Laboratorijska vežba 8

Parametrizovani i hijerarhijski dizajn

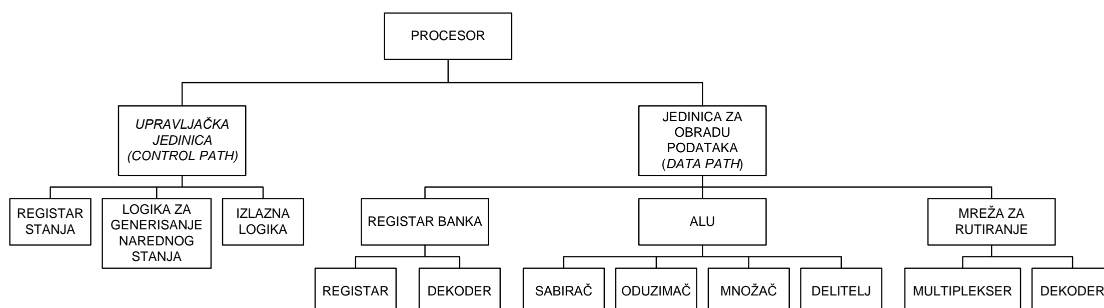
Složenost savremenih digitalnih sistema je tolika da klasičan, centralizovan, pristup njihovom projektovanju rezultuje u neprihvatljivo dugačkom vremenu razvoja. Kako bi se povećala produktivnost i skratilo vreme razvoja novog digitalnog sistema, dizajneri se danas oslanjaju na dve tehnike:

- hijerarhijski, modularni dizajn
- parametrizovani dizajn

Obe ove tehnike ommogućavaju značajno povećanje produktivnosti i skraćivanje vremena potrebnog za projektovanje novog digitalnog sistema oslanjajući se na podelu i paralelizaciju posla, kao i na ponovno korišćenje ranije razvijenih VHDL modela.

Pod hijerarhijskim (modularnim) dizajnom podrazumevamo hijerarhijsku dekompoziciju složenog sistema na skup međusobno povezanih modula. Svaki od ovih modula ima svoj jasno definisani interfejs, način kako se povezuje i komunicira sa svojim okruženjem, kao i funkciju koju treba da ostvaruje. Vrlo često svaki od ovih modula ima toliko složenu strukturu da se proces modularizacije može ponovo primeniti. Rekurzivnom primenom modularizacije dobijamo hijerarhijsku predstavu strukture sistema koji projektujemo. Ovo je pristup koji je generalan, u smislu da se ne primenjuje samo prilikom projektovanja složenih digitalih sistema, već se primenjuje gotovo u svakom inženjerskom poduhvatu (prilikom razvoja softvera, fabrikacije mašina, izgradnje infrastrukturnih objekata, itd.). Hijerarhijska dekompozicija je trenutno najbolji odgovor na problem efikasnog projektovanja složenih sistema.

U slučaju projektovanja složenih digitlanih sistema proces modularizacije obično se zaustavlja kada dođemo do modula čija se funkcionalnost može ostvariti korišćenjem neke od standradnih kombinacionih i sekvencijalnih mreža koje smo razmatrali u prethodnim vežbama. Na kraju modularizacije sistem je predstavljen hijerarhijskom strukturom modula koja se može predstaviti strukturom obrnutog stabla, prikazanog na slici 1. Na slici 1 prikazana je jedna moguća hijerarhijska dekompozicija mikroprocesora



Slika 1. Hijerarhijska dekompozicija hipotetičkog procesora

Na vrhu stabla nalazi se jedan čvor, koren, koji predstavlja čitav sistem koji se projektuje. U našem slučaju to bi bio ceo procesor koji projektujemo. Sledeći nivo čvorova predstavlja module čijim međusobnim povezivanjem nastaje čitav sistem koji želimo da isprojektujemo. U našem slučaju sledeći nivo u hijerahijskog strukturi čine dva modula, upravljačka jedinica (*control path*) i jedinica za obradu podataka (*data path*). Ukoliko je neki od ovih modula složen (ne predstavlja standardnu kombinacionu ili sekvencijalnu mrežu), on se dalje dekomponuje na podmodule, kao što je prikazano na slici 1. Na primer, jedinicu za obradu podataka možemo dalje dekomponovati na:

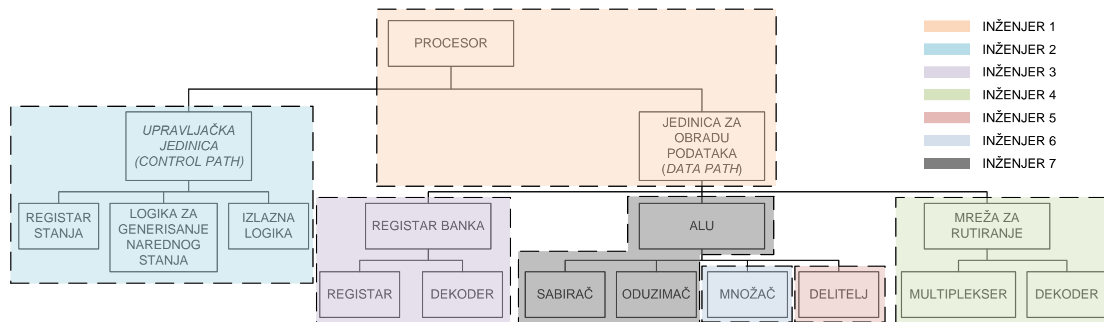
- registarsku banku – u kojoj će se nalaziti svi unutrašnji registri procesora
- aritmetičko logičku jedinicu (ALU) – koja je zadužena za izvođenje aritmatičkih i logičkih operacija na podacima
- mrežu za rutiranje – koja je zadužena za prenos podataka od registarske banke do ALU i obrnuto, kao i za prenos podataka od i ka procesoru

Kao što je ranije napomenuto, proces modularizacije individualnih modula završava se kada dođemo do nivoa standarnih kombinacionih i sekvencijalnih mreža, što se takođe jasno vidi na slici 1. Na primer, proces dekompozicije ALU završava se kada stignemo do nivoa standardnih kombinacionih mreža za realizaciju osnovnih aritmetičkih i logičkih operacija.

Na koji način modularizacija povećava produktivnost i skraćuje vreme potrebno za razvoj novog digitalnog sistema?

Prvi način ogleda se u mogućnosti paralelizacije procesa projektovanja u trenutku kada smo sistem modularizovali i predstavili pomoću hijerarhijske strukture. Na kraju procesa modularizacije svaki od modula ima jasno definisan interfejs i jasno definisanu funkcionalnost koju treba da enkapsulira. Dizajn svakog od ovih modula može se sada poveriti jednom inžinjeru ili inženjer timu, u zavisnosti od složenosti dizajna. Proces projektovanja modula može se izvoditi u paraleli, uključujući i pisanje modela modula na različitim nivoima hijerahije. Ovo je moguće zbog toga što je svaki modul dobro definisan i sve informacije koje su neophodne za njegov razvoj su na raspolaganju. Za svaki modul jasno je kako treba da izgleda njegov interfejs, šta treba da bude njegova funkcionalnost i kakva je njegova unutrašnja struktura (od kojih podmodula se sastoji i kako su oni međusobno povezani). Ovo su sve informacije koje su potrebne dizajneru kako bi pristupio procesu modelovanja. Na slici 2 prikazana je jedna moguća paralelizacija procesa projektovanja procesora sa hijerarhijskom strukturom sa slike 1.

Na slici 2 prikazana je paralelizacija procesa projektovanja procesora u sedam paralelnih procesa. Za svaki od paralelnih procesa zadužen je drugi inženjer. Ukoliko pretpostavimo da je vreme potrebno za kompletiranje svakog od sedam paralelnih procesa izbalansirano (podjednako), onda je paralelizacijom procesa projektovanja procesora ostvareno ubrzanje od sedam puta. Naravno, u praksi je prilično teško ostvariti ovakvu idealnu paralelizaciju posla, tako da je kranji efekat paralelizacije obično manji. Na primer, u našem primeru nerealno je da proces projektovanja mreže za rutiranje zahteva istu količinu posla kao proces projektovanja upravljačke jedinice.



Slika 2. Paralelizacija procesa projektovanja procesora

Drugi način povećanja produktivnosti korišćenjem modularizacije ogleda se u mogućnosti ponovnog korišćenja ranije razvijenih modula (*design reuse*) u novom dizajnu. Umesto da trošimo vreme na razvoj nekog modula, možemo jednostavno iskoristiti postojeći uz nikakve ili minimalne modifikacije. Na ovaj način se opet štedi vreme i smanjuju troškovi razvoja nogov uređaja. Na primer, prilikom projektovanja našeg hipotetičkog procesora, možda bi bilo moguće iskoristiti ranije razvijene module množača i delitelja. Slično, modeli multipleksera i dekodera bi takođe mogli lako da se iskoriste koristeći ranije razvijene modele. Ukoliko nemamo odgovarajuće module, možda ih je moguće kupiti od kompanija koje se bave razvojem i prodajom standardnih modula (*IP cores*). Ako je reč o složenom modulu, možda je isplativije kupiti ga od neke kompanije nego se upuštati u sopstveni razvoj.

Da bi se povećala mogućnost ponovnog korišćenja nekog modula, on bi trebao da ima mogućnost *parametrizacije*. Pod parametrizovanim modelom nekog modula podrazumevamo model koji se jednostavno može prilagoditi tekućim potrebama, prostim konfigurisanjem skupa parametara koji su pridruženi modelu. Na primer, ukoliko napišemo model multipleksera kod koga se može jednostavno promeniti širina ulaznih i izlaznih portova za podatke, onda ga kasnije možemo koristiti i kao 8-bitni, 16-bitni ili 32-bitni multiplekser. Slična logika može se primeniti i prilikom razvoja većine ostalih standardnih kombinacionih i sekvencijalnih mreža (demultiplekseri, komparatori, registri, brojači, itd.).

Parametrizovani VHDL modeli

U zavisnosti od toga šta se parametrizuje, svi parametrizovani modeli mogu se podeliti u dve velike grupe:

- modeli sa parametrizovanom širinom
- modeli sa parametrizovanim ponašanjem

Kod parametrizacije širine, korišćenjem odgovarajućeg broja parametara moguće je definisati potrebnu širinu ulaznih i izlaznih portova modela. U praksi je vrlo čest slučaj da je jedina modifikacija koju je neophodno izvršiti, da bi se ranije razvijeni model mogao koristiti u nekom drugom dizajnu, promena širine ulaznih ili izlaznih portova modula. Parametrizacija širine omogućena je unutar VHDL jezika postojanjem *generic* deklarativnog dela unutar deklaracije entiteta. Definisanjem odgovarajućeg broja generika, koji se kasnije mogu koristiti kao parametri unutar arhitekturnog tela, pa čak i u nastavku *entity* deklaracije, moguće je izvršiti parametrizaciju širine bilo kog VHDL modela. Ponekad se prilikom uvođenja parametrizacije širine ulaznih ili izlaznih portova javlja potrebna i za strukturnim modifikacijama unutar samog modela. U tom slučaju je neophodno koristiti VHDL *generate* naredbe i razviti regularne iterativne modele.

Drugu grupu parametrizovanih modela čine modeli kod kojih je parametrizovano ponašanje. Jedan tip ponašanja koji se često parametrizuje jeste parametrizacija različitih vremenskih karakteristika modela koji se razvija (propagacionog kašnjenja, vremena uspostavljanja i držanja, vremena stabilizacije novih vrednosti na izlazima nakon nailaska rastuće, ili opadajuće, ivice sinhronizacionog signala, itd.). Ovakva vrsta parametrizovanog ponašanja od velikog je interesa prilikom razvoja simulacionih modela (VITAL IEEE standard razvijen je upravo u ovu svrhu). U ovu svrhu ponovo se koriste *generic* deklaracije unutar kojih se definišu odgovarajući parametri koji se zatim koriste unutar samog modela za parametrizaciju vremenskih karakteristika. Naravno, moguće je i parametrizovati samu funkcionalnost samog modela, uključujući ili isključujući odgovarajuće funkcije koje model obezbeđuje. Na ovaj način se model konfiguriše i obezbeđuje samo onaj deo funkcionalnosti koji je neophodan u tekućem sistemu u kojem će se model koristiti. Ovaj vid parametrizacije ponašanja uključuje korišćenje odgovarajućih *generate* naredbi unutar VHDL modela, pomoću kojih se uključuju ili isključuju odgovarajuće komponente prisutne unutar modela. Većina IP jezgara (modela koje razvijaju specijalizovane kompanije koji su namenjeni masovnom ponovnom korišćenju) odlikuje se ovakvim tipom parametrizacije ponašanja.

VHDL modeli sa parametrizovanom širinom

Kao što je ranije već rečeno kod parametrizacije širine nekog VHDL modela obavezno je uključivanje *generic* dela unutar *entity* deklaracije modula. Ovi generici se zatim koriste unutar ostatka *entity* deklaracije, konkretno u njenom delu gde se deklarišu ulazni i izlazni portovi entiteta, kao i u samom arhitekturnom telu.

Da bi smo ilustrovali proces razvoja VHDL modela sa parametrizovanom širinom ulaznih i izlaznih portova, prikazaćemo razvoj sledećih VHDL modela:

- multipleksera 4-na-1 sa parametrizovanom širinom ulaznih i izlaznih portova za podatke

- registra sa serijskim upisom i čitanjem sa parametrizovanom širinom ulaznih i izlaznih portova podataka

VHDL model multipleksera 4-na-1 sa parametrizovanom širinom

Prvi korak je da se napiše odgovarajuća *entity* deklaracija za parametrizovani model multipleksera. Kao što je ranije rečeno, deklaracija će uključiti i *generic* deo u kome ćemo definisati parametar WIDTH, pomoću kojega će biti moguće kontrolisati širinu ulaznih portova za podatke kao i izlaznog porta za podatke. Obzirom da širine ulaznih portova i izlaznog porta za podatke moraju biti iste, dovoljno je koristiti samo jedan parametar. Imajući ovo u vidu, kompletna e*ntity* deklaracija parametrizovanog multipleksera
4-na-1 ima sledeći izgled

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

entity mux4na1 is
   generic (WIDTH: positive := 8);
   port (x1: in std_logic_vector(WIDTH-1 downto 0); -- ulazni port podataka 1
         x2:  in std_logic_vector(WIDTH-1 downto 0); -- ulazni port podataka 2
         x3:  in std_logic_vector(WIDTH-1 downto 0); -- ulazni port podataka 3
         x4:  in std_logic_vector(WIDTH-1 downto 0); -- ulazni port podataka 4
         sel:  in std_logic_vector(1 downto 0);            -- selekcioni ulaz
         y:   out std_logic_vector(WIDTH-1 downto 0)); -- izlazni port podataka
end entity mux4na1;
```

Analizom *entity* deklaracije parametrizovanog modela multipleksera 4-na-1 možemo videti kako je parametar WIDTH iskorišćen za parametrizaciju širine ulaznih portova i izlaznog porta za podatke. Umesto navođena eksplicitnog broja bitova od kojih se sastoje ulazni i izlazni portovi za podatke, gornja granica je parametrizovana pomoću parametra WIDTH. Prilikom svakog instancioniranja komponente mux4na1, parametru WIDTH će biti dodeljena neka konkretna vrednost (koja mora biti pozitivna, jer je WIDTH parametar deklarisan kao tipa *positive*, što u ovom slučaju ima smisla jer širina ulaznog ili izlaznog porta za podtatke mora biti veća od nule), ili će se pak koristiti podrazumevana vrednost (u našem primeru podrazumevana vrednost WIDTH parametra iznosi 8). Ova konkretna vrednost WIDTH parametra će se zatim iskoristiti za specifikaciju širine ulaznih i izlaznog porta unutar posmatrane instance. Bitno je napoenuti da se prilikom svakog instancioniranja komponente mux4na1 može potpuno nezavisno i proizvoljno specificirati željena vrednost WIDTH parametra.

Bihevijani model

Varijanta 1: Model multiplesera 4-na-1 sa parametrizovanom širinom ulaznih i izlaznog porta za podatke bazirana na korišćenju *case* i *process* naredbe ima sledeći izgled.

```
architecture beh1 of mux4na1 is
begin
	mux: process (x1, x2, x3, x4, sel) is
	begin
		case sel is
			when "00" =>
				y <= x1;
			when "01" =>
				y <= x2;
			when "10" =>
				y <= x3;
			when others =>
				y <= x4;
		end case;
	end process;
end architecture beh1;
```

Kao što se može videti u prikazanom modelu nigde se ne koristi WIDTH parametar. Ovde je to potpuno razumljivo obzirom da je za bihevijalni opis multipleksera potpuno nebitno kolika je širina ulaznih i izlaznog porta podataka.

NAPOMENA: Umesto prikazanog bihevijalnog modela mogao se koristiti bilo koji bihevijalni model rada multipleksera prikazan u vežbi 1, bez ikakvih modifikacija.

Verifikaciono okruženje

Sledeći testbenč ilustruje proces instancioniranja dve instance multipleksera 4-na-1, od kojih prva instanca ima 8-bitne ulazne i izlazni port podatka, a druga 16-bitne ulazne i izlazni port podataka.

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

entity mux4na1_tb is
end entity mux4na1_tb;


architecture beh of mux4na1_tb is
  -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
  -- generatora sa ulazima DUV-a
  signal sel_8bit_s: std_logic_vector(1 downto 0);
  signal x1_8bit_s: std_logic_vector(7 downto 0);
  signal x2_8bit_s: std_logic_vector(7 downto 0);
  signal x3_8bit_s: std_logic_vector(7 downto 0);
  signal x4_8bit_s: std_logic_vector(7 downto 0);
  signal y_8bit_s: std_logic_vector(7 downto 0);

  signal sel_16bit_s: std_logic_vector(1 downto 0);
  signal x1_16bit_s: std_logic_vector(15 downto 0);
  signal x2_16bit_s: std_logic_vector(15 downto 0);
  signal x3_16bit_s: std_logic_vector(15 downto 0);
  signal x4_16bit_s: std_logic_vector(15 downto 0);
  signal y_16bit_s: std_logic_vector(15 downto 0);
```

```
begin
    -- Komponente koja se verifikuju
    -- 8-bitni multiplekser 4-na-1
    duv1_mux8bit: entity work.mux4na1(beh1)
      port map (
        sel     =>  sel_8bit_s,
        x1      =>  x1_8bit_s,
        x2      =>  x2_8bit_s,
        x3      =>  x3_8bit_s,
        x4      =>  x4_8bit_s,
        y       =>  y_8bit_s);

    -- 16-bitni multiplekser 4-na-1
    duv2_mux16bit: entity work.mux4na1(beh1)
      generic map (WIDTH => 16)
      port map (
        sel     =>  sel_16bit_s,
        x1      =>  x1_16bit_s,
        x2      =>  x2_16bit_s,
        x3      =>  x3_16bit_s,
        x4      =>  x4_16bit_s,
        y       =>  y_16bit_s);

    -- Stimulus generator koji generise potrebne vrednosti na
    -- ulaznim portovima 8-bitnog multipleksera 4-na-1 na osnovu
    -- kojih ce biti moguce proveriti da li DUV1 implementira
    -- potrebnu funkcionalnost
    stim_gen1: process
    begin
      x1_8bit_s <= X“00”, X“FF” after 100 ns,
                   X“00” after 200 ns, X“FF” after 600 ns,
                   X“00” after 700 ns, X“FF” after 1000 ns,
                   X“00” after 1100 ns, X“FF” after 1400 ns,
                   X“00” after 1500 ns;

      x2_8bit_s <= X“00”, X“FF” after 200 ns,
                   X“00” after 300 ns, X“FF” after 500 ns,
                   X“00” after 600 ns, X“FF” after 1100 ns,
                   X“00” after 1200 ns, X“FF” after 1500 ns,
                   X“00” after 1700 ns;

      x3_8bit_s <= X“00”, X“FF” after 300 ns,
                   X“00” after 400 ns, X“FF” after 700 ns,
                   X“00” after 800 ns, X“FF” after 900 ns,
                   X“00” after 1000 ns, X“FF” after 1600 ns,
                   X“00” after 1700 ns;

      x4_8bit_s <= X“00”, X“FF” after 400 ns,
                   X“00” after 500 ns, X“FF” after 800 ns,
                   X“00” after 900 ns, X“FF” after 1200 ns,
                   X“00” after 1400 ns;
```

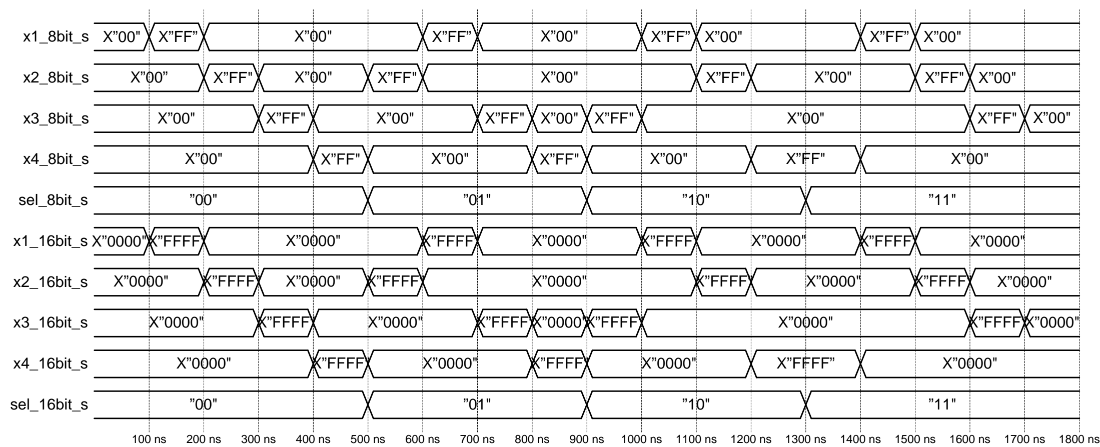
```
        sel_8bit_s <= “00” , “01” after 500 ns, “10” after 900 ns,
                      “11” after 1300 ns;

        wait;
    end process;

    -- Stimulus generator koji generise potrebne vrednosti na
    -- ulaznim portovima 16-bitnog multipleksera 4-na-1 na osnovu
    -- kojih ce biti moguce proveriti da li DUV1 implementira
    -- potrebnu funkcionalnost
    stim_gen2: process
    begin
        x1_16bit_s <= X“0000”, X“FFFF” after 100 ns,
                    X“0000” after 200 ns, X“FFFF” after 600 ns,
                    X“0000” after 700 ns, X“FFFF” after 1000 ns,
                    X“0000” after 1100 ns, X“FFFF” after 1400 ns,
                      X“0000” after 1500 ns;

        x2_16bit_s <= X“0000”, X“FFFF” after 200 ns,
                    X“0000” after 300 ns, X“FFFF” after 500 ns,
                    X“0000” after 600 ns, X“FFFF” after 1100 ns,
                    X“0000” after 1200 ns, X“FFFF” after 1500 ns,
                      X“0000” after 1700 ns;

        x3_16bit_s <= X“0000”, X“FFFF” after 300 ns,
                    X“0000” after 400 ns, X“FFFF” after 700 ns,
                    X“0000” after 800 ns, X“FFFF” after 900 ns,
                    X“0000” after 1000 ns, X“FFFF” after 1600 ns,
                      X“0000” after 1700 ns;

        x4_16bit_s <= X“0000”, X“FFFF” after 400 ns,
                      X“0000” after 500 ns, X“FFFF” after 800 ns,
                      X“0000” after 900 ns, X“FFFF” after 1200 ns,
                      X“0000” after 1400 ns;

        sel_16bit_s <= “00” , “01” after 500 ns, “10” after 900 ns,
                      “11” after 1300 ns;

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

Prikazani testbenč instancionira dve instance komponente *mux4na1*, ali sa različitom vrednošću parametra WIDTH. Prva instanca, *duv1_mux8bit*, ne navodi vrednost za parametar WIDTH, pa će se u ovom slučaju koristiti podrazumevana vrednost, 8. Druga instanca, *duv1_mux16bit*, navodi željenu vrednost parametra WIDTH, 16, pa će kod ove instance širina ulaznih i izlaznog porta biti 16 bita.

Testenč koristi dva stimulus generatora. Prvi timulus generator, *stim_gen1*, generiše potrebnu povorku 8-bitnih ulaznih signala podataka kao i selekcionog ulaza koji će omogućiti verifikaciju ispravnog rada 8-bitnog multipleksera. Slično, drugi stimulus

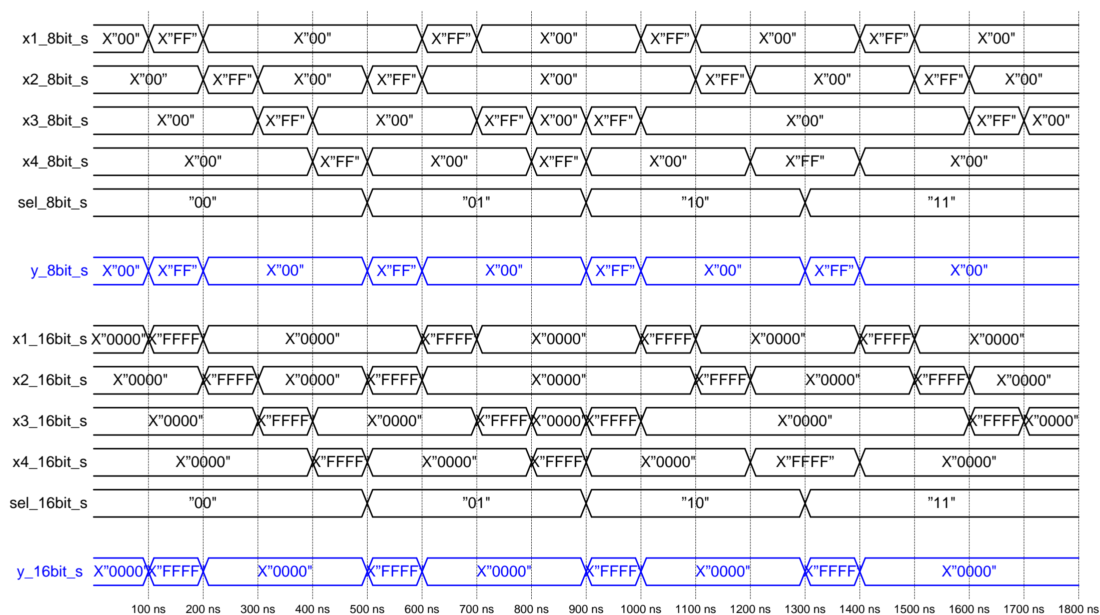
generator, *stim_gen2*, generiše povorku ulaznih signala potrebnih za verifikaciju 16-bitnog multipleksera.

Talasni oblici koji će biti generisani pomoću ove dve *process* naredbe prikazani su na slici 3.



Slika 3. Generisani talasni oblici na ulaznim signalima 8-bitnog i 16-bitnog multipleksera

Nakon što se izvrši simulacija kombinovanog rada testbenča i modela 8-bitnog i 16-bitnog multipleksera trebalo bi da izgledaju kao na slici 4.



Slika 4. Rezultat simulacije modela 8-bitnog i 16-bitnog multipleksera

VHDL modeli registra sa serijskim upisom i čitanjem sa parametrizovanom širinom

Entity deklaracija modela registra sa serijskim upisom i čitanjem sa parametrizovanom širinom ima sledeći izgled

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

entity sipo_reg is
   generic (WIDTH: positive := 8);
   port (clk: in std_logic;
         d:    in std_logic;  -- ulazni port podataka
         q:    out std_logic_vector(WIDTH-1 downto 0) -- izlazni port podataka
        );
end entity sipo_reg;
```

Entity deklaracija po svojoj strukturi slična je *entity* deklaraciji multipleksera 4-na-1 sa parametrizovanom širinom. U deklaraciji entiteta opet se koristi jedan parametar, WIDTH, pomoću kojega se može specificirati željena širina registra. Obzirom da se za ovu vrstu registra lako može napisati i bihevijalni i strukturni model na njemu će biti ilustrovano korišćenje parametra unutar arhitekturnog tela, kao i korišćenje *generate* naredbi u cilju pisanja modela sa parametrizovanom širinom.

Bihevijani model

Varijanta 1: Model registra sa serijskim upisom i čitanjem sa parametrizovanom širinom, baziran na korišćenju *process* i *if* naredbe

```
architecture beh1 of sipo_reg is
        signal q_s: std_logic_vector(WIDTH-1 downto 0);
begin
        reg: process (clk) is
        begin
                if (clk'event and clk = '1') then
                        q_s <= d&q_s(WIDTH-1 downto 1);
                end if;
        end process;

        q <= q_s;
end architecture beh1;
```

NAPOMENA: Obratite pažnju da smo prilikom deklaracije unutrašnjeg signala *q_s*, iskoristili parametar WIDTH da bi smo parametrizovali potrebnu širinu ovog signala. Obzirom da je reč o modelu sa parametrizovanom širinom unapred nije poznata potrebna širina registra. Ona će biti poznata tek prilikom instancioniranja komponente. Zbog ovoga je neophodno napisati model na generički način, korišćenjem parametara.

Strukturni model

Za registar sa serijskim upisom i paralelnim čitanjem sa parametrizovanom širinom moguće je razviti i strukturni model. Ovaj model koristi komponentu D flip flopa koja se zatim instancionira WIDTH puta u okviru arhitekturnog tela, kao što je prikazano u modelu u nastavku. Obratite pažnju da prilikom pisanja modela nije poznato kolika će

biti širina registra. Zbog toga nije moguće direktno instancionirati odgovarajući broj D flip flopova. Jedini način da se napiše generički strukturni model registra jeste da se iskoristi *generate* VHDL naredba.

```
architecture struct of sipo_reg is
   component d_ff is
      port (clk: in std_logic;    -- ulazni port dozvole upisa
            d:   in std_logic;    -- ulazni port podataka
            q:   out std_logic);  -- izlazni port podataka
   end component d_ff;

   signal q_s: std_logic_vector(WIDTH-1 downto 0);
begin
   sipo_reg: for i in WIDTH-1 downto 0 generate
         reg_bits: if i = WIDTH-1 generate
                     msb_bit: d_ff
                        port map (
                           clk => clk,
                           d   => d,
                           q   => q_s(WIDTH-1));
                  else generate
                     bit: d_ff
                        port map (
                           clk => clk,
                           d => q_s (i+1),
                           q => q_s (i));
                  end generate;

   q <= q_s;
end architecture struct;
```

NAPOMENA: U gornjem modelu iskorišćena je jedna *for generate* naredba da se iterativno generiše niz D flip flopova. Niz flip flopova povezan je na takav način da je izlaz prethodnog flip flopa ulaz u sledeći. Izuzetak je prvi flip flop, koji se nalazi na poziciji najveće značajnosti (MSB bit). Ulaz u ovaj flip flop treba da bude povezan sa *d* ulazom SIPO registra. Da bi se ovo korektno modelovalo iskorišćena je jedna *if generate* naredba. U svakoj iteraciji *for generate* naredbe pomoću *if generate* naredbe proverava se da li trenutni D flip flop koji se generiše na MSB poziciji. Ukoliko jeste, onda se *D* ulaz flip flopa povezuje na *D* ulaz SIPO registra. Ukoliko nije, onda se na *D* ulaz flip flopa dovodi *Q* izlaz flip flopa generisanog u prethodnoj iteraciji *for generate* naredbe.

Verifikaciono okruženje

Sledeći testbenč ilustruje proces instancioniranja dve instance SIPO registra, prve koja predstavlja 8-bitni SIPO registar, a druga 16-bitni SIPO registar.

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

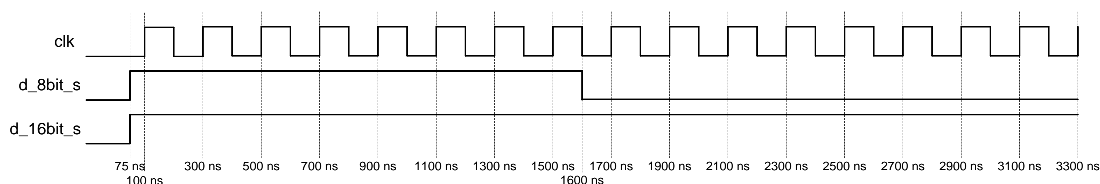
```
entity sipo_reg_tb is
end entity sipo_ reg_tb;

architecture beh of sipo_reg_tb is
   -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
   -- generatora sa ulazima DUV-a
   signal clk_s: std_logic;
   signal d_8bit_s: std_logic;
   signal q_8bit_s: std_logic_vector(7 downto 0);

   signal d_16bit_s: std_logic;
   signal q_16bit_s: std_logic_vector(15 downto 0);

begin
         -- Komponente koja se verifikuju
         duv1: entity work.sipo_reg(beh1)
            generic map (WIDTH => 8)
            port map (
               clk => clk_s,
               d => d_8bit_s,
               q => q_8bit_s);

         duv2: entity work.sipo_reg(struct)
            generic map (WIDTH => 16)
            port map (
               clk => clk_s,
               d => d_16bit_s,
               q => q_16bit_s);

         -- Klok generator koji generise periodicni clk_s signal koji
         -- ce se koristiti za aktiviranje registara
         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-ova na osnovu kojih ce biti moguce
         -- proveriti da li DUV-ovi implementira potrebnu
         -- funkcionalnost
         stim_gen: process
         begin
                  d_8_bit_s <= ‘0’, ‘1’ after 75 ns, ‘0’ after 1600 ns;

                  d_16_bit_s <= ‘0’, ‘1’ after 75 ns;

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

Prikazani testbenč instancionira dve instance komponente SIPO registra. Prva instanca, *duv1*, predstavlja 8-bitni SIPO registar, što se može videti na osnovu vrednosti koja je pridružena WIDTH parametru (WIDTH ima vrednost 8). Takođe, ova instanca treba da koristi *beh1* bihevijalni model prilikom simulacije ili automatske sinteze. Druga instanca, *duv2*, predstavlja 16-bitni SIPO registar. Ova instanca treba da bude modelovana korišćenjem strukturnog modela, opisanog arhitekturom *struct*.

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

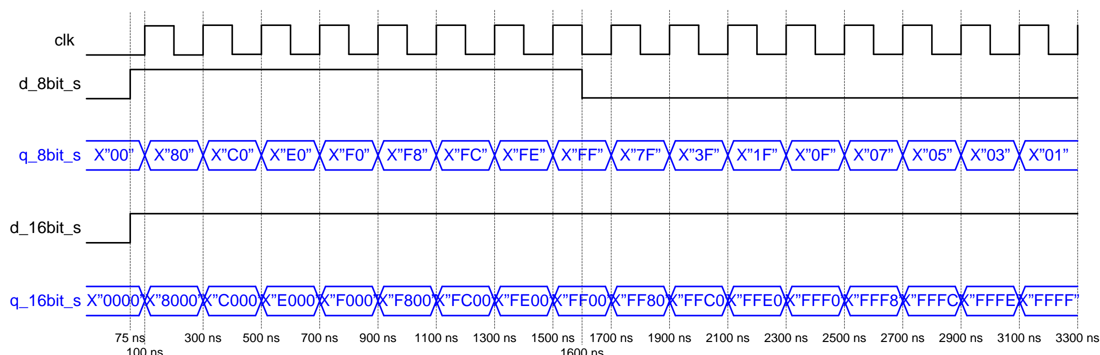
Stim_gen proces generiše vremenske oblike za signale *d_8bit_s* i *d_16bit_s* koristeći dve naredbe sekvencijalne dodele vrednosti signalu.

Talasni oblici koji će biti generisani pomoću ove dve *process* naredbe prikazani su na slici 5.



Slika 5. Generisani talasni oblici na ulaznim signalim 8-bitnog i 16-bitnog SIPO registra

Nakon što se izvrši simulacija kombinovanog rada testbenča i modela 8-bitnog i 16-bitnog SIPO registra, talasni oblik izlaznih signala podataka *q_8bit_s* i *q_16bit_s* trebalo bi da izgleda kao na slici 6.



Slika 6. Generisani talasni oblici ulaznim signalima 8-bitnog i 16-bitnog SIPO registra zajedno sa talasnim oblicima izlaznih signala podataka *q_8bit_s* i *q_16bit_s* koji su dobijeni kao rezultat simulacije

Zadaci za vežbu

Zadatak 1:

Napisati model demultipleksera 1-na-4 sa parametrizovanom širinom ulaznih i izlaznog porta podataka. Za razvijeni VHDL model napisati odgovarajući testbenč u okviru kojega će biti instancionirane dve instance parametrizovanog demultipleksera, širine 8 i 10 bita. Izvršiti funkcionalnu verifikaciju modela.

Zadatak 2:

Napisati model komparatora veličine sa parametrizovanom širinom ulaznih portova za podatke i sa tri izlaza, *eq*, *lt* i *gt*, koji daju informaciju o tome da li su ulazni brojevi jednaki, da li je broj *a* manji od broja *b* i da li je broj *a* veći od broja *b*. Za razvijeni VHDL model napisati odgovarajući testbenč u okviru kojega će biti instancionirane dve instance parametrizovanog komparatora jednakosti, širine 16 i 32 bita. Izvršiti funkcionalnu verifikaciju modela.

Zadatak 3:

Napisati model PIPO registra sa parametrizovanom širinom. Napisati odgovarajući bihevijalni i strukturni parametrizovani model. Za razvijene VHDL modele napisati odgovarajući testbenč u okviru kojega će biti instancionirane dve instance parametrizovanog PIPO registra, širine 16 i 32 bita. 16-bitna instanca treba da koristi bihevijalni model, a 32-bitna instanca strukturni. Izvršiti funkcionalnu verifikaciju modela.

Zadatak 4:

Napisati model sa parametrizovanom širinom registarske banke sa 2 pristupa za čitanje i 2 pristupa za upis. Model treba da sadrži tri parametra:

- WIDTH – pomoću kojega se specificira širina magistrala podataka na pristupima za upis i čitanje, kao i širina registara od kojih je sastavljena registarska banka.
- NUM_REGISTERS – pomoću kojega se specificira broj registara unutar registarske banke
- ADR_BUS_WIDTH – pomoću kojega se specificira širina adresnih magistrala na pristupima za upis i čitanje

Za razvijeni VHDL model napisati odgovarajući testbenč u okviru kojega će biti instancionirane dve instance parametrizovanog registarske banke. Prva registarska banka treba da bude organizovana kao banka od 32 32-bitna registra, a druga kao banka od 64 16-bitna registra. Izvršiti funkcionalnu verifikaciju modela.

Zadatak 5:

Napisati model binarnog brojača sa parametrizovanom širinom brojačkog registra. Za razvijeni VHDL model napisati odgovarajući testbenč u okviru kojega će biti instancionirane dve instance parametrizovanog binarnog brojača, širine 8 i 10 bita. Izvršiti funkcionalnu verifikaciju modela.

VHDL modeli sa parametrizovanim ponašanjem

Kao što je u uvodnom delu već rečeno modeli sa parametrizovanim ponašanjem obično se dele u dve grupe:

- modele sa parametrizovanim vremenskim karakteristikama
- modele sa parametrizovanom funkcionalnošću koju model implementira

Da bi smo ilustrovali proces razvoja VHDL modela sa parametrizovanim ponašanjem ulaznih i izlaznih portova, prikazaćemo razvoj sledećih VHDL modela:

- D flip flopa sa parametrizovanim vremenskim karakteristikama
- modulo brojača sa parametrizovanom funkcionalnošću

VHDL model D flip flopa sa parametrizovanim vremenskim karakteristikama

Model D flip flopa sa parametrizovanim vremenskim karakteristikama koji će biti razvijen uključivaće sledeće parametre:

- parametar *Tcq* – koji specificira vreme potrebno da se novoupisana vrednost u D flip flop pojavi na njegovom *Q* izlazu nakon nailaska rastuće ivice *clk* signala
- parametar *Tsetup* – vreme uspostavljanja na *D* ulazu flip flopa
- parametar *Thold* – vreme držanja na *D* ulazu flip flopa

Imajući ovo u vidu, kompletna e*ntity* deklaracija parametrizovanog modela D flip flopa ima sledeći izgled

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

entity d_ff is
   generic (Tcq:      time := 0 ns;
            Tsetup: time := 0 ns;
            Thold:  time := 0 ns
           );
   port (clk: in std_logic;    -- klok ulaz
         d:    in std_logic;    -- ulazni port podataka
         q:    out std_logic); -- izlazni port podataka
end entity d_ff;
```

Analizom *entity* deklaracije parametrizovanog modela D flip flopa možemo videti da se ni jedan od deklarisanih parametara ne koristi u okviru dela za deklaraciju portova entiteta. Ovo je očekivano jer ni jedan od deklarisanih parametara nije parametar širine. Svi deklarisani parametri (T_{cq}, T_{setup} i T_{hold}) opisuju vremenske karakteristike D flip flopa i za očekivati je da će biti korišćeni unutar arhitekturnog tela, prilikom modelovanja funkcionalnosti D flip flopa. Podrazumevane vrednosti za svaki od tri parametra su *0 ns*, što znači da će ukoliko se prilikom instancioniranja komponente *d_ff* oni eksplicitno ne navedu, biti instancionirana komponenta idealnog D flip flopa.

Bihevijani model

Varijanta 1: Model D flip flopa sa parametrizovanim vremenskim karakteristikama mogao bi da ima sledeći izgled.

```
architecture beh1 of d_ff is
begin
   dff: process is
   begin
      wait until clk'event and clk = '1';
      if (d'last_event >=  Tsetup) then
         q <= d after Tcq;
      else
         q <= 'X';
      end if;
      wait until clk'delayed(Thold) = '1';
      if (d'delayed'last_event < Thold) then
         q <= 'X';
      end if;
   end process dff;

   check_setup: process is
   begin
      wait until clk = '1';
      assert d'last_event >=  Tsetup
         report "Narusavanje vremena uspostavljanja!";
   end process check_setup;

   check_hold: process is
   begin
      wait until clk'delayed(Thold) = '1';
      assert d'delayed'last_event >= Thold
      report "Narusavanje vremena drzanja!";
   end process check_hold;
end architecture beh1;
```

Prikazani model sastoji se iz tri procesa.

Prvi proces zapravo predstavlja model D flip flopa koji modeluje rad D flip flopa uzimajući u obzir realne vremenske karakteristike. Na početku proces čeka na nailazak rastuće ivice na *clk* ulazu. Kada se detektuje rastuća ivica, proverava se da li je bilo promena na d ulazu u periodu od *Tsetup* vremena pre nailaska rastuće ivice. Ukoliko je bilo promena na *d* ulazu u tom periodu znači da je došlo do narušavanja vremena uspostavljanja na *d* ulazu i da će D flip flop ući u metastabilno stanje. Ulazak u metastabilno stanje modelovan je dodelom vrednosti 'X' (*unknown*) izlaznom portu *q*. Ukoliko nije došlo do narušavanja vremena uspostavljanja q izlazu se dodeljuje trenutna vrednost koja se nalazi na d ulazu, ali sa kašnjenjem od *Tcq* vremena. Nakon toga proces čeka na pojavu zakašnjene rastuće ivice na *clk* ulazu. Clk ulaz je zakašnjen za tačno *Thold* vremena. Kada se detektuje pojava ove zakasnele rastuće ivice proverava se da li je *d* ulaz bio stabilan barem *Thold* vremena pre nailaska ove zakasnele rastuće ivice na *clk* ulazu. Ova provera je ekvivalentna proveri da li je d ulaz bio stabilan barem *Thold* vremena nakon nailaska rastuće ivice na originalnom *clk* ulazu. Ukoliko se *d* ulaz menjao u ovom vremenskom intervalu znači da je došlo

do narušavanja vremena držanja na *d* ulazu u flip flop te će flip flop ponovo ući u metastabilno stanje pa se q izlaz opet postavlja na vrednost 'X'. Nakon ove provere *dff* proces se zaustavlja, čekajući nailazak sledeće rastuće ivice na *clk* ulazu.

Preostala dva procesa (*check_setup* i *check_hold*) koriste *assert* naredbe u cilju provere ispunjenosti vremena uspostavljanja (*check_setup* process) i vremena držanja (*check_hold* proces) na *d* ulazu flip flopa. U slučaja da barem jedna od ovih vremenskih karakteristika flip flop nije ispunjena, prilikom simulacije će biti generisana odgovarajuća poruka.

NAPOMENA: Prikazani VHDL model D flip flopa sa parametrizovanim vremenskim karakteristikama može se koristiti samo za potrebe simulacije. Ako se pokuša automatska sinteza ovog modela dobiće se poruka o grešci od strane alata za sintezu.

Verifikaciono okruženje

Primer jednog mogućeg testbenča koji se može iskoristiti za verifikaciju D flip flopa, osetljivog na rastuću ivicu *clk* signala, prikazan je u nastavku.

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

entity d_ff_tb is
end entity d_ff_tb;


architecture beh of d_ff_tb is
   -- Deklaracija komponente VHDL modula koji se verifikuje (DUV)
   -- U ovom slučaju je to D flip flop
   component d_ff is
         generic (Tcq:      time := 0 ns;
                  Tsetup: time := 0 ns;
                  Thold:  time := 0 ns
                 );
         port (clk:      in std_logic;   -- ulazni port dozvole upisa
               d:        in std_logic;   -- ulazni port podataka
               q:        out std_logic);-- izlazni port podataka 1
   end component d_ff;

   -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
   -- generatora sa ulazima DUV-a
   signal d_s, clk_s: std_logic;
   signal q_s: std_logic;

begin
         -- Komponenta koja se verifikuje
         duv: d_ff
            generic map (
               Tcq      => 25 ns,
               Tsetup => 10 ns,
               Thold   => 10 ns)
```

```
        port map (
            clk => clk_s,
            d => d_s,
            q => q_s);

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

    -- Stimulus generator koji generise potrebne vrednosti na
    -- D ulaznom portu DUV-a na osnovu kojih ce biti moguce
    -- proveriti da li DUV implementira potrebnu funkcionalnost
    stim_gen: process
    begin
            d_s <= '0', '1' after 75 ns, '0' after 125 ns,
                        '1' after 295 ns, '0' after 325 ns,
                        '1' after 475 ns, '0' after 525 ns,
                        '1' after 675 ns, '0' after 705 ns,
                        '1' after 875 ns, '0' after 925 ns,
                        '1' after 1095 ns, '0' after 1105 ns,
                        '1' after 1275 ns, '0' after 1325 ns;
            wait;
    end process;
end architecture beh;
```

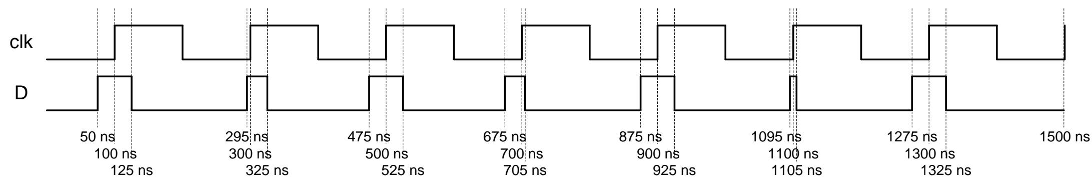
Prikazani testbenč instancionira komponentu D flip flop i koristeći *clk_gen* i *stim_gen* procese generiše povorku ulaznih signala koja može da se iskoristi za proveru ispravnog rada napisanog modela D flip flopa.

Clk_gen proces generiše periodičnu povorku pravougaonih impulsa na signalu *clk_s*. Perioda generisanog signala iznosi 200 ns.

Stim_gen proces generiše vremenski oblik za signal *d_s* koristeći jednu naredbu sekvencijalne dodele vrednosti signalu. Generisani vremenski oblik pažljivo je odabran tako da se tokom simulacije pojave sledeće karakteristične situacije na *d* ulazu u flip flop prilikom nailaska ratuće ivice na *clk* ulazu:

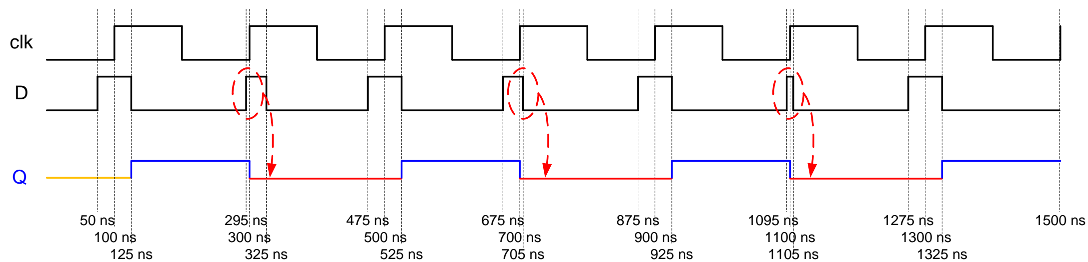
- situacija kada nije narušeno ni vreme uspostavljanja ni vreme držanja na *d* ulazu u flip flop,
- situacija kada je narušeno vreme uspostavljanja na *d* ulazu u flip flop,
- situacija kada je narušeno vreme držanja na *d* ulazu u flip flop,
- situacija kada je narušeno i vreme uspostavljanja i vreme držanja na *d* ulazu u flip flop,

Talasni oblici koji će biti generisani pomoću ove dve process naredbe prikazani su na slici 7.



Slika 7. Generisani talasni oblici na dva ulazna signala D flip flopa

Nakon što se izvrši simulacija kombinovanog rada testbenča i modela D flip flopa talasni oblik izlaznog signala podataka *q_s* trebalo bi da izgleda kao na slici 8.



Slika 8. Generisani talasni oblici na dva ulazna signala D flip flopa zajedno sa talasnim oblicima izlaznong signala podataka *q_s* koji je dobijen kao rezultat simulacije

Na slici 8 se jasno vide trenuci kada dolazi do narušavanja jednog ili oba vremenska ograničenja prilikom rada flip flopa, vremena uspostavaljanja ili vremena držanja. Model D flip flopa je napisan na takav način da se na *q* izlazu nakon detekcije narušavanja vremenskih karakteristika pojavljuje vrednost 'X' koja traje do trenutka nailaska sledeće rastuće ivice na kojoj ne dolazi do narušavanja vremenskih karakteristika flip flopa. Ovo je takođe jasno vidljivo na vremenskom dijagramu sa slike 8.

VHDL model modulo brojača sa parametrizovanom funkcionalnošću

Da bi smo ilustrovali mogućnost pisanja VHDL modela sa parametrizovanom funkcionalnošću u nastavku će biti prikazan model modulo brojača kod kojega je moguće parametrizovati smer brojanja kao i moduo po kome broji brojač. Da bi se ovo omogućilo potrebno je definisati sledeće parametre:

- UP – parametar koji određuje da li će brojač raditi kao brojač na gore ili kao brojač na dole. Parametar je Bulovog tipa, ako ima vrednost *true* brojač broji na gore, a ako ima vrednost *false* brojač broji na dole.
- MODUO – parametar koji određuje moduo po kome će brojati brojač
- WIDTH – potreban broj bita brojačkog registra

Entity deklaracija parametrizovanog modulo brojača ima sledeći izgled

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

entity counter is
   generic (UP: boolean := true;
            MODUO:    positive := 100;
            WIDTH:    positive := 7);
   port (clk: in std_logic;
         q:   out std_logic_vector(WIDTH-1 downto 0)
        );
end entity counter;
```

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

Bihevijani model

Varijanta 1: Model modulo brojača sa parametrizovanom funkcionalnošću

```
architecture beh of counter is
   subtype cnt_range_t is integer range 0 to MODUO-1;
   signal count_s: cnt_range_t := 0;
begin
      cnt: process (clk) is
      begin
            if (clk'event and clk = '1') then
               if (UP = true) then
                  count_s <= count_s + 1;
               else
                  count_s <= count_s - 1;
               end if;
            end if;
      end process;

      q <= conv_std_logic_vector(count_s, WIDTH);
end architecture beh;
```

Verifikaciono okruženje

Primer jednog mogućeg testbenča koji se može iskoristiti za verifikaciju modulo brojača modula sa parametrizovanom funkcionalnošću prikazan je u nastavku.

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

entity counter_tb is
end entity counter_tb;

architecture beh of counter_tb is
```

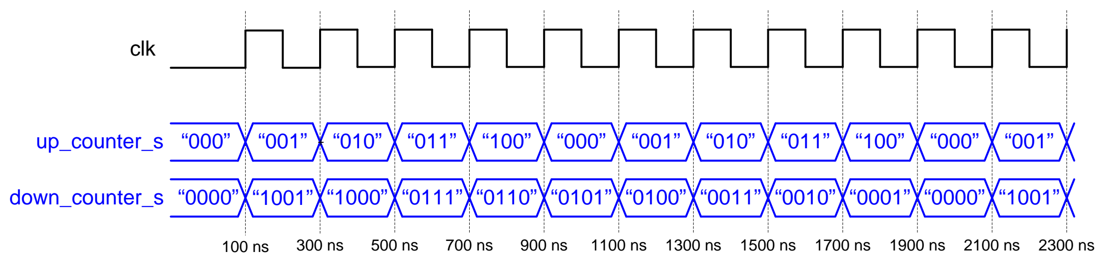
```
    -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
    -- generatora sa ulazima DUV-a
    signal clk_s: std_logic;
    signal up_counter_s: std_logic_vector(2 downto 0);
    signal down_counter_s: std_logic_vector(3 downto 0);

begin
        -- Komponente koja se verifikuju
        up_counter: entity work.counter (beh)
            generic map (
                UP => true,
                MODUO => 5,
                WIDTH => 3)
            port map (
                clk => clk_s,
                q => up_counter_s);

        down_counter: entity work.counter (beh)
            generic map (
                UP => false,
                MODUO => 10,
                WIDTH => 4)
            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 parametrizovanih modulo brojača, ali prilikom svakog instancioniranja instancioniranom entitetu vrši drugačiju parametrizaciju. Prva instanca je modulo brojač na gore koji broji po modulu 5, a druga instanca je modulo brojač na dole koji broji po modulu 10.

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 9.



Slika 8. Generisani talasni oblici na izlazima modulo brojača na gore modula 5 i modulo brojača na dole modula 10, koji su dobijeni kao rezultat simulacije

Zadaci za vežbu

Zadatak 1:

Napisati VHDL model 8-bitnog registra sa paralelnim ulazom i paralelnim izlazom koji će biti parametrizovanim vremenskim karakteristikama na sledeći način. Za svaki od osam flip flopova postoji po jedan odvojeni parametar, *Tcq*, pomoću kojega je moguće specificirati vreme potrebno da se novoupisana vrednost u posmatrani flip flop pojavi na njegovom *Q* izlazu nakon nailaska rastuće ivice *clk* signala. Model treba da ima ukupno osam parametara, *Tcq1*, *Tcq2*, ..., *Tcq8*. Za razvijeni VHDL model napisati odgovarajući testbenč pomoću kojega je moguće izvršiti verifikacija ispravnog rada modela.

Zadatak 2:

Napisati VHDL model brojača koji broji od *m* do *n*, a kada stigne do vrednosti *n* ponovo počinje brojanje od *m*. Koristiti dve generika, *M* i *N*, pomoću kojih je moguće zadati vrednosti za početnu i krajnju vrednost brojanja. Za razvijeni VHDL model napisati odgovarajući testbenč pomoću kojega je moguće izvršiti verifikacija ispravnog rada modela.

Hijerarhijski dizajn

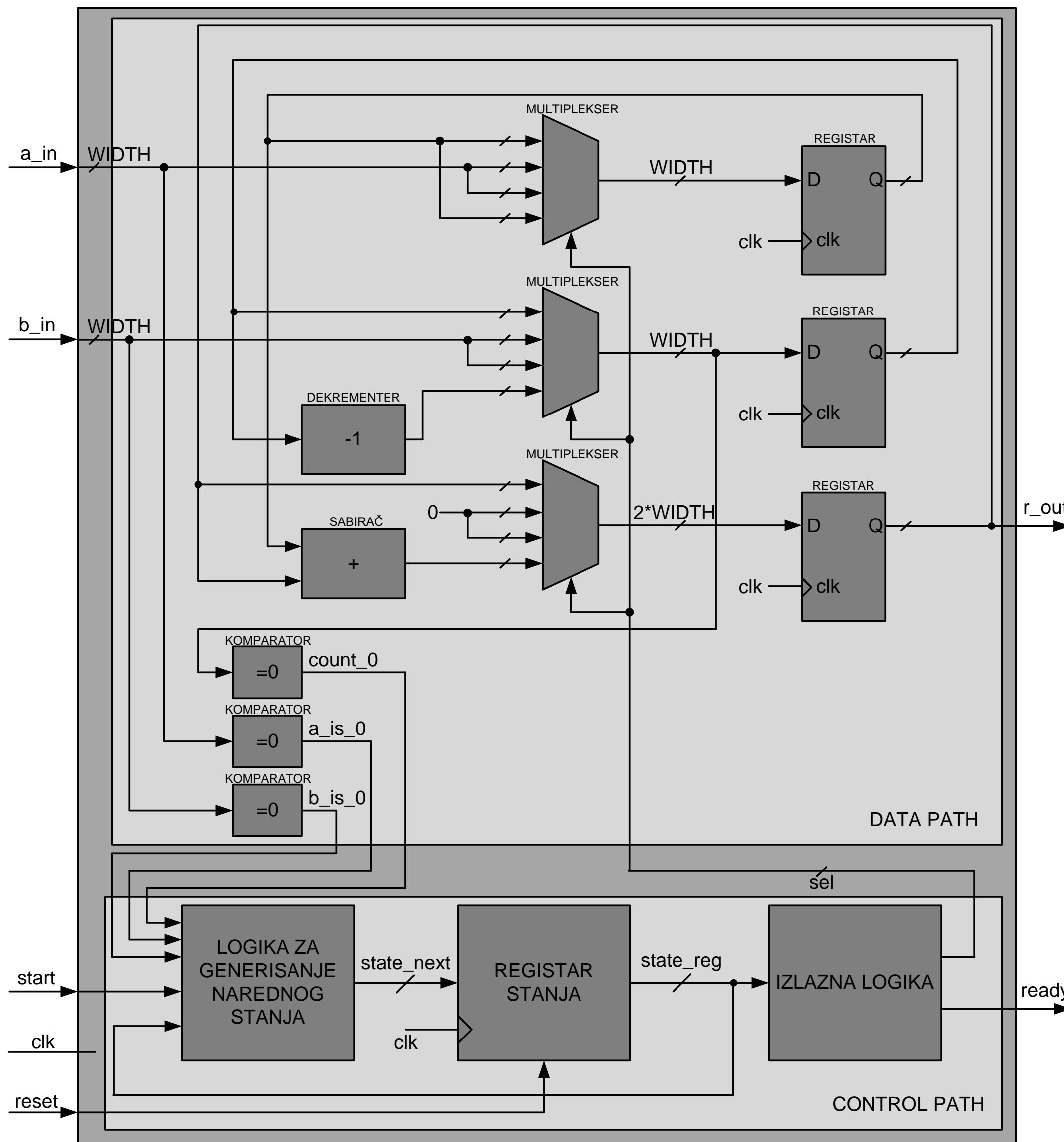
Hijerarhijski dizajn biće ilustrovan na primeru projektovanja množača baziranog na sukcesivnom sabiranju. Algoritam množenja dva broja baziran na sukcesivnom sabiranju prikazan je u nastavku.

```
if (a_in=0 or b_in=0) then
{
      r = 0;
}
else
{
      a = a_in;
      n = b_in;
      r = 0;
      while (n != 0 )
      {
         r = r + a;
         n = n - 1;
      }
}
r_out = r;
```

Ulaz u algoritam su brojevi koje je potrebno pomnožiti, zadati preko promenljivih *a_in* i *b_in*. U prvom koraku vrši se provera da li je neki od brojeva jednak nuli. Ukoliko je barem jedan od brojeva jednak nuli algoritam kao rezultat izbacuje vrednost nula (rezultat množenja smešta se u privremenu promenljivu *r*) i završava svoj rad. Ukoliko su oba broja različita od nule, algoritam ulazi u petlju koju ponavlja *b_in* puta. U svakom prolazu petlje pomoćna promenljiva r uvećava se za vrednost proenljive *a_in*. Nakon *b_in* prolaza kroz petlju, vrednost promenljive *r* iznosiće *r = b_in * a_in*, vrednost proizvoda dva ulazna broja. Nakon završetka operacije množenja, konačni rezultat smešta se u promenljivu *r_out*.

Ovaj pseudo kod algoritma lako može da se prevede u odgovarajući program (na primer napisan u C programskom jeziku), što bi predstavljalo *softversku implementaciju algoritma*. Međutim, na osnovu pseudo koda algoritma može se isprojektovati i digitalni sistem koji će ga implementirati. Ovaj postupak implementacije algoritma pomoću odgovarajućeg digitalnog sistema naziva se *hardverska implementacija algoritma* i često se primenjuje u praksi kada se žele postići optimalne performanse (najveća brzina izvršavanja algoritma, najmanja potrošnja energije prilikom izvršavanja algoritma, itd.).

Način kako se proizvoljni algoritam implementira pomoću specijalizovanog digitalnog sistema izlazi iz okvira ovog kursa. Na višim godinama studija postoje posebni kursevi koji se bave ovom problematikom. U ovom trenutku nas samo zanima kako izgleda hijerarhijska struktura odgovarajućeg digitalnog sistema koji implementira algoritam množenja brojeva baziran na sukcesivnom sabiranju, bez ulaženja u detalje kako smo do te strukture došli. Slika 9 prikazuje jednu moguću strukturu digitalnog sistema za množenje dva broja koji implementira algoritam množenja baziran na sukcesivnom sabiranju.



Slika 9. Struktura digitalnog sistema koji implementira algoritam za množenje dva broja baziran na sukcesivnom sabiranju

Sa slike 9 vidimo da je digitalni sistem sastavljen od većeg broja modula, odnosno da postoji odgovarajuća hijerarhijska struktura koja opisuje unutrašnju strukturu predloženog sistema. Ova hijerarhijska struktura prikazana je na slici 10.

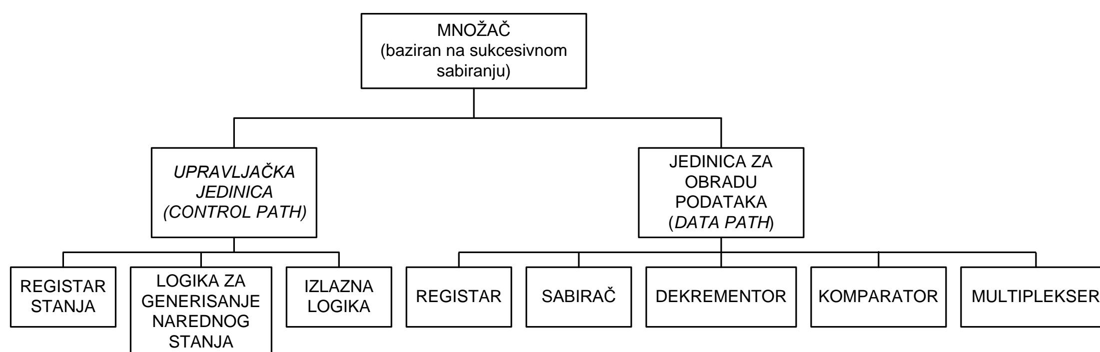
Sa slike 10 možemo videti da je na prvom nivou hijerarhije celokupan digitalni sistem podljen na dva podsistema, upravljački podsistem (*control path*) i sistem za obradu podataka (*data path*). Ovo je uobičajena dekompozicija i vrlo često se sreće u praksi prilikom projektovanja složenih digitalnih sistema. Sistem za obradu podataka implementira sve potrebne operacije koji će biti neophodne prilikom izvršavanja algoritma i obezbeđuje memorijski prostor za smeštanje ulaznih podataka, međurezultata kao i izlaznih podataka. Upravljački podsistem upravlja radom podsistema za obradu podataka, kontrolišući redosled izvršavanja operacija i manipulacije podacima i međurezultatima na osnovu implementiranog algoritma.

Na sledećem nivou hijerarhije sa slike 10 možemo videti da se upravljački podsistem dalje dekomponuje na tri podsistema: registar stanja, logiku za generisanje narednog stanja i izlaznu logiku. Obzirom da se upravljački podsistem najčešće implementira kao konačni automat, ovo je standardna dekompozicija.

Istovremeno, podsistem za obradu podataka dekomponovan je na sledeće podsisteme:

- skupa registara – koji služe za smeštanje ulaznih podataka, međurezultata do kojih se dolazi prilikom izvršavanja algoritma, kao i konačnih rezultata izvršavanja algoritma
- sabirača – kao jedne od funkcionalnih jedinica neophodnih za izvršavanje numeričkih operacija definisanih u algoritmu
- dekrementera – kao druge funkcionalne jedinice zahtevane od strane algoritma koji se implementira
- komparatora – kao treće zahtevane funkcionalne jedinice
- multipleksera – koji služe za rutiranje podataka ka odredišnim registrima u koje će biti smešteni

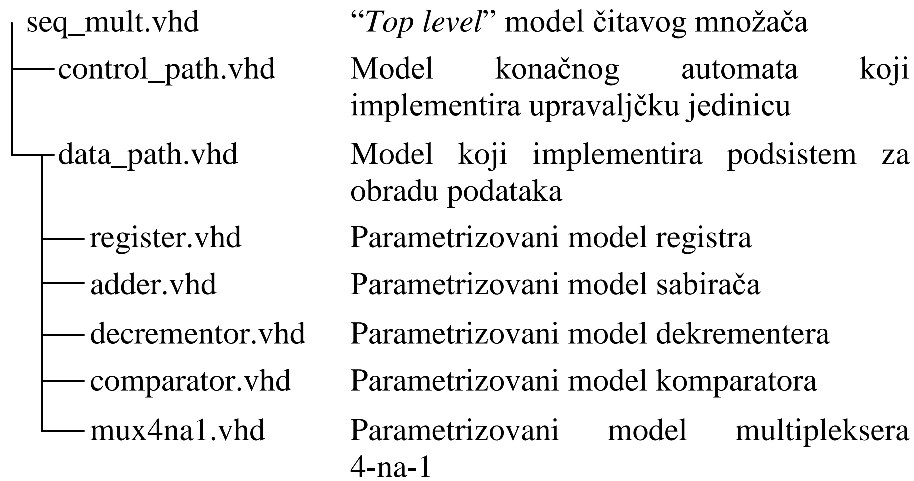
Obzirom da svaki od ovih podsistema predstavlja standardnu kombinacionu ili skevencijalnu mrežu, proces dalje dekompozicije sistema se u ovom trenutku završava.



Slika 10. Hijerarhijska struktura digitalnog sistema koji implementira algoritam za množenje dva broja baziran na sukcesivnom sabiranju

Nakon što je utvrđena hijerarhijska struktura sistema (prikazana na slici 10) koji se želi isprojektovati i nacrtan blok dijagram koji pokazuje kako su moduli od kojih je sistem sastavljen međusobno povezani i kakav im je interfejs (slika 9), može se pristupiti pisanju VHDL modela.

Kao što je ranije rečeno dobra praksa je da se VHDL model svakog od potrebnih modula piše u odvojenom dizajn fajlu. Imajući ovo u vidu, model množača baziranog na sukcesivnom sabiranju ima strukturu VHDL fajlova prikazanu na slici 11.



Slika 11. Struktura dizajn fajlova za množač baziran na sukcesivnom sabiranju

Konačno možemo pristupiti pisanju VHDL modela množača.

"*Top level*" VHDL model množača

Kao i obično, prvi korak prilikom pisanja VHDL modela nekog sistema je da se napiše njegova *entity* deklaracija. Sekvencijalni množač koji razvijamo ima sledeće portove:

- *a_in* – ulazni port preko koga se zadaje vrednost prvog činioca u operaciji množenja
- *b_in* – ulazni port preko koga se zadaje vrednost drugog činioca u operaciji množenja
- *start* – ulazni kontrolni port preko kojega se startuje proces množenja. Dokle god je ovaj port jednak nuli, množač se nalazi u stanju mirovanja čekajući na komandu. Kada se ovaj port postavi na jedinicu započinje proces množenja sa operandima koji odgovaraju trenutnim vrednostima na ulaznim portovima *a_in* i *b_in*.
- *r_out* – izlazni port preko koga se prosleđuje rezultat množenja. Vrednost ovog izlaznog porta je validna samo ako je vrednost ready izlaznog porta jednaka jedan.
- *ready* – izlazni port preko kojega se saopštava trenutni status množača. Kada je ready postavljen na jedinicu, množač se nalazi u stanju mirovanja i spreman je da započne sledeću operaciju množenja.
- *clk* – ulazni port za dovođenje globalnog sinhronizacionog signala
- *reset* – ulazni port za inicijalizaciju sistema. Reč je o sinhronom ulazu.

Da bi smo povećali mogućnost ponovnog korišćenja modela, izvršićemo njegovu parametrizaciju. U ovom slučaju moguće je izvršiti parametrizaciju širine ulaznih oprtova za operande *a_in* i *b_in* kao i izlaznog porta za rezultat množenja, *r_out*. Parametrizaciju ćemo izvršiti uvođenjem jednog parametra, WIDTH, pomoću kojega će biti moguće specificirati broj bita koji se koristi za predstavljanje ulaznih brojeva koje je potrebno pomnožiti.

Na osnovu ovih informacija možemo napisati *entity* deklaraciju sekvencijalnog množača.

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

entity seq_mult is
   generic (WIDTH: positive := 8);

   port (clk:    in std_logic;
         reset:  in std_logic;

         a_in:   in std_logic_vector(WIDTH-1 downto 0);
         b_in:   in std_logic_vector(WIDTH-1 downto 0);
         start:  in std_logic;

         r_out:  out std_logic_vector(2*WIDTH-1 downto 0);
         ready:  out std_logic);
end entity seq_mult;
```

Arhitektura, koja predstavlja strukturni model sekvencijalnog množača napisan na osnovu slike 9, prikazana je u nastavku.

```
architecture struct of seq_mult is
        signal a_is_0_s, b_is_0_s, count_0_s: std_logic;
        signal sel_s: std_logic_vector(1 downto 0);
begin
        -- Upravljacka jedinica
        controlpath: entity work.control_path(beh)
                port map (
                        clk => clk,
                        reset => reset,
                        start => start,
                        count_0 => count_0_s,
                        a_is_0 => a_is_0_s,
                        b_is_0 => b_is_0_s,
                        sel => sel_s,
                        ready => ready);

        -- Podsistem za obradu podataka
        datapath: entity work.data_path(struct)
                generic map (WIDTH => WIDTH)
                port map (
                        clk => clk,
                        a_in => a_in,
                        b_in => b_in,
                        sel => sel_s,
                        count_0 => count_0_s,
                        a_is_0 => a_is_0_s,
                        b_is_0 => b_is_0_s,
                        r_out => r_out);
end architecture struct;
```

VHDL model upravljačke jedinice

Upravljačka jedinica ima sledeće portove:

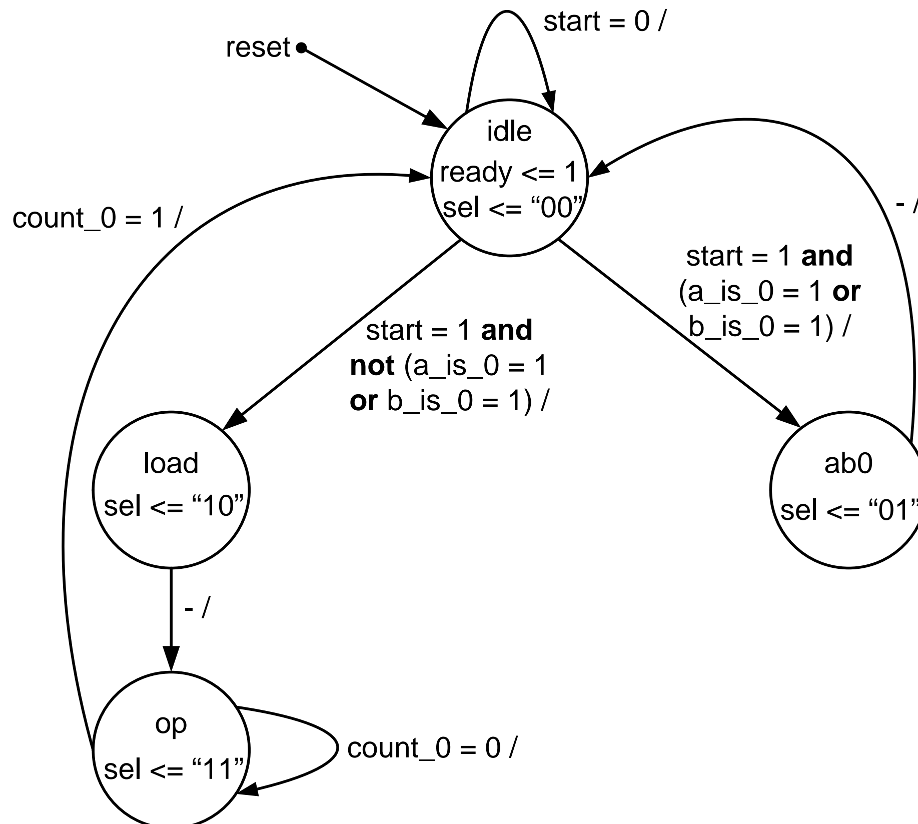
- *start* – ulazni kontrolni port preko kojega se startuje proces množenja. Dokle god je ovaj port jednak nuli, množač se nalazi u stanju mirovanja čekajući na komandu. Kada se ovaj port postavi na jedinicu započinje proces množenja sa operandima koji odgovaraju trenutnim vrednostima na ulaznim portovima *a_in* i *b_in*.
- *count_0* – ulazni port preko koga se prosleđuje informacija da li je rezultat operacije dekrementovanja jednak nuli. Kada ovo bude slučaj, count_0 se aktivira, a to je indikacija upravljačkoj jedinici da je tekući proces množenja završen.
- *a_is_0* – ulazni port preko kojega se saopštava da li je trenutna vrednost prvog operanda jednaka nuli.
- *b_is_0* – ulazni port preko kojega se saopštava da li je trenutna vrednost drugog operanda jednaka nuli.
- *clk* – ulazni port za dovođenje globalnog sinhronizacionog signala
- *reset* – ulazni port za inicijalizaciju sistema. Reč je o sinhronom ulazu.

Na osnovu ovih informacija možemo napisati *entity* deklaraciju upravljačke jedinice sekvencijalnog množača.

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

entity control_path is
   port (clk:      in std_logic;
         reset:    in std_logic;

         start:    in std_logic;

         count_0: in std_logic;
         a_is_0:   in std_logic;
         b_is_0:   in std_logic;

         sel:       out std_logic_vector(1 downto 0);
         ready:    out std_logic);
end entity control_path;
```

Blok dijagram konačnog automata koji je implementiran unutar upravljačke jedinice, prikazan je na slici 12. Konačni automat koji implementira algoritam množenja brojeva baziran na sukcesivnom sabiranju ima ukupno četiri stanja: *idle*, *ab0*, *load* i *op*. Konačni automat se inicijalno nalazi u *idle* stanju u kojem čeka na komandu za množenje brojeva. Kada ulazni signal *start* postane aktivan to predstavlja indikaciju konačnom automatu da je potrebno započeti novu operaciju množenja dva broja. U zavisnosti od toga da li je barem jedan od operanada jednak nuli ili su oba operanda različita od nule, konačni automat odlazi ili u *ab0* ili u *load* stanje.Stanje *ab0* posećuje se ako je barem jedan od ulaznih operanada jednak nuli. U ovom slučaju rezultat

operacije množenja će sigurno biti jednak nuli, te se sada operacija množenja i ne izvodi već se automat odmah vraća u *idle* stanje. Ukoliko su oba operanda različita od nule, konačni automat odlazi u *load* stanje u kojem se u unutrašnje registre smeštaju trenutne vrednosti činilaca u predstojećoj operaciji množenja. Automat odmah, bezuslovno prelazi u *op* stanje u kojem se zapravo izvodi proces množenja baziran na sukcesivnom sabiranju. U *op* stanju automat proverava vrednost *count_0* ulaznog porta i ukoliko je on aktivan (jednak jedinici) to predstavlja indikaciju da je tekuća operacija množenja završena, te se automat vraća u *idle* stanje. Ukoliko je count_0 ulaz neaktivan (jednak nuli), automat ostaje u *op* stanju te se operacija sukcesivnog sabiranja se nastavlja.



Slika 12. Blok dijagram stanja konačnog automata koji je implementiran unutar upravljačke jedinice

Arhitektura, koja model konačnog automata čiji je dijagram stanja prikazan na slici 12, prikazana je u nastavku.

```
architecture beh of control_path is
        type state_type is (idle, ab0, load, op);
        signal state_reg, state_next: state_type;
begin
        -- control path: state register
        process (clk, reset)
        begin
           if reset = '1' then
               state_reg <= idle;
```

```
        elsif (clk'event and clk = '1') then
            state_reg <= state_next;
        end if;
    end process;

    -- control path: next-state/output logic
    process (state_reg, start, a_is_0, b_is_0, count_0)
    begin
        case state_reg is
            when idle =>
                sel <= "00";
                ready <= '1';
                if start = '1' then
                    if (a_is_0 = '1' or b_is_0 = '1') then
                        state_next <= ab0;
                    else
                        state_next <= load;
                    end if;
                else
                    state_next <= idle;
                end if;

            when ab0 =>
                sel <= "01";
                ready <= '0';
                state_next <= idle;

            when load =>
                sel <= "10";
                ready <= '0';
                state_next <= op;

            when op =>
                sel <= "11";
                ready <= '0';
                if count_0 = '1' then
                    state_next <= idle;
                else
                    state_next <= op;
                end if;
        end case;
    end process;
end architecture beh;
```

VHDL model podsistema za obradu podataka

Kao što se na slici 9 može videti, podsistem za obradu podataka sastoji se od skupa unutrašnjih registara, sabirača, dekrementora, skupa komparatora sa nulom i skupa multipleksera. Portove podsistema za obradu podataka su sledeći:

- *a_in* – ulazni port preko koga se zadaje vrednost prvog činioca u operaciji množenja

- *b_in* – ulazni port preko koga se zadaje vrednost drugog činioca u operaciji množenja
- *sel* – ulazni kontrolni port preko kojega se kontroliše rad multipleksera i određuje koji će se podaci upisati u unutrašnje registre u narednom taktu.
- *r_out* – izlazni port preko koga se prosleđuje rezultat množenja. Vrednost ovog izlaznog porta je validna samo ako je vrednost ready izlaznog porta jednaka jedan.
- *count_0* – izlazni port preko koga se prosleđuje informacija da li je rezultat operacije dekrementovanja jednak nuli. Kada ovo bude slučaj, count_0 se aktivira, a to je indikacija upravljačkoj jedinici da je tekući proces množenja završen.
- *a_is_0* – izlazni port preko kojega se saopštava da li je trenutna vrednost prvog operanda jednaka nuli.
- *b_is_0* – izlazni port preko kojega se saopštava da li je trenutna vrednost drugog operanda jednaka nuli.
- *clk* – ulazni port za dovođenje globalnog sinhronizacionog signala

Na osnovu ovih informacija možemo napisati *entity* deklaraciju sekvencijalnog množača.

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

entity data_path is
        generic (WIDTH: positive := 8);

        port (clk:      in std_logic;

                a_in:     in std_logic_vector(WIDTH-1 downto 0);
                b_in:     in std_logic_vector(WIDTH-1 downto 0);

                sel:      in std_logic_vector(1 downto 0);

                count_0: out std_logic;
                a_is_0:  out std_logic;
                b_is_0:  out std_logic;

                r_out:    out std_logic_vector(2*WIDTH-1 downto 0));
end entity data_path;
```

NAPOMENA: Obzirom da razvijamo model sekvencijalnog množača sa parametrizovanom širinom ulaznih i izlaznih portova podataka, i *entity* deklaracija podsistema za obradu podataka mora takođe da se parametrizuje. Ovo je neophodno jer podsistem za obradu podataka sadrži registre u kojima će biti smešteni podaci, funkcionalne jedinice pomoću kojih će se podaci transformisati i multipleksere

pomoću kojih će se upravljati tokom podataka. Da bi čitav sistem bio pravilno parametrizovan i svi ovi podmoduli moraju se takođe parametrizovati.

Arhitektura, koja predstavlja strukturni model podsistema za obradu podataka napisan na osnovu slike 9, prikazana je u nastavku.

```
architecture struct of data_path is
   signal a_reg_s, a_next_s: std_logic_vector(WIDTH-1 downto 0);
   signal n_reg_s, n_next_s: std_logic_vector(WIDTH-1 downto 0);
   signal r_reg_s, r_next_s: std_logic_vector(2*WIDTH-1 downto 0);
   signal adder_out_s: std_logic_vector(2*WIDTH-1 downto 0);
   signal sub_out_s: std_logic_vector(WIDTH-1 downto 0);
begin
         -- Povezivanje unutrasnjeg signala sa izlaznim portom
         r_out <= r_reg_s;

         -- Unutrasnji registri
         a_reg: entity work.reg(beh)
                  generic map (WIDTH => WIDTH)
                  port map (clk => clk,
                                 d => a_next_s,
                                 q => a_reg_s);

         n_reg: entity work.reg(beh)
                  generic map (WIDTH => WIDTH)
                  port map (clk => clk,
                                 d => n_next_s,
                                 q => n_reg_s);

         r_reg: entity work.reg(beh)
                  generic map (WIDTH => 2*WIDTH)
                  port map (clk => clk,
                                 d => r_next_s,
                                 q => r_reg_s);

         -- Funkcionalne jedinice
         add: entity work.adder(beh)
                  generic map (WIDTH => WIDTH)
                  port map (op1 => a_reg_s,
                                 op2 => r_reg_s,
                                 res => adder_out_s);

         decrement: entity work.decrementor(beh)
                  generic map (WIDTH => WIDTH)
                  port map (op1 => n_reg_s,
                                 res => sub_out_s);

         zero_comp1: entity work.comparator(beh)
                  generic map (WIDTH => WIDTH)
                  port map (op1 => n_next_s,
                                 res => count_0);
```

```
zero_comp2: entity work.comparator(beh)
        generic map (WIDTH => WIDTH)
        port map (op1 => a_in,
                  res => a_is_0);

zero_comp3: entity work.comparator(beh)
        generic map (WIDTH => WIDTH)
        port map (op1 => b_in,
                  res => b_is_0);

-- Multiplekseri za rutiranje
mux1: entity work.mux4na1(beh)
        generic map (WIDTH => WIDTH)
        port map (x1 => a_reg_s,
                  x2 => a_in,
                  x3 => a_in,
                  x4 => a_reg_s,
                  sel => sel,
                  y   => a_next_s);

mux2: entity work.mux4na1(beh)
        generic map (WIDTH => WIDTH)
        port map (x1 => n_reg_s,
                  x2 => b_in,
                  x3 => b_in,
                  x4 => sub_out_s,
                  sel => sel,
                  y   => n_next_s);

mux3: entity work.mux4na1(beh)
        generic map (WIDTH => 2*WIDTH)
        port map (x1 => r_reg_s,
                  x2 => conv_std_logic_vector(0, 2*WIDTH),
                  x3 => conv_std_logic_vector(0, 2*WIDTH),
                  x4 => adder_out_s,
                  sel => sel,
                  y   => r_next_s);
end architecture struct;
```

VHDL model registra sa paralelnim upisom i paralelnim čitanjem

Registar sa paralelnim upisom i čitanjem je standardna sekvencijalna mreža čiji je model ranije prikazan i detaljno analiziran. Radi kompletnosti prikaza modela sekvencijalnog množanja, ovde ćemo samo priložiti gotov VHDL model registra sa paralelnim upisom i čitanjem, bez detaljnog komentarisanja.

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

entity reg is
   generic (WIDTH: positive := 8);
```

```
   port (clk: in std_logic;
         d:    in std_logic_vector(WIDTH-1 downto 0);
         q:    out std_logic_vector(WIDTH-1 downto 0));
end entity reg;

architecture beh of reg is
begin
      reg: process (clk) is
      begin
            if (clk'event and clk = '1') then
                  q <= d;
            end if;
      end process;
end architecture beh;
```

VHDL model sabirača

Sabirač takođe predstavlja standardnu kombinacionu mrežu čije je modelovanje ranije već bilo razmatrano. U nastavku je priložen jedan mogući VHDL model sabirača koji bi se mogao koristiti prilikom razvoja sekvencijalnog množača.

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

entity adder is
   generic (WIDTH: positive := 8);
   port (op1:  in std_logic_vector(WIDTH-1 downto 0);
         op2:  in std_logic_vector(2*WIDTH-1 downto 0);
         res:   out std_logic_vector(2*WIDTH-1 downto 0));
end entity adder;

architecture beh of adder is
begin
      add: process (op1, op2) is
      begin
            res <= ("00000000" & op1) + op2;
      end process;
end architecture beh;
```

VHDL model dekrementora

Dekrementor takođe predstavlja standardnu kombinacionu mrežu čije je modelovanje ranije već bilo razmatrano. U nastavku je priložen jedan mogući VHDL model dekrementora koji bi se mogao koristiti prilikom razvoja sekvencijalnog množača.

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

entity decrementor is
```

```
   generic (WIDTH: positive := 8);
   port (op1:  in std_logic_vector(WIDTH-1 downto 0);
         res:  out std_logic_vector(WIDTH-1 downto 0));
end entity decrementor;

architecture beh of decrementor is
begin
        decrement: process (op1) is
        begin
                res <= op1 - 1;
        end process;
end architecture beh;
```

VHDL model komparatora

Dekrementor takođe predstavlja standardnu kombinacionu mrežu čije je modelovanje ranije već bilo razmatrano. U nastavku je priložen jedan mogući VHDL model dekrementora koji bi se mogao koristiti prilikom razvoja sekvencijalnog množača.

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

entity comparator is
   generic (WIDTH: positive := 8);
   port (op1:  in std_logic_vector(WIDTH-1 downto 0);
         res:  out std_logic);
end entity comparator;

architecture beh of comparator is
begin
        comp: process (op1) is
        begin
                if (op1 = conv_std_logic_vector(0, WIDTH)) then
                        res <= '1';
                else
                        res <= '0';
                end if;
        end process;
end architecture beh;
```

VHDL model multipleksera

Multiplesker 4-na-1 takođe predstavlja standardnu kombinacionu mrežu čije je modelovanje ranije već bilo razmatrano.

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

entity mux4na1 is
   generic (WIDTH: positive := 8);
```

```
    port (x1: in std_logic_vector(WIDTH-1 downto 0);
          x2:  in std_logic_vector(WIDTH-1 downto 0);
          x3:  in std_logic_vector(WIDTH-1 downto 0);
          x4:  in std_logic_vector(WIDTH-1 downto 0);
          sel:  in std_logic_vector(1 downto 0);
          y:   out std_logic_vector(WIDTH-1 downto 0));
end entity mux4na1;

architecture beh of mux4na1 is
begin
        mux: process (x1, x2, x3, x4, sel) is
        begin
                case sel is
                        when "00" =>
                                y <= x1;
                        when "01" =>
                                y <= x2;
                        when "10" =>
                                y <= x3;
                        when others =>
                                y <= x4;
                end case;
        end process;
end architecture beh;
```

Verifikaciono okruženje

Nakon projektovanja sekvencijalnog množača, moramo razviti odgovarajući testbenč koji može poslužiti za funkcionalnu verifikaciju razvijenog modela množača. U nastavku sledi primer jednog testbenča koji se može iskoristiti u tu svrhu.

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

entity seq_mult_tb is
end entity seq_mult_tb;


architecture beh of seq_mult_tb is
    constant width_c: integer := 8;

    -- Deklaracija unutrašnjih signala potrebnih za povezivanje stimulus
    -- generatora sa ulazima DUV-a
    signal clk_s: std_logic;
    signal reset_s: std_logic;
    signal start_s: std_logic;
    signal a_in_s: std_logic_vector(width_c-1 downto 0);
    signal b_in_s: std_logic_vector(width_c-1 downto 0);

    signal ready_s: std_logic;
```

```
    signal res_s: std_logic_vector(2*width_c-1 downto 0);

begin
        -- Komponente koja se verifikuju
        duv: entity work.seq_mult(struct)
           generic map (WIDTH => width_c)
           port map (
              clk    => clk_s,
              reset  => reset_s,
              start  => start_s,
              a_in   => a_in_s,
              b_in   => b_in_s,
              r_out => res_s,
              ready => ready_s);

        -- Klok generator koji generise periodicni clk_s signal koji
        -- ce se koristiti za aktiviranje registara
        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-ova na osnovu kojih ce biti moguce
        -- proveriti da li DUV implementira potrebnu
        -- funkcionalnost
        stim_gen: process
                type op_array_t is array (1 to 10) of integer;
                variable a_v: op_array_t := (3, 5, 23, 55, 79, 123, 145,
                                             178, 201, 224);
                variable b_v: op_array_t := (2, 9, 13, 35, 89, 132, 153,
                                             163, 211, 254);
        begin
                -- Inicijalizacija signala
                start_s <= '0';
                a_in_s <= (others => '0');
                b_in_s <= (others => '0');

                -- Resetujmo sistem
                reset_s <= '1';
                for i in 1 to 2 loop
                   wait until falling_edge(clk_s);
                end loop;
                reset_s <= '0';
                wait until falling_edge(clk_s);

                for i in 1 to 10 loop
                   -- Dovedi operande koje je potrebno pomnoziti
                   a_in_s <= conv_std_logic_vector(a_v(i), width_c);
                   b_in_s <= conv_std_logic_vector(b_v(i), width_c);

                   -- Startuj mnozac
```

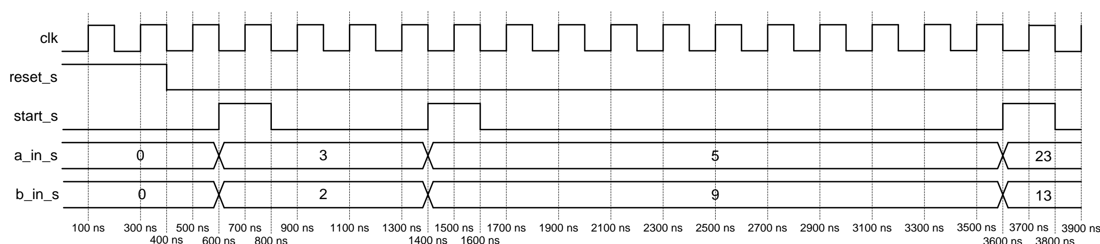
```
                start_s <= '1';

                -- Cekaj dok mnozac ne zavrsi sa mnozenjem
                loop
                   wait until falling_edge(clk_s);
                   start_s <= '0';
                   if (ready_s = '1') then
                      exit;
                   end if;
                end loop;
             end loop;
             wait;
          end process;
end architecture beh;
```

Prikazani testbenč instancionira instancu sekvencijalnog množača i koristeći *clk_gen* i *stim_gen* procese generiše povorku ulaznih signala koja može da se iskoristi za proveru ispravnog rada napisanog modela sekvencijalnog množača.

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

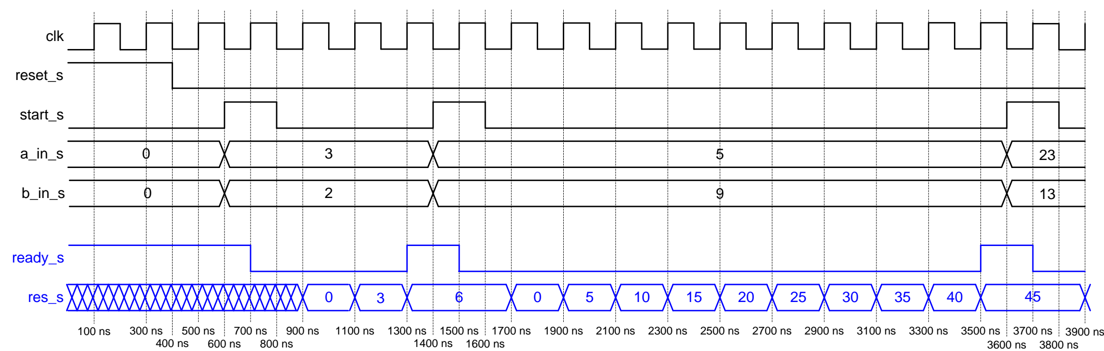
Stim_gen proces generiše vremenske oblike za signale *reset_s*, *start_s*, *a_in_s* i *b_in_s* koji su neophodni da se izvrši verifikacija rada sekvencijalnog množača. U okviru procesa deklarisane su dve promenljive, *a_v* i *b_v*, koje su zapravo nizovi od deset celobrojnih vrednosti. Ovi nizovi predstavljaju test vektore, posebno osmišljene vrednosti ulaznih signala pomoću kojih se može verifikovati korektan rad komponente koja se verifikuje. Nakon inicijalizacije sistema, stimulus generator dovodi na ulaze množača jedan po jedan test vektor. Kada se dovede sledeći test vektor na ulaze množača inicira se operacija množenja, postavljanjem signala *start_s* na jedinicu. Pošto vreme trajanja operacije množenja dva broja zavisi od njihovih vrednosti (jer je reč o sekvencijalnom množaču), stimulu generator mora da čeka dok sekvencijalni množač ne završi tekuću operaciju množenja. Ovaj trenutak će biti siglaniziran podizanjem *ready_s* signala na jedinicu od strane množača. Stimulus generator se vrti u petlji gde čeka na pojavu ovog događaja. Kada *ready_s* postane jedan, stimulus generator dovodi naredni test vektor na ulaze množača i inicira sledeću operaciju množenja. Ovaj postupak se ponavlja sve dok se ne iskoriste svi test vektori, u našem slučaju deset puta.

Talasni oblici koji će biti generisani pomoću ove dve *process* naredbe prikazani su na slici 13.



Slika 13. Generisani talasni oblici na ulaznim signalim sekvencijalnog množača

Nakon što se izvrši simulacija kombinovanog rada testbenča i modela sekvencijalnog množača, talasni oblici izlaznih signala trebalo bi da izgledaju kao na slici 14.

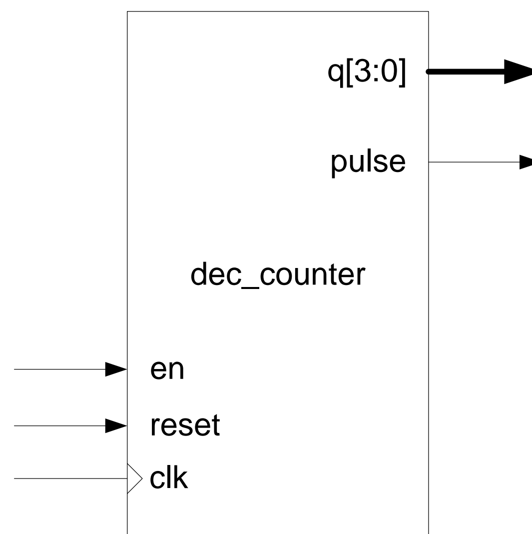


Slika 14. Generisani talasni oblici na ulaznim signalima sekvencijalnog množača zajedno sa talasnim oblicima izlaznih signala sekvencijalnog množača koji su dobijeni kao rezultat simulacije

Zadaci za vežbu

Zadatak 1:

Napisati VHDL model trocifrenog decimalnog brojača koji broji od 000 do 999, a zatim kreće od početka. Prilikom projektovanja sistema, primeniti tehniku hijerarhijskog dizajna, koristeći komponentu *dec_counter*. Komponenta *dec_counter* realizuje jednocifreni decimalni brojač koji broji od 0 do 9, a zatim kreće od početka. Interfejs komponente *dec_counter* prikazan je na Slici 15.



Slika 15. Interfejs komponente *dec_counter*

Ulazi *en* i *reset* su sinhroni kontrolni ulazi. Kada je *reset* aktivan, brojač se vraća u početno stanje i na izlazu *q* se nalazi vrednost „0000“, a izlaz *pulse* ima vrednost ’0’. Brojač broji na gore samo ako je kontrolni ulaz *en* aktivan. Trenutna vrednost brojača prosleđuje se ostatku sistema preko *q* izlaza. Funkcija izlaza *pulse* jeste da signalizira da je brojač stigao do maksimalne vrednosti, „1001“. U tom trenutku *pulse* ima vrednost ’1’, a u svim ostalim ima vrednost ’0’.

a) Nacrtati blok dijagram trocifrenog decimalnog brojača i označiti sve signale na njemu.

b) Na osnovu blok dijagrama napisati VHDL model.

c) Model pod b) realizovati korišćenjem *component* naredbe.

d) Model pod b) realizovati korišćenjem konfiguracije.

Za razvijene modele napisati testbenč pomoću kojega je moguće izvršiti njihovu verifikaciju.

Zadatak 2:

Množač dva neoznačena broja, baziran na sukcesivnom sabiranju, koji je bio razmatran u ovoj vežbi, iako baziran na jednostavnom algoritmu nije pogodan za praktičnu upotrebu jer je vreme potrebno za izračunavanje rezultata množenja reda $O(2^n)$, gde je n broj bita koji se koriste za predstavljanje vrednosti brojeva koji se množe. Na ovom mestu ćemo razmotriti drugi, efikasniji, algoritam množenja koji je baziran na metodi sabiranja i pomeranja („Add-and-Shift").

Množenje dva 4-bitna broja pomoću pomenutog metoda prikazano je u nastavku.

				a_3	a_2	a_1	a_0	Operand 1
X				b_3	b_2	b_1	b_0	Operand 2
				a_3b_0	a_2b_0	a_1b_0	a_0b_0	
			a_3b_1	a_2b_1	a_1b_1	a_0b_1		
		a_3b_2	a_2b_2	a_1b_2	a_0b_2			
+	a_3b_3	a_2b_3	a_1b_3	a_0b_3				
y_7	y_6	y_5	y_4	y_3	y_2	y_1	y_0	Proizvod

Množenje dva 4-bitna broja pomoću „Add-and-Shift" metode uključuje sledeće korake:

1. Pomnoži cifre Operanda 2 (b_3, b_2, b_1, b_0) sa Operandom 1 (A) jednu po jednu da bi se dobili sledeći proizvodi: b_3*A, b_2*A, b_1*A, b_0*A. Proizvod b_i*A se računa na sledeći način,

$$b_i*A = (a_3 \cdot b_i, a_2 \cdot b_i, a_1 \cdot b_i, a_0 \cdot b_i)$$

2. Pomeri proizvod b_i*A za i pozicije u levo.
3. Saberi pomerene članove b_i*A kako bi dobio konačni rezultat.

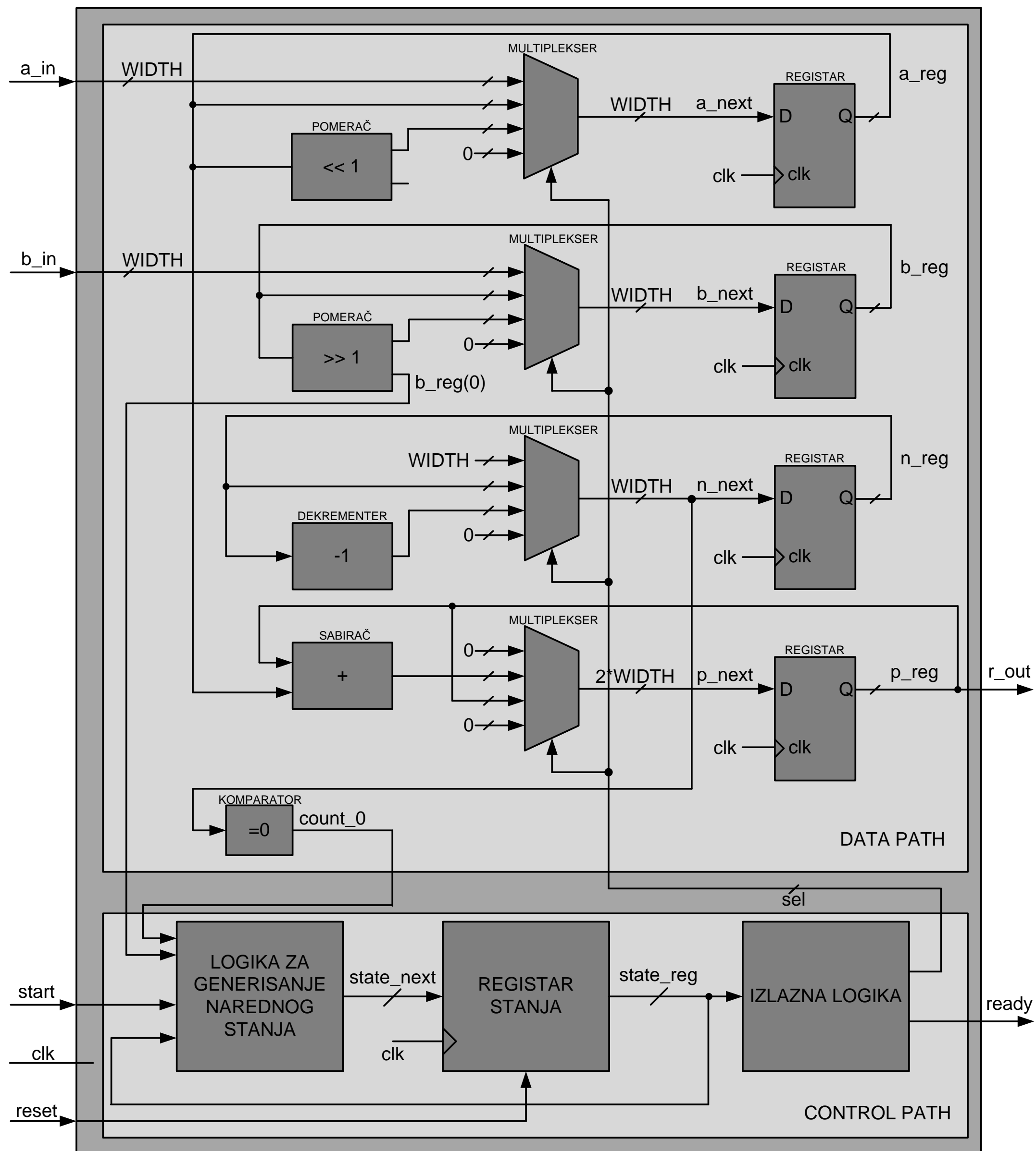
Opisana „Add-and-Shift" metoda lako se može prevesti u sekvencijalni algoritam. Možemo procesirati jednu cifru Operanda 2 (b_i) u jednom trenutku i ponoviti postupak za sve cifre Operanda 2 (B). U svakoj iteraciji, izračunaćemo po jedan proizvod b_i*A, pomeriti ga za i mesta u levo, i zatim ga dodati na tekuću vrednost proizvoda. Obzirom da je b_i zapravo binarna cifra, može imati samo dve vrednosti, 0 ili 1. Umesto da računamo proizvod b_i*A možemo iskoristiti *if* naredbu da proverimo vrednost cifre b_i i ukoliko je ona jednaka 1 dodati pomerenu vrednost Operanda 1 (A) na tekuću vrednost proizvoda. Pretpostavimo da su vrednosti dva broja koja je potrebno pomnožiti zadata pomoću m-bitnih ulaza *a_in* i *b_in*. Tada se „Add-and-Shift" metoda može prikazati u vidu algoritma prikazanog dole.

```
n = 0;
p = 0;
while (n != m)
{
        if (b_in(n) = 1) then
        {
                p = p + (a_in << n);
        }
        n = n + 1;
}
r_out = p;
```

Hardverska implementacija operacija indeksiranja (na primer, b_in(n)) i uopštenog pomeranja (na primer, a_in << n) je „skupa". Međutim, ove operacije se mogu izbeći ukoliko u svakoj iteraciji operande *a_in* i *b_in* pomerimo za jedno mesto u levo. Modifikovani „Add-and-Shift" algoritam prikazan je dole.

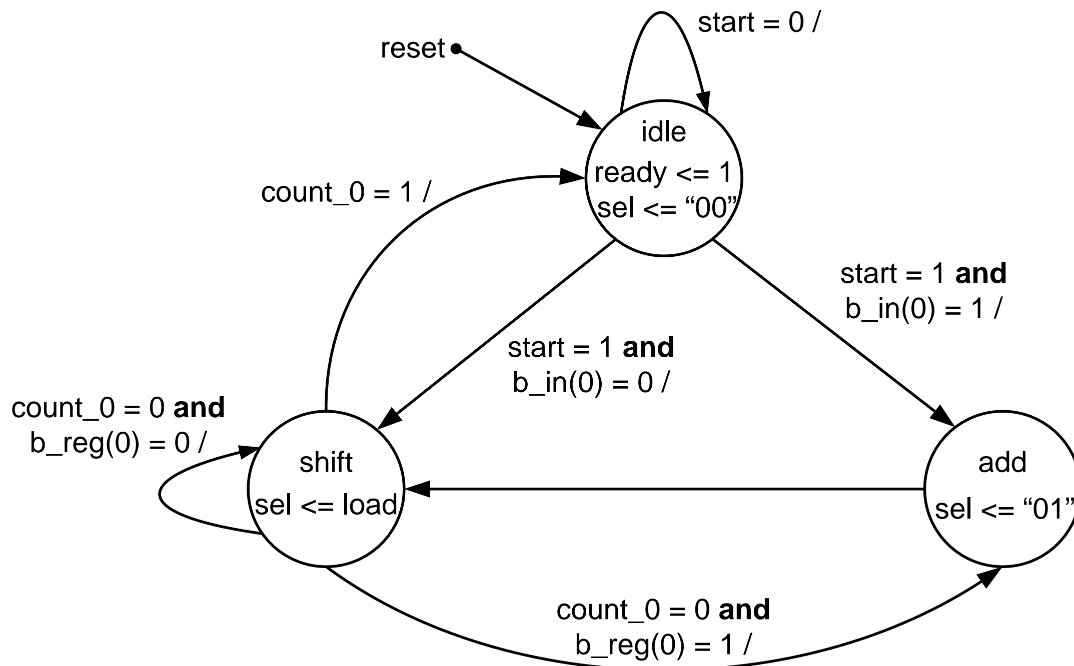
```
a = a_in;
b = b_in;
n = m;
p = 0;
while (n != 0)
{
        if (b(0) = 1) then
        {
                p = p + a;
        }
        a = a << 1;
        b = b >> 1;
        n = n - 1;
}
r_out = p;
```

Digitalni sistem koji implementira poslednju varijantu „Add-and-Shift" algoritma ima strukturu prikazanu na slici 16.



Slika 16. Blok dijagram digitalnog sistema koji implementira „Add-and-Shift“ algoritam

Dijagram stanja konačnog automata unutar upravljačke jedinice „Add-and-Shift“ množača prikazan je na slici 17.



Slika 17. Dijagram stanja upravljačke jedinice „Add-and-Shift“ množača

Koristeći blok dijagram digitalnog sistema sa slike 16 i dijagram stanja konačnog automata sa slike 17:

- Nacrtati hijerarhijsku strukturu „Add-and-Shift“ množača
- Na osnovu hijerarhijske strukture napisati potrebne VHDL modele koji čine kompletan model „Add-and-Shift“ množača
- Napisati testbenč pomoću kojega je moguće izvršiti verifikaciju „Add-and-Shift“ množača