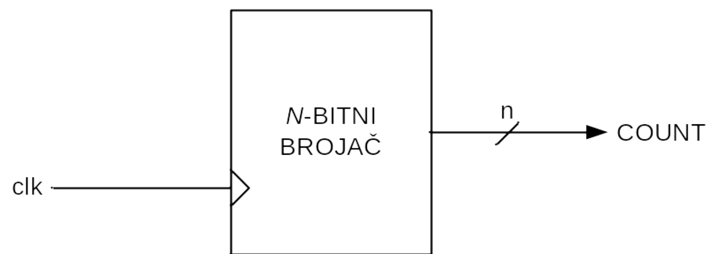
- *n*-bitne brojače
- modulo brojače

Kod *n*-bitnih brojača opseg brojanja, *m*, određen je brojem bita brojačkog registra, *n*, i uvek iznosi $M = 2^n$.

Kod modulo brojača opseg brojanja može biti proizvoljan, sve dotle dok ne izlazi iz maksimalnog opsega koji je dati brojač u stanju da izbroji, $M \leq 2^n$. Na osnovu ove definicije, vidimo da su *n*-bitni brojači zapravo modulo brojači kod kojih je moduo maksimalan mogući za dati brojač, $M = 2^n$.

N-bitni brojači

Osnovni interfejs *n*-bitnog brojača prikazan je na slici 4.



Slika 4. Osnovni interfejs n-bitnog brojača

N-bitni brojač u najosnovnijoj konfiguraciji poseduje samo dva porta:

- *clk* ulaz za sinhronizaciju rada brojača
- izlazni port, *count*, preko koga se dobija informacija o tekućem stanju brojačkog registra

Pored navedenih portova brojači mogu posedovati i čitav niz dodatnih portova:

- ulazni port dozvole rada brojača - *en* (*enable*)
- *clock enable* (*ce*) signal za selekciju rastućih, odnosno opadajućih, ivica *clk* ulaza na koje brojač treba da reaguje
- ulazni port dozvole upisa novog sadržaja u brojački registar - *load*
- ulazni port podataka za prosleđivanje nove vrednosti koja se želi upisati u brojački registar
- indikator smera brojanja, u slučaju obostranog brojača - *updown*
- izlazni port indikacije prekoračenja opsega - *overflow*
- ulazne portove za inicijalizaciju sadržaja brojačkog registra - *reset*, *clear*

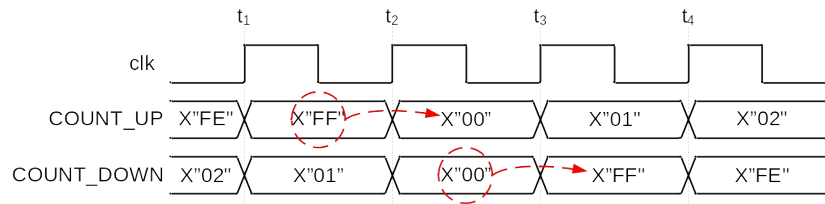
Većina navedenih dodatnih portova su sinhroni (osim *clear* porta koji je asinhroni).

Funkcionalnost *n*-bitnog brojača na gore i brojača na dole može se prikazati sledećom tabelom.

	Brojač na gore	Brojač na dole
clk	*Q*(t+1)	*Q*(t+1)
0	*Q*(t)	*Q*(t)
1	*Q*(t)	*Q*(t)
↓	*Q*(t)	*Q*(t)
↑	*Q*(t)+1	*Q*(t)-1

Kao što se može primetiti na osnovu prethodne tabele, sve dok se na *clk* ulazu ne pojavi rastuća ivica, brojač čuva prethodnu vrednost brojačkog registra. Tek kada se na *clk* ulazu pojavi "sinhronizacioni događaj" (rastuća ili opadajuća ivica), brojač inkrementuje (uvećava za jedan) ili dekrementuje (umanjuje za jedan) trenutnu vrednost brojačkog registra, u zavisnosti od tipa brojača. U slučaju da prilikom inkrementovanja ili dekrementovanja dođe

do prekoračenja opsega on se ignoriše, a brojanje se nastavlja po modulu $m = 2^n$. Na primer, vremenski dijagram rada 8-bitnog brojača na gore i 8-bitnog brojača na dole prikazan je na slici 5. Na slici su prikazane i karakteristične situacije kada dolazi do prekoračenja opsega kako bi se ilustrovao način rada brojača u tim situacijama.



Slika 5. Vremenski dijagram rada 8-bitnog brojača na gore i 8-bitnog brojača na dole

VHDL modeli *n*-bitnog brojača

Da bi smo ilustrovali različite načine modelovanja n-bitnog brojača koristićemo 4-bitni brojač. *Entity* deklaracija 4-bitnog brojača ima sledeći izgled

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

entity counter4bit is
   port (clk: in std_logic;
         q:   out std_logic_vector(3 downto 0) – stanje brojača
         );
end entity counter4bit;
```

Nakon što smo opisali interfejs 4-bitnog brojača, potrebno je modelovati njegovu funkcionalnost. Za ovo je potrebno napisati odgovarajuće arhitekturno telo.

Bihevioralni model

Varijanta 1: Model 4-bitnog brojača na gore

```
architecture up_counter of counter4bit is
   signal count_s: std_logic_vector(3 downto 0) := (others => ‘0’);
begin
  cnt: process (clk) is
  begin
     if (clk’event and clk = '1') then
        count_s <= std_logic_vector(unsigned(count_s) + 1);
     end if;
  end process;

  q <= count_s;
end architecture up_counter;
```

Varijanta 2: Model 4-bitnog brojača na dole

```
architecture down_counter of counter4bit is
   signal count_s: std_logic_vector(3 downto 0) := (others => ‘0’);
begin
  cnt: process (clk) is
  begin
     if (clk’event and clk = '1') then
        count_s <= std_logic_vector(unsigned(count_s) - 1);
     end if;
  end process;

  q <= count_s;
end architecture down_counter;
```

Kombinovanjem entity deklaracije i odgovarajućeg arhitekturnog tela dobijamo kompletan model 4-bitnog brojača. U nastavku je prikazan kompletan model 4-bitnog brojača na gore.

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

entity counter4bit is
   port (clk: in std_logic;
         q:   out std_logic_vector(3 downto 0) – stanje brojača
         );
end entity counter4bit;

architecture up_counter of counter4bit is
   signal count_s: std_logic_vector(3 downto 0) := (others => ‘0’);
begin
  cnt: process (clk) is
  begin
     if (clk’event and clk = '1') then
        count_s <= std_logic_vector(unsigned(count_s) + 1);
     end if;
  end process;

  q <= count_s;
end architecture up_counter;
```

Verifikaciono okruženje

Primer jednog mogućeg testbenča koji se može iskoristiti za verifikaciju 4-bitnog brojača na gore i 4-bitnog brojača na dole, prikazan je u nastavku.

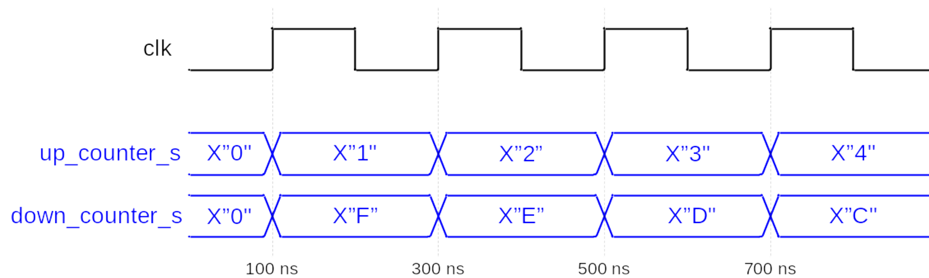
```
library ieee;
use ieee.std_logic_1164.all;

entity counter4bit_tb is
end entity counter4bit_tb;


architecture beh of counter4bit_tb is
   -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
   -- generatora sa ulazima DUV-a
   signal clk_s: std_logic;
   signal up_counter_s: std_logic_vector(3 downto 0);
   signal down_counter_s: std_logic_vector(3 downto 0);

begin
  -- Komponente koja se verifikuju
  up_counter: entity work. counter4bit (up_counter)
     port map (
        clk => clk_s,
        q =>   up_counter_s);

  down_counter: entity work. counter4bit (down_counter)
     port map (
        clk => clk_s,
        q =>   down_counter_s);

  -- Klok generator koji generise periodicni clk_s signal koji
  -- ce se koristiti za aktiviranje brojača
  clk_gen: process
  begin
           clk_s <= ‘0’, ‘1’ after 100 ns;
           wait for 200 ns;
  end process;

  -- Obzirom da brojači nema niti jedan drugi ulazni port
   -- osim clk ulaza, nemamo stimulu generator proces
end architecture beh;
```

Prikazani testbenč instancionira dve komponente 4-bitnog brojača, ali prilikom svakog instancioniranja instancioniranom entitetu pridružuje drugu arhitekturu. Na ovaj način je moguće instancionirati i 4-bitni brojač na gore i 4-bitni brojač na dole u istom testbenču.

Clk_gen proces generiše periodičnu povorku impulsa na signalu *clk_s* koji se dovodi na *clk* ulaze oba brojača.

Obzirom da razvijeni 4-bitni brojači nemaju nikakav dodatni ulazni port osim *clk* porta, nema potrebe za pisanjem stimulus generatora, tako da ovaj testbenč ne sadrži *stim_gen* proces.

Nakon što se izvrši simulacija kombinovanog rada testbenča i modela 4-bitnih brojača na gore i na dole, talasni oblici izlaznih signala stanja brojačkih registara, *up_counter_s* i *down_counter_s*, trebalo bi da izgledaju kao na slici 6.



Slika 6. Generisani talasni oblici na izlazima 4-bitnog brojača na gore i 4-bitnog brojača na dole, koji su dobijeni kao rezultat simulacije

Zadaci za vežbu

Zadatak 1:

Napisati VHDL model 8-bitnog brojača na gore sa sledećim dodatnim ulaznim portovima:

- signal dozvole brojanja – *en*
- *clock enable* ulaz - *ce*
- sinhroni *reset* ulaz

Za tako napisani model brojača razviti i potrebno verifikaciono okruženje koje će moći da se iskoristi za njegovu funkcionalnu verifikaciju.

Zadatak 2:

Napisati VHDL model 8-bitnog brojača na dole sa sledećim dodatnim ulaznim portovima:

- signal dozvole brojanja – *en*
- *clock enable* ulaz - *ce*
- ulazni port dozvole upisa novog sadržaja u brojački registar - *load*
- ulazni port podataka za prosleđivanje nove vrednosti koje se želi upisati u brojački registar – *counter_value*

Za tako napisani model brojača razviti i potrebno verifikaciono okruženje koje će moći da se iskoristi za njegovu funkcionalnu verifikaciju.

Zadatak 3:

Napisati VHDL model 16-bitnog brojača na gore sa sledećim dodatnim ulaznim i izlazni portovima:

- signal dozvole brojanja – *en*
- ulazni port dozvole upisa novog sadržaja u brojački registar - *load*
- ulazni port podataka za prosleđivanje nove vrednosti koje se želi upisati u brojački registar – *counter_value*
- indikator prekoračenja opsega - *overflow*

Za tako napisani model brojača razviti i potrebno verifikaciono okruženje koje će moći da se iskoristi za njegovu funkcionalnu verifikaciju.

Zadatak 4:

Napisati VHDL model 10-bitnog obostranog brojača sa sledećim dodatnim ulaznim i izlazni portovima:

- signal dozvole brojanja – *en*
- *clock enable* ulaz - *ce*
- indikator smera brojanja, u slučaju obostranog brojača - *updown*
- ulazni port dozvole upisa novog sadržaja u brojački registar - *load*
- ulazni port podataka za prosleđivanje nove vrednosti koje se želi upisati u brojački registar – *counter_value*
- indikator prekoračenja opsega - *overflow*

Za tako napisani model brojača razviti i potrebno verifikaciono okruženje koje će moći da se iskoristi za njegovu funkcionalnu verifikaciju.

Modulo brojači

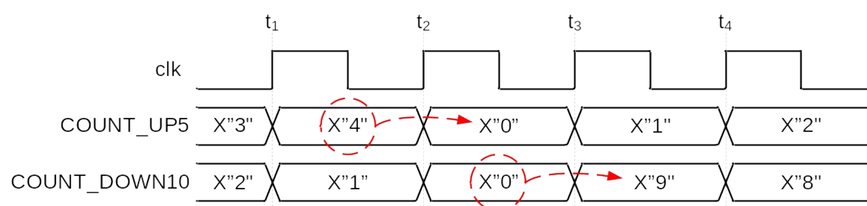
Modulo brojači imaju isti interfejs kao i *n*-bitni brojači, prikazan na slici 4. Osnovna razlika između modulo brojača i *n*-bitnih brojača jeste u opsegu brojanja. Modulo brojači broje po modulu *M*, koji uvek mora da zadovolji sledeću relaciju, $M \leq 2^n$, gde je *n* broj bita unutar brojačkog registra. Dok kod binarnih brojača moduo uvek mora biti stepen broja 2, kod modulo brojača to ne mora biti slučaj. Moduo brojanja modulo brojača može biti proizvoljan broj iz intervala $[2\ 2^n]$.

Prilikom projektovanja modulo brojača za zadati modulo brojanja *M*, od interesa je odabrati minimalno potrebno širinu brojačkog registra, *n*. Minimalna širina brojačkog registra modulo brojača modula *M* iznosi, $n = \lceil \log_2 M \rceil$.

Funkcionalnost modulo brojača modula *M* na gore i modulo brojača modula *M* na dole prikazana je sledećom tabelom.

	Brojač na gore	Brojač na dole
Clk	*Q*(t+1)	*Q*(t+1)
0	*Q*(t)	*Q*(t)
1	*Q*(t)	*Q*(t)
↓	*Q*(t)	*Q*(t)
↑	*ako je* *Q*(t)<*M*-1, *Q*(t)+1 *ako je* *Q*(t)=*M*-1, 0	*ako je* *Q*(t)>0, *Q*(t)-1 *ako je* *Q*(t)=0, *M*

Na primer, vremenski dijagram rada modulo brojača modula 5 na gore i modulo brojača modula 10 na dole prikazan je na slici 7. Na slici su prikazane i karakteristične situacije kada dolazi do prekoračenja opsega kako bi se ilustrovao način rada brojača u tim situacijama.



Slika 7. Vremenski dijagram rada modulo brojača modula 5 na gore i modulo brojača modula 10 na dole

VHDL modeli modulo brojača

Da bi smo ilustrovali različite načine modelovanja modulo brojača, koristićemo modulo brojač modula 155. *Entity* deklaracija modulo brojača ima sledeći izgled

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

entity counter155 is
   port (clk: in std_logic;
```

```
        q:   out std_logic_vector(7 downto 0) -- stanje brojača
        );
end entity counter155;
```

Nakon što smo opisali interfejs modulo brojača, potrebno je modelovati njegovu funkcionalnost. Za ovo je potrebno napisati odgovarajuće arhitekturno telo.

Bihevioralni model

Varijanta 1: Model modulo brojača modula 155 na gore

```
architecture up_counter of counter155 is
   signal count_s: std_logic_vector(7 downto 0) := (others => ‘0’);
begin
  cnt: process (clk) is
  begin
        if (clk’event and clk = '1') then
           if (count_s < std_logic_vector(to_unsigned(154, 8))) then
              count_s <= std_logic_vector(unsigned(count_s) + 1);
           else
              count_s <= (others => ’0’);
           end if;
        end if;
  end process;

  q <= count_s;
end architecture up_counter;
```

Varijanta 2: Model modulo brojača modula 155 na dole

```
architecture down_counter of counter155 is
   signal count_s: std_logic_vector(3 downto 0) := (others => ‘0’);
begin
  cnt: process (clk) is
  begin
        if (clk’event and clk = '1') then
           if (count_s > std_logic_vector(to_unsigned(0, 8))) then
              count_s <= std_logic_vector(unsigned(count_s) - 1);
           else
              count_s <= std_logic_vector(to_unsigned(154, 8));
           end if;
        end if;
  end process;

  q <= count_s;
end architecture down_counter;
```

Kombinovanjem *entity* deklaracije i odgovarajućeg arhitekturnog tela, dobijamo kompletan model modulo brojača. U nastavku je prikazan kompletan model modulo brojača na gore, modula 155.

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

entity counter155 is
   port (clk: in std_logic;
         q:   out std_logic_vector(7 downto 0) -- stanje brojača
         );
end entity counter155;


architecture down_counter of counter155 is
   signal count_s: std_logic_vector(3 downto 0) := (others => ‘0’);
begin
  cnt: process (clk) is
  begin
         if (clk’event and clk = '1') then
            if (count_s > std_logic_vector(to_unsigned(0, 8))) then
               count_s <= std_logic_vector(unsigned(count_s) - 1);
            else
               count_s <= std_logic_vector(to_unsigned(154, 8));
            end if;
         end if;
  end process;

  q <= count_s;
end architecture down_counter;
```

Verifikaciono okruženje

Primer jednog mogućeg testbenča koji se može iskoristiti za verifikaciju modulo brojača modula 155 na gore i modulo brojača modula 155 na dole prikazan je u nastavku.

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

entity counter155_tb is
end entity counter155_tb;


architecture beh of counter155_tb is
   -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
   -- generatora sa ulazima DUV-a
   signal clk_s: std_logic;
   signal up_counter_s: std_logic_vector(7 downto 0);
   signal down_counter_s: std_logic_vector(7 downto 0);

begin
```

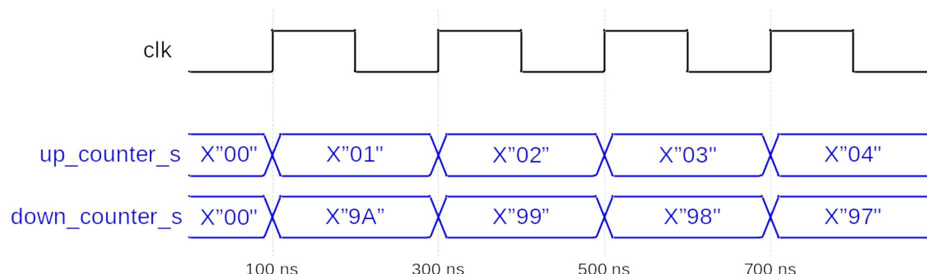
```
-- Komponente koja se verifikuju
up_counter: entity work.counter155 (up_counter)
   port map (
      clk => clk_s,
      q =>    up_counter_s);

down_counter: entity work.counter155 (down_counter)
   port map (
      clk => clk_s,
      q =>    down_counter_s);

-- Klok generator koji generise periodicni clk_s signal koji
-- ce se koristiti za aktiviranje brojača
clk_gen: process
begin
         clk_s <= '0', '1' after 100 ns;
         wait for 200 ns;
end process;

-- Obzirom da brojači nema niti jedan drugi ulazni port
-- osim clk ulaza, nemamo stimulu generator proces
end architecture beh;
```

Prikazani testbenč instancionira dve komponente modulo brojača, ali prilikom svakog instancioniranja instancioniranom entitetu pridružuje drugu arhitekturu. Na ovaj način je moguće instancionirati i modulo brojač na gore i modulo brojač na dole u istom testbenču.

Clk_gen proces generiše periodičnu povorku impulsa na signalu *clk_s* koji se dovodi na *clk* ulaze oba brojača.

Obzirom da razvijeni modulo brojači nemaju nikakav dodatni ulazni port osim *clk* porta, nema potrebe za pisanjem stimulus generatora, tako da ovaj testbenč ne sadrži *stim_gen* proces.

Nakon što se izvrši simulacija kombinovanog rada testbenča i modela modulo brojača na gore i na dole, talasni oblici izlaznih signala stanja brojačkih registara, *up_counter_s* i *down_counter_s*, trebalo bi da izgledaju kao na slici 8.



Slika 8. Generisani talasni oblici na izlazima modulo brojača modula 155 na gore i modulo brojača modula 155 na dole, koji su dobijeni kao rezultat simulacije

Zadaci za vežbu

Zadatak 1:

Napisati VHDL model modulo brojača modula 100 na gore sa sledećim dodatnim ulaznim portovima:

- signal dozvole brojanja – *en*
- *clock enable* ulaz - *ce*
- sinhroni *reset* ulaz

Za tako napisani model brojača razviti i potrebno verifikaciono okruženje koje će moći da se iskoristi za njegovu funkcionalnu verifikaciju.

Zadatak 2:

Napisati VHDL model modulo brojača modula 100 na dole sa sledećim dodatnim ulaznim portovima:

- signal dozvole brojanja – *en*
- *clock enable* ulaz - *ce*
- ulazni port dozvole upisa novog sadržaja u brojački registar - *load*
- ulazni port podataka za prosleđivanje nove vrednosti koja se želi upisati u brojački registar – *counter_value*

Za tako napisani model brojača razviti i potrebno verifikaciono okruženje koje će moći da se iskoristi za njegovu funkcionalnu verifikaciju.

Zadatak 3:

Napisati VHDL model modulo brojača modula 100 na gore sa sledećim dodatnim ulaznim i izlazni portovima:

- signal dozvole brojanja – *en*
- ulazni port dozvole upisa novog sadržaja u brojački registar - *load*
- ulazni port podataka za prosleđivanje nove vrednosti koja se želi upisati u brojački registar – *counter_value*
- indikator prekoračenja opsega - *overflow*

Za tako napisani model brojača razviti i potrebno verifikaciono okruženje koje će moći da se iskoristi za njegovu funkcionalnu verifikaciju.

Zadatak 4:

Napisati VHDL model modulo brojača na gore sa opsegom brojanja od 13 do 89, sa sledećim dodatnim ulaznim i izlazni portovima:

- signal dozvole brojanja – *en*
- ulazni port dozvole upisa novog sadržaja u brojački registar - *load*
- indikator prekoračenja opsega – *overflow*

- sinhroni *reset* ulaz

Za tako napisani model brojača razviti i potrebno verifikaciono okruženje koje će moći da se iskoristi za njegovu funkcionalnu verifikaciju.