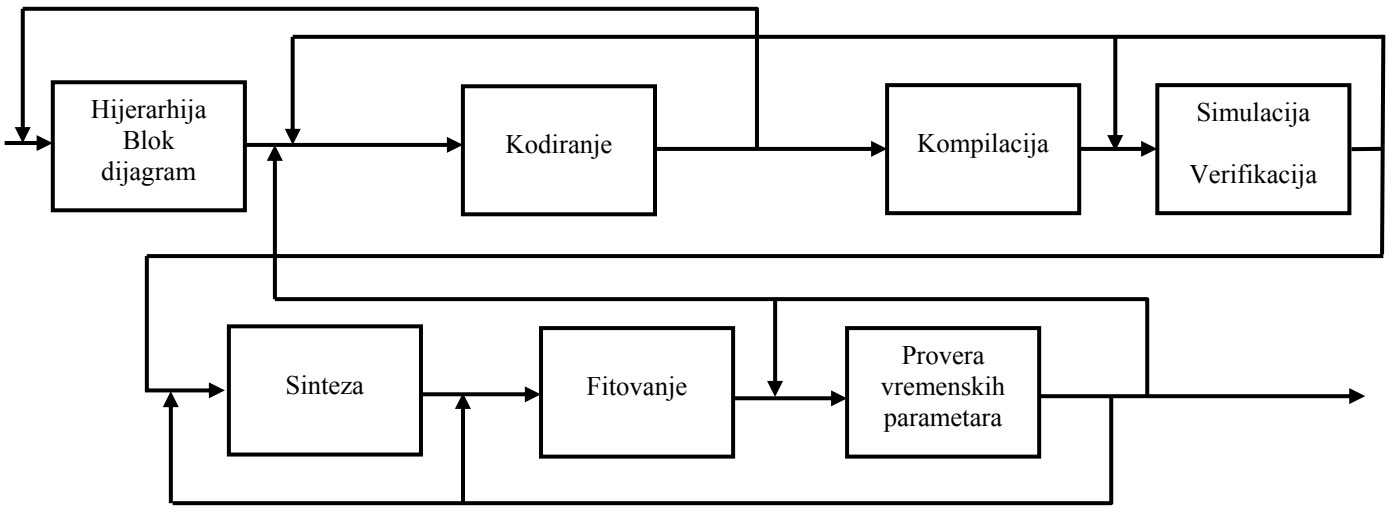
Sredinom 80tih godina U.S.Department of Defence i IEEE pokrenuli su razvoj jezika za opis hardvera sa velikim mogućnostima, VHDL-a (Very High Speed Integrated Circuits Hardware Description Language). Jezik je počeo sa sledećim karakteristikama:

- Dizajn može biti dekomponovan hijerarhijski
- Svaki element u dizajnu ima dobro definisan interfejs (za povezivanje sa ostalim elementima) i precizno definisano ponašanje (za simulaciju)
- Za specifikacije ponašanja mogu se uzeti ili algoritam ili hardverska struktura da definišu operacije elemenata
- Konkurentnost, vremenski parametri i takt mogu biti modelovani. VHDL podržava sinhrone i asinhrone strukture
- Logičke operacije se mogu simulirati

Tako je VHDL počeo kao jezik za modelovanje sistema i za njegovo dokumentovanje, pri čemu je ponašanje digitalnog sistema precizno specificirano i data je mogućnost simulacije.
Glavna prednost VHDL-a je u tome što se na osnovu napisanog koda, upotrebom komercijalnih VHDL alata za sintezu, mogu dobiti logičke šeme sistema direktno iz opisa ponašanja. Korišćenjem VHDL-a se mogu projektovati, simulirati i sintetizovati jednostavna kombinaciona kola, ali i složeni digitalni sistemi.

VHDL Dizajn

Postoji nekoliko koraka u VHDL dizajnu (slika 12.1).



Slika 12.1. Koraci u VHDL dizajnu

Hijerarhija (blok dijagram) podrazumeva izradu blokova na nivou blok dijagram strukture . Definišu se moduli i njihova povezanost.

Kodiranje. Pisanje VHDL koda za module u prethodnom koraku, njihovih interfejsa i internih detalja. Pošto je VHDL tekstualni jezik, može se koristiti bilo koji tekstualni editor. Većina okruženja za rad sa VHDL-om nudi specijalizovani VHDL tekst editor (uključuje automatsko izdvajanje ključnih reči i još neke pogodnosti).

Kompilacija. Kada se napiše kod potrebno ga je kompajlirati. VHDL kompajler analizira kod i traži sintaksne greške, proverava kompatibilnost sa ostalim modulima. Takođe kreira interne informacije koje su potrebne za simulaciju kasnije.

Simulacija. VHDL simulator dozvoljava korisniku da definiše i primeni ulaze na strukturu koju je definisao u VHDL-u, i da posmatra izlaze iz te strukture, bez potrebe za fizičkom realizacijom. Takođe, VHDL daje mogućnost kreiranja „test bench"–eva, posebne vrste VHDL koda, koji omogućavaju testiranje projektovane strukture bez korišćenja simulatora sa grafičkim editorom (koji su obično skuplji).

Verifikacija. Simulacija je samo jedan deo većeg procesa koji se zove verifikacija. Svrha simulacije je da se na osnovu zadatih vrednosti ulaznih signala proveri da li kolo radi onako kako je predviđeno. U velikim projektima bi takva provera bila komplikovana i trajala bi dugo, tako da se koriste postupci verifikacije dizajna.. Postoje dva nivoa verifikacije:

- Funkcionalna, kada se posmatra samo da li kolo zadovoljava funkcionalno zadate specifikacije, sva kašnjenja su nula
- Vremenska verifikacija, kada se posmatra funkcionisanje kola sa uključenim kašnjenjima (procenjenim)

  Vremenska verifikacija na ovom nivou je ograničena, jer vremenske karakteristike kola zavise od sinteze, tako da se detaljna vremenska verifikacija vrši posle sinteze.

Sinteza. Postupci i alati u ovom procesu su raznoliki i zavise od izabrane tehnologije. Mogu se izdvojiti tri osnovna koraka:

- **Sinteza.** Konvertovanje VHDL opisa u skup primitiva ili komponenata koje mogu biti ostvarene u izabranoj tehnologiji. Na primer, sa PLD ili CPLD komponentama alat za sintezu može generisati sume proizvoda.
- **Fitovanje.** Alatka za fitovanje mapira sintetizovane primitive ili komponente u raspoložive resurse komponente. Za PLD i CPLD je to dodeljivanje jednačina (sume proizvoda) raspoloživim AND-OR elementima. Dizajner često zadaje uslov fiteru u smislu raspodele modula u čipu, odnosno dodelu ulaznih i izlaznih pinova odgovarajućim signalima.
- **Vremenska verifikacija.** U ovom momentu je moguće precizno utvrditi vremenske parametre kola (kašnjenja u zavisnosti od dužina žica, električnog opterećenja itd.....). Obično se primenjuju isti test vektori kao u vreme funkcionalne verifikacije, ali u ovom slučaju oni se primenjuju na komponentu koja će se fizički realizovati.

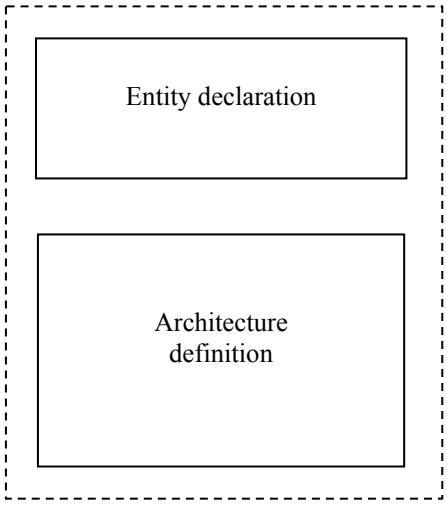
Na dijagramu toka VHDL projektovanja (slika 12.1), je pokazano da ako nisu zadovoljeni određeni zahtevi u dizajnu moraju se ponoviti neki koraci. Može se desiti da ako se ne zadovolje vremenski kriterijumi ceo dizajn pretrpi izmene (najnepovoljniji mogući slučaj).

Struktura VHDL koda

VHDL je zasnovan na principima strukturnog programiranja. Pozajmljene su ideje iz Pascala i Ade. Glavna ideja je definisati interfejs hardverskog modula, dok interni detalji nisu bitni. Svaki modul u okviru VHDL opisa (modul može biti elementarni ili složen, sastavljen iz više elementarnih modula) se sastoji iz dva dela:

1. Entity
2. Architecture

Entity je interfejs sistema. Sistem ima neke ulaze i izlaze preko kojih komunicira sa okolinom (ulazno-izlazni portovi). Dizajn uvek počinje od entitija.




```
entity Inhibit is
   port (X,Y: in   BIT;
         Z: out BIT);
end    Inhibit


architecture Inhibit_arch of Inhibit is
begin
    Z<='1' when X='1' and Y='0' else '0';
end  Inhibit_arch
```

Architecture. Sistem se projektuje da bi se izvršila neka transformacija od ulaza do izlaza . Ta transformacija se ostvaruje u unutrašnjim elementima modula i ona je definisana u okviru arhitekture VHDL koda.

VHDL definiše specijalne stringove koji se zovu rezervisane reči (keywords). U ovom primeru postoji nekoliko takvih reči {entity, port, is, in, out, end, architecture, begin, when, else, not}. Korisnički definisani identifikatori počinju slovom i sadrže slova, brojeve, donju crtu (underscore _). Donja crta ne sme pratiti drugu donju crtu niti biti poslednja.

Pravilno: Id_1 ;
Ident_1;
iDent_8;

Nepravilno: 1Id_1;
_Ident;
Ident_ _1;

U datom primeru to su identifikatori Inhibit, X, Z, BIT, Y i Inhibit_arch. "BIT" je ugrađeni identifikator za predefinisan tip. Ne smatra se rezervisanom reči, jer može biti redefinisan.

Bazična entity deklaracija ima sledeću sintaksu:

```
entity  entity-name is

 port (signal-names:mode signal-type;
       signal-names:mode signal-type;
       ……..
       signal-names:mode signal-type);

end  entity-name;
```

entity-name - Korisnički identifikator za naziv entitija

signal-names - Jedan ili više (razdvojenih zarezom) identifikatora za eksterne signale koji služe kao interfejs

mode - Definiše smer signala (**in** signal je ulazni, **out** signal j izlazni), vrednost takvog signala se može čitati unutar arhitekture pridružene datom entitiju. Može se koristiti samo od strane drugog entitija (ne od njegove arhitekture). Signal može biti još i **inout** tipa to jest biti i ulazni i izlazni.

signal-type - Ugrađeni odnosno korisnički definisani tip signala. Iza poslednjeg signal-type se ne stavlja ";".

Portovi entity-ja i njihovi modovi i tipovi su sve što se vidi od strane drugih modula koji komuniciraju sa datim modulom, Interne operacije entity-ja se specificiraju u okviru arhitekture.

Sintaksa arhitekture ima sledeću formu:

```
architecture architecture-name of entity-name is

    type          declarations
    signal        declarations
    constant      declarations
    function      definition
    procedure     definition
    component     declarations

begin
    concurrent-statement
    ……
    concurrent-statement
end  architecture-name;
```

architecture-name. Korisnički odabran identifikator, obično ima neku smisaonu vezu sa imenom entity-ja kojem se pridružuje data arhitektura.

entity-name. Isto ime koje nosi entity kojem se pridružuje data arhitektura.

Eksterni signali koji su interfejs datoj arhitekturi se nasleđuju iz port deklaracije entity-ja. Arhitektura može uključiti signale koji su interni, samo vidljivi u okviru njenog tela. Deklaracije i definicije se u okviru arhitekture mogu navesti bilo kojim redosledom.

signal declarations. Daje istu informaciju o signalima kao u slučaju entity-ja, osim što se ne specificira mod signala.

```
signal signal-names : signal-type;
```

U okviru arhitekture se može definisati više signala, a ne mora se definisati ni jedan signal. Obično signali odgovaraju nazivima internih linija u logičkom dijagramu sistema. Signali se mogu očitavati ili postavljati, dodeljivati im se vrednosti u okviru tela arhitekture.

variable. VHDL varijable su slične signalima sem što one nemaju fizičku interpretaciju u kolu. U opisu sintakse arhitekture ne postoji mesto za deklaraciju varijabli. Varijable se koriste u VHDL funkcijama, procedurama i procesima. U ovim elementima programa sintaksa deklaracije varijable bi bila:

```
variable variable -names : variable -type;
```

Tipovi i konstante

Svi signali, varijable i konstante u VHDL-u moraju imati pridruženi tip. Tip specificira skup ili opseg vrednosti koje objekat može uzimati. Takođe, postoji i set operatora pridruženih svakom tipu. VHDL ima nekoliko predefinisanih tipova.

Predefinisani tipovi su:

bit, bit-vector, boolean, character, integer, real, string, time

Najkorišćeniji tipovi su integer, character, boolean.

Integer je definisan u opsegu brojeva $-2^{31}+1$ do $2^{31}-1$

Boolean ima dve vrednosti **true** i **false**

Character sadrži sve karaktere u ISO 8 bitnom skupu karaktera, prvih 128 su ASCII karakteri. Ugrađeni operatori za integer i boolean tip su

Integer operator	**Značenje**
+	sabiranje
-	oduzimanje
*	množenje
/	deljenje
mod	celobrojno deljene
rem	ostatak cel. deljenja
abs	apsolutna vrednost

Boolean operator	**Značenje**
and	logičko i
or	logičko ili
nand	logičko ni
nor	logičko nili
xor	eks ili
xnor	eks nili
not	komplement

Najkorišćeniji tipovi podataka u VHDL-u su **user-defined** tipovi, a najbrojniji su **enumerated** tipovi (nabrojivi tipovi), koji se definišu listanjem svih vrednosti koje promenjiva tog tipa može imati. Tipovi boolean i character su takvi tipovi. Deklaracija tipa je sledeća:

```
type type-name is (value-list);
```

value-list je lista odvojena zarezom za sve moguće vrednosti tog tipa. Vrednosti mogu biti korisničko definisani identifikatori ili karakteri (gde je karakter ISO karakter pod apostrofom npr ‘x’). Nabrojivi tip se može koristiti da se opiše mašina stanja. Prvi slučaj sa identifikatorima je:

```
type traffic_light_state is (reset,stop,wait,go);
```

Drugi slučaj sa karakterima je primer **std_logic** tipa koji je deo IEEE 1164 standardnog paketa:

```
type STD_ULOGIC is (   'U', --neinicijalizovano
                       'X' , --nepoznato
                       '0', --nula
                       '1',--jedinica
                       'Z', --visoka impedansa
                       'W', --slabo nepoznata
                       'L', --slaba nula
                       'H', --slaba jedinica
                       '_', --nije bitno
                    );
```

```
subtype STD_LOGIC is resolved STD_ULOGIC;
```

VHDL dozvoljava korisnicima kreiranje podtipova (subtypes) datih tipova podataka.

```
subtype subtype-name is type-name start to end;

subtype subtype-name is type-name start downto end;
```

Vrednosti u podtipu moraju biti u kontinuiranom opsegu vrednosti osnovnog tipa, od **start** do **end.** Za nabrojive tipove **start** pokazuje prvi u nizu, **end** poslednji u nizu. Primeri:

```
subtype twoval_logic is std_logic range '0' to '1';

subtype fourval_logic is std_logic range 'X' to 'Z';

subtype negint is integer range -2147483647 to -1;

subtype bitnum is integer range 31 downto 0;
```

Redosled u opsegu može biti specificiran kao opadajući ili rastući, u zavisnosti od oznake **to** ili **downto**.

VHDL ima dva predefinisana integer podtipa:

```
subtype natural is integer range 0 to highest-integer;

subtype positive is integer range 1 to highest-integer;
```

Constants su konstante, pogodne zbog čitljivosti, održavanja i portabilnosti programa u bilo kom jeziku.

```
constant constant-name: type-name:= value;
```

Primeri:

constant BUS_SIZE: integer:= 32; -- širina magistrale

constant MSB: integer := BUS_SIZE-1; -- most significant bit

constant z: character:='z'; --sinonim za vrednost visoke impendanse

Konstanta se sme koristiti bilo gde se koristi i vrednost koju ona predstavlja.

Nizovi

Druga kategorija **user_defined** tipova su nizovi. Niz se definiše kao uređeni skup elemenata istog tipa, gde je svaki element pozicioniran indeksom u okviru niza.

```
type type-name is array (start to end) of element-type;

type type-name is array (start downto end) of element-type;

type type-name is array (range type) of element-type;

type type-name is array (range-type range start to end) of
element-type;

type type-name is array (range-type range start downto end) of
element-type;
```

U prva dva slučaja **start** i **end** su integeri koji određuju mogući opseg indeksa u okviru niza i stoga totalan broj elemenata niza. U poslednja tri slučaja, svi ili podskup vrednosti postojećeg tipa (range-type) je opseg vrednosti za indekse niza (indeksiraju se sa elementima range-type).

Primeri:

```
type monthly_count is array (1 to 12) of integer;

type byte is array (7 downto 0) of STD_LOGIC;

constant WORD_LEN: integer: =32;

type word is array (WORD_LEN-1 downto 0) of STD_LOGIC;

constant NUM_REGS: integer: =8;

type reg_file is array (1 to NUM_REGS) of word;

type statecount is array (traffic_light_state) of integer;
```

U trećem primeru je pokazano da element takođe može biti niz, prema tome definiše se dvodimenzioni niz. U poslednjem primeru niz ima četiri elementa, jer toliko imamo indeksa (indeksi su elementi nabrojivog tipa traffic_light_state, koji je definisan ranije). Smatra se da se elementi niza slažu sleva u desno u istom smeru kao i opseg indeksa. Tako da element u nizu koji je prvi sleva ima sledeće indekse:

monlty_count	1
Byte	7
word	31
reg_file	1
statecount	reset

U VHDL-u pojedinim elementima u okviru niza se pristupa uzimanjem imena niza i indeksa kao parametra. Ako su M, B, W, R i S signali ili varijable pet gore definisanih tipova, onda M(11), B(5), W(WORD_LEN-5), R(1,0), R(0) i S(reset) su korektno označeni elementi.

Dodela vrednosti nizovima se može izvršiti na sledeći način (primer niza tipa byte):

B:=(‘1’,’1’,’0’,’1’,’1’,’0’,’0’,’1’);

Takođe se može vršiti dodela elementima u okviru niza specificiranjem indeksa.

W:=(0=>’0’, 8=>’0’, 16=>’0’, 24=>’0’, others=>’1’);

Rezultat je:

W= 11111110111111101111111011111110

Moguće je pristupati podskupu niza specificiranjem početne i krajnje pozicije. M(6 to 9), B(3 downto 0), W(15 downto 8), R(1, 7 downto 0), R(1 to 2), S(stop to go). Smer u podskupu mora biti isti kao u originalnom nizu.

Funkcije i procedure

Funkcije su preslikavanja koja jedan skup preslikavaju u drugi. Imaju neke argumente i vraćaju rezultat. Svaki od argumenata i rezultat u VHDL funkcijskoj definiciji ili njenom pozivu ima odgovarajući tip. Posle imena funkcije, listaju se njeni formalni argumenti (nekoliko ili ni jedan), koji se koriste u telu funkcije. Kada se funkcija pozove, formalni argumenti u definiciji funkcije se zamenjuju realnim argumentima. VHDL ima strogo pravilo o saglasnosti tipova, tako da realni parametri moraju da odgovaraju formalnim parametrima (argumentima).

```
function function-name (
        signal-names: signal-type,
        signal-names: signal-type,
                .....
        signal-names: signal-type,
    ) return return-type is
            type declarations
          constant declarations
          variable declarations
          function declarations
         procedure declarations
begin
    sequential-statement
    .....
    sequential-statement
end function-name;
```

Kada se funkcija pozove iz neke arhitekture, vrednost koja je tipa **return-type** se vraća na mesto poziva funkcije. Funkcija može definisati svoje sopstvene tipove, konstante, varijable i ugneždene funkcije i procedure. Između **begin** i **end** se nalaze sekvencijalne linije koje se izvršavaju kada se funkcija pozove.

Primer funkcije:

```
architecture Inhibit_archf of Inhibit is
function  ButNot (A, B : bit) return bit is
begin
    if  B='0' then return A;
    else return '0';
    end if;
end  ButNot;

begin
  Z<= ButNot (X,Y);
end Inhibit_archf
```

return indicira da kada se funkcija pozove vraća ono što se nalazi iza return u okviru definicije funkcije. Tip rezultata mora odgovarati tipu u deklaraciji funkcije (return-type).

VHDL pruža mogućnost preklapanja operatora. Ovo omogućava korisniku da definiše neku funkciju korišćenjem ugrađenih operatorskih simbola (and, or, +.....) za odgovarajući tip operanda. Može postojati nekoliko definicija za određeni operatorski simbol. Kompajler automatski uzima onu definiciju koja odgovara tipu operanada koji se koriste na tom mestu.
U definiciji funkcije, apostrofi **"operator-name"** ukazuju da je operator preklopljen. Često je potrebno konvertovati jedan tip signala u drugi, IEEE paketi sadrže nekoliko funkcija za konverziju, npr iz bit u STD_LOGIC i obrnuto. Često je potrebno izvršiti konverziju iz STD_LOGIC_VECTOR u odgovarajući integer. IEEE 1164 paket ne sadrži ugrađenu funkciju za konverziju, već se one moraju definisati.

```
function CONV_INTEGER (x:STD_LOGIC_VECTOR) return integer is
      variable result: integer;
begin
    result:=0 ;
    for i in x'range loop
        result:= rsult*2;
        case x(i) is
          when '0'|'L'=>null;
          when '1'|'H'=>result:=result+1;
          when others =>null;
        end case;
    end loop;
    return result;
end CONV_INTEGER;
```

```
type STD_LOGIC_VECTOR is array (natural range <>) of STD_LOGIC;
```

STD_LOGIC_VECTOR definiše uređeni skup STD_LOGIC bita. U ovom slučaju opseg niza nije specificiran , sem da on mora biti subopseg određenog tipa, u ovom slučaju **natural**. Opseg se specificira kada se signal ili varijabla dodeljuju ovom tipu. U kodu je korišćena naredba null koja znači "ne radi ništa". Opseg za for petlju je specificiran sa **x'range**, pri čemu x' znači izdvajanje nekog atributa (svojstva) signala x, a range je ugrađeni identifikator za atribut koji se koristi samo za nizove i znači "opseg ovog niza od leva na desno".

Sa druge strane možemo konvertovati integer u STD_LOGIC_VECTOR. Ovde se mora specificirati još i broj bita u željenom rezultatu. Parametri su ARG-šta se konvertuje i SIZE-broj bita u koji se konvertuje.

```
function CONV_ STD_LOGIC_VECTOR (ARG:integer;SIZE:integer) return STD_LOGIC_VECTOR is
      variable result: STD_LOGIC_VECTOR (SIZE-1 downto 0);
      veriable temp : integer;
begin
    temp:=ARG ;
    for i in 0 to SIZE-1 loop
        if (temp mod 2)=1 then result (i):='1';
        else result (i):='0';
        end if;
        temp:=temp/2;
    end loop;
    return result;
end CONV_ STD_LOGIC_VECTOR;
```

Funkcija deklariše lokalnu varijablu **result** tipa STD_LOGIC_VECTOR, čiji opseg indeksa zavisi od SIZE, SIZE mora biti poznata kada se kompajlira funkcija CONV_ STD_LOGIC_VECTOR.

Procedure

Procedure su slične funkcijama, samo što ne vraćaju rezultat, retko se koriste i ovde neće biti razmatrane.

Biblioteke i paketi (packages)

VHDL biblioteka je mesto gde VHDL kompajler skladišti informacije oko projekta na kome se radi, uključujući i privremene fajlove koji se koriste u analizi, simulaciji i sintezi. Lokacija biblioteka na računaru je zavisna od implementacije, ponekad VHDL kompajler automatski kreira i koristi biblioteku **work**. VHDL dizajn se sastoji od mnogo fajlova, svaki sadrži odgovarajuću jedinicu dizajna (entity-je, architeture). Kada VHDL kompajler analizira svaki fajl u dizajnu, rezultate stavlja u direktorijum **work**, a takođe pretražuje ovu biblioteku za potrebne definicije, npr. drugih entity-ja. Zbog ove osobine veliki dizajni se mogu dekomponovati. Nekoliko projekata može koristiti istu biblioteku. Svaki projekat ima svoj **work**, ali takođe mora ukazivati i na zajedničku biblioteku koja sadrži deljene informacije. Čak i mali projekti mogu koristiti standardne biblioteke kao npr. onu koja sadrži IEEE standard definicije. U dizajnu se može specificirati ta biblioteka upotrebom **library** klauzule na početku fajla. Npr. **library ieee;**

Klauzula **library work** je implicitno uključena već na početku svakog projekta. Specificiranje biblioteka daje pristup do analiziranih Entity-ja i arhitektura postavljenih u biblioteci, ali ne daje pristup do definicija tipova. To je funkcija paketa (packages).

VHDL package je fajl koji sadrži definicije objekata koji mogu biti korišćeni u drugim VHDL fajlovima.

Vrsta objekata koji mogu biti uključeni u paketu su:

- Signali
- Tipovi
- Konstante
- Funkcije
- Procedure
- Deklaracije komponenata

Signali koji se definišu u paketu su globalni signali, dostupni svakom VHDL entity-ju koji koristi taj paket. Tipovi i konstante u paketu su dostupni bilo kom fajlu koji koristi taj paket. Funkcije i procedure definisane u paketu se mogu koristiti (pozivati) u fajlovima koji koriste taj paket. Komponente se mogu inicijalizovati u arhitekturama koje koriste taj paket.

Dizajn može koristiti paket posredstvom klauzule **use** na početku dizajna, na primer:

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

Ovde je **ieee** ime biblioteke koje je prethodno dato u **library** klauzuli. U ovoj biblioteci fajl pod imenom **std_logic_1164** sadrži željene definicije. Sufiks **all** govori kompajleru da upotrebi sve definicije iz fajla. Umesto **all** možemo pisati ime određenog objekta da bismo iskoristili samo njegovu definiciju. Na primer:

```
use ieee.std_logic_1164.std_ulogic;
```

Bilo ko može definisati pakete koristeći se sledećom sintaksom:

```
package package-name is
   type declarations
   signal declarations
   constant declarations
   component declarations
   function declarations
   procedure declarations
end package-name;
package body package-name is
   type declarations
   constant declarations
   function definitions
   procedure definitions
end   package-name;
```

Svi objekti deklarisani između **package** i prvog **end** su vidljivi u okviru bilo kog fajla dizajna koji koristi taj paket. Objekti koji stoje iza **package body** su lokalni. Prvi deo uključuje deklaraciju funkcije i procedure, ne definicije. Deklaracije funkcija listaju samo ime funkcije, argumente i tipove ali ne uključuju **is** reč kao u definiciji. Kompletna definicija funkcije je data u telu paketa i nije vidljiva za korisnike funkcije.

Elementi strukturnog dizajna

U okviru arhitekture u VHDL dizajnu, svaka konkurentna linija (concurrent statement) se izvršava simultano sa ostalim konkurentnim linijama u okviru istog tela arhitekture. Ovo ponašanje se razlikuje u odnosu na ostale komercijalne programske jezike, gde se linije izvršavaju sekvencijalno. Konkurente linije su neophodne da bi se simuliralo ponašanje hardvera, gde povezani elementi utiču jedan na drugi stalno, ne samo u određenom vremenu i određenim redosledom. Tako npr. ako poslednja linija dodeli vrednost signalu koji se koristi u prvoj liniji, tada će se simulator vratiti na tu prvu liniju i korigovati njene rezultate u skladu sa signalom koji se upravo promenio.

Osnovna konkurentna linija je **component statement** čija je sintaksa:

```
label: component-name port map (signal1, signal2,....,signaln);

label: component-name port map (port1=>signal1,
       port2=>signal2,....,portn=>signaln);
```

component-name je ime prethodno definisanog entity-ja koji će se koristiti, ili instancirati u okviru tela arhitekture. Jedna instanca imenovanog entity-ja se kreira za svaku navedenu komponentu koja poziva taj entity i svaka instanca mora biti imenovana jedinstvenom labelom.

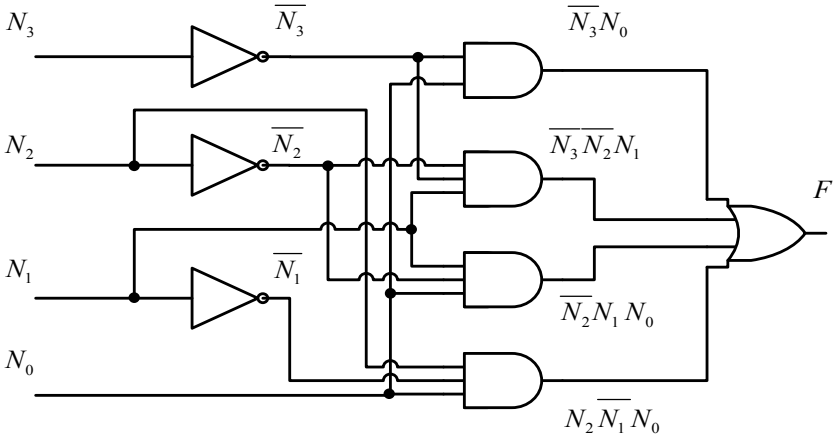
Port map reč uvodi listu pridruženih portova imenovanog entity-ja sa signalima u tekućoj arhitekturi. Lista može biti napisana u dva prikazana stila. Prvi je pozicioni, signali u listi su dodeljeni portovima entity-ja onim redosledom kojim se pojavljuju u definiciji entity-ja. Drugi stil je eksplicitan, svaki port entity-ja je vezan za signal vezom => operatora, i ovo dodeljivanje može biti u bilo kom redosledu.

Pre nego što se instancira u arhitekturi, komponenta mora biti navedena u deklarativnom delu te arhitekture, kako je ranije opisano.

```
component  component-name
      port (signal-names: mode signal-type;
            signal-names: mode signal-type;
            .....
            signal-names: mode signal-type);
end component;
```

Komponente koje se koriste u arhitekturi mogu biti one koje su prethodno definisane kao deo dizajna, ili mogu biti deo biblioteke.

Primer. **Detektor prostih brojeva**. Strukturni opis odgovara sledećoj šemi:



Šema detektora prostih brojeva

```
library IEEE;
use IEEE.std_logic_1164.all;
entity prime is
   port (N : in STD_LOGIC_VECTOR (3 downto 0);
         F : out STD_LOGIC);
end prime;
architecture prime1_arch of prime is
  signal N3_L, N2_L, N1_L: STD_LOGIC;
  signal  N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
  component INV port ( I : in STD_LOGIC; O : out STD_LOGIC);
  end component;
  component AND2 port (I0, I1 : in STD_LOGIC; O : out STD_LOGIC);
  end component;
  component AND3 port (I0, I1,I2 : in STD_LOGIC; O : out STD_LOGIC);
  end component;
  component OR4 port (I0, I1,I2,I3 : in STD_LOGIC; O : out STD_LOGIC);
  end component;
 begin
  U1 : INV    port map ( N(3), N3_L);
  U2 : INV    port map ( N(2), N2_L);
  U3 : INV    port map ( N(1), N1_L);
  U4 : AND2 port map ( N3_L, N(0), N3L_N0);
  U5 : AND3 port map ( N3_L, N2_L, N(1), N3L_N2L_N1);
  U6 : AND3 port map ( N2_L,  N(1), N(0), N2L_N1_N0);
  U7 : AND3 port map ( N(2),  N1_L, N(0), N2_N1L_N0);
  U8 : OR4    port map (N3L_N0,  N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0, F);
end prime1_arch;
```

U entity-ju su deklarisani ulazni i izlazni signali sistema. U deklarativnom delu arhitekture su deklarisani svi interni signali i komponente koje će se koristiti (komponente INV, AND2, AND3, OR4 su definisane u okruženju dizajna). Linije u kojima su instancirane komponenate u arhitekturi se izvršavaju konkurentno. Čak da su linije navedene u drugom redosledu, sintetizovalo bi se isto kolo, i simulirana operacija kola bi bila ista. VHDL arhitektura koja koristi komponente se često zove **strukturan opis** ili **strukturan dizajn,** jer definiše strukturu povezivanja signala i entiteta koji čine taj dizajn. Ovakav opis je ekvivalentan šemi ili net listi u električnom kolu.

U nekim aplikacijama je neophodno kreirati više kopija određene strukture u okviru arhitekture. Npr. potrebno je napraviti neku kaskadu ćelija. VHDL poseduje **generate** direktivu koja dozvoljava kreiranje takvih struktura koje se ponavljaju upotrebom neke vrste for petlje, bez potrebe za pisanjem svih instanci komponente (strukture) koja se ponavlja.
Sintaksa jednostavne iterativne **generate** petlje je:

```
label : for identifier in range generate
            concurrent-statement
        end generate;
```

Indentifier je implicitno deklarisan kao varijabla sa tipom kompatibilnim kao **range.** Konkurentna linija se izvrši jednom za svaku moguću vrednost identifier-a u okviru opsega.

Primer 8-bitnog invertora.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity inv8 is
     port ( X: in STD_LOGIC_VECTOR (1 to 8);
            Y: out STD_LOGIC_VECTOR (1 to 8);
end inv8;

architecture inv8_arch of inv8 is

component INV port (I: in STD_LOGIC; O: out STD_LOGIC);
end component;

begin
  g1: for b in 1 to 8 generate
         U1: INV port map (X(b), Y(b));
      end generate;
end  inv8_arch;
```

Vrednosti konstanti se moraju znati u vreme kada se VHDL program kompajlira. U mnogim aplikacijama je korisno dizajnirati i kompajlirati entity i njegovu arhitekturu ostavljajući neke od njihovih parametara, kao npr. širinu magistrale, nespecificiranom. U VHDL-u je to moguće korišćenjem **generic** parametara. Jedan ili više generičkih konstanti se mogu definisati u deklaraciji entity-ja u delu **generic declaration**, sa sledećom sintaksom:

```
entity entity-name is
  generic (constant-names: constant-type;
           .....
           constant-names: constant-type);
  port     (signal-names: mode signal-type;
           .....
           signal-names: mode signal-type;
end      entity-name;
```

Svaka od imenovanih konstanti se može koristiti u okviru definicije arhitekture koja se odnosi na taj entity. Konstanta dobija vrednost tek kada se taj entity instancira upotrebom **component** klauzule u okviru neke druge arhitekture. U okviru **component** klauzule, vrednosti se pridružuju generičkim konstantama upotrebom **generic map** klauzule na isti način kao **port map.** U sledećem primeru se kombinuju **generic** i **generate** klauzule da se definiše invertor magistrale sa širinom koju specificira korisnik.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity businv is
     generic (WIDTH: positive);
     port ( X: in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
            Y: out STD_LOGIC_VECTOR (WIDTH-1 downto 0);
end businv;

architecture businv _arch of businv is

component INV port (I: in STD_LOGIC; O: out STD_LOGIC);
end component;

begin
  g1: for b in WIDTH-1 downto 0 generate
         U1: INV port map (X(b), Y(b));
      end generate;
end  businv_arch;
```

Korišćenje ove komponente dato je u narednom primeru. Postoje 3 ovakve komponente za 8, 16, 32 bitnu magistralu.

```
library IEEE;
use IEEE.std_logic_1164.all;
entity businv_example is
port ( IN8: in STD_LOGIC_VECTOR (7 downto 0);
OUT8: out STD_LOGIC_VECTOR (7 downto 0);
IN16: in STD_LOGIC_VECTOR (15 downto 0);
OUT16: out STD_LOGIC_VECTOR (15 downto 0);
IN32: in STD_LOGIC_VECTOR (31 downto 0);
OUT32: out STD_LOGIC_VECTOR (31 downto 0);
end businv_example;
architecture businv _ex_arch of businv_example is
component businv
generic (WIDTH: positive);
port (X: in STD_LOGIC_VECTOR (WIDTH-1 downto 0);
Y: out STD_LOGIC_VECTOR (WIDTH-1 downto 0));
end component;
begin
U1: businv generic map (WIDTH=>8) port map (IN8, OUT8);
U2: businv generic map (WIDTH=>16) port map (IN16, OUT16);
U3: businv generic map (WIDTH=>32) port map (IN32, OUT32);
end  businv_ex_arch;
```

Dataflow dizajn

VHDL podržava neke konkurentne naredbe koje dozvoljavaju da se kolo opiše u duhu kretanja podataka i operacija nad njima. To je **dataflow** opis ili **dataflow** dizajn.

Konkurentna **signal-assignement** naredba se koristi za konkurentno dodeljivanje vrednosti signalu. Tip izraza koji se dodeljuje signalu mora da odgovara tipu signala (može se desiti da je izraz istog tipa ili podtip, a u slučaju nizova, i tip i dužina moraju odgovarati).

```
signal-name <=expression;
signal-name <= expression when boolean-expression else
               .....
               expression when boolean-expression else
               expression;
```

Primer arhitekture detektora prostih brojeva napisane u stilu **dataflow** dizajna.

```
architecture prime2_arch of prime is
  signal  N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
begin
  N3L_N0         <= not N(3)                                  and N(0);
  N3L_N2L_N1  <= not N(3)   and not N(2) and      N(1)            ;
  N2L_N1_N0   <=                  not N(2) and      N(1)  and N(0);
  N2_N1L_N0   <=                      N(2) and not N(1)  and N(0);
  F <= N3L_N0  or N3L_N2L_N1 or N2L_N1_N0 or  N2_N1L_N0;
end prime2_arch;
```

Ovde ne prikazujemo eksplicitno gejtove i njihovu međusobnu povezanost, već koristimo ugrađene VHDL operatore and, or, not. Operator **not** ima najveći prioritet pa nisu potrebne zagrade kod **and not N(1).**

Možemo koristiti i drugu, kondicionalnu formu konkurentne **signal-assignment** naredbe upotrebom **when-else** konstrukcije.

```
architecture prime3_arch of prime is
  signal  N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
begin
  N3L_N0         <='1' when  N(3)='0' and N(0)='1' else '0';
  N3L_N2L_N1  <='1' when N(3)='0' and N(2)='0' and N(1)='1' else '0';
  N2L_N1_N0   <='1'  when N(2)='0' and N(1)='1' and N(0)='1' else '0';
  N2_N1L_N0   <= '1' when N(2)='1' and N(1)='0' and N(0)='1' else '0';
  F <= N3L_N0  or N3L_N2L_N1 or N2L_N1_N0 or  N2_N1L_N0;
end prime3_arch;
```

Konkurentna **selected signal assignement** sintaksa je:

```
with expression select
  signal-name <= signal-value when choices,
                 signal-value when choices,
                 ..............................
                 signal-value when choices;
```

Ova struktura izračunava dati izraz (expression), i kad vrednost postane jednaka nekoj od ponuđenih vrednosti za selekciju (choices) dodeljuje se odgovarajući signal-value tom signalu. Selekcije u svakoj when klauzuli mogu biti usamljeni izrazi ili lista vrednosti odvojena vertikalnim linijama. Selekcije moraju biti međusobno isključivi i uključene sve moguće. Reč **others** se može upotrebiti u poslednjoj **when** klauzuli, da označi sve vrednosti izraza koje nisu bile pokrivene.

Primer detektora prostih brojeva.

```
architecture prime4_arch of prime is
begin
  with N select
    F<= '1' when "0001",
        '1' when "0010",
        '1' when "0011"|"0101"|"0111",
        '1' when "1011"|"1101",
        '0' when  others;
end prime4_arch;
```

Svi uslovi kada F ima vrednost **1** su mogli biti u jednoj **when** liniji, iz čisto edukativnih razloga je ovako raščlanjeno. Prethodna arhitektura se mogla modifikovati upotrebom CONV_INTEGER funkcije koju smo prethodno definisali.

```
architecture prime5_arch of prime is
begin
  with CONV_INTEGER (N) select
    F<= '1' when 1|2|3|5|7|11|13,
        '0' when  others;
end prime5_arch;
```

Dizajn na nivou opisa ponašanja sistema

Jedna od glavnih prednosti VHDL-a je mogućnost kreiranja dizajna baziranog na opisu ponašanja sistema. Takav način predstavljanja sistema je mnogo bliži projektantu jer ne zahteva dodatnu transformaciju podataka o ponašanju sistema, koji se obično dostavljaju kao ulazni parametri u procesu dizajna. Osnovni element u VHDL opisu ponašanja sistema je **proces**. **Proces** je kolekcija sekvencijalnih koraka, a izvršava se konkurentno (paralelno) sa ostalim konkurentnim koracima i ostalim procesima.
Upotrebom procesa mogu se specificirati kompleksne interakcije signala i događaja onako kako bi se one odvijale u simulaciji sistema (bez vremenske dimenzije). VHDL **process statement** se može koristiti bilo gde na mestu konkurentne linije. Sintaksa procesa je:

```
process (signal-name, signal-name,....., signal-name)
  type   declarations
  variable declarations
  constant declarations
  function definitions
  procedure definitions
begin
 sequential-statement
 .............
 sequential-statement
end process;
```

Obzirom da se proces definiše u oblasti neke arhitekture, proces vidi tipove, signale, konstante, funkcije, procedure koje su deklarisane ili su vidljive u okviru te arhitekture. Takođe, u procesu se mogu definisati tipovi, varijable, konstante, funkcije i procedure da budu lokalni za proces. Proces ne može deklarisati signale, samo varijable. VHDL varijabla čuva informaciju o stanju u okviru procesa i nije vidljiva izvan procesa. Sintaksa definisanja varijable u okviru procesa je

```
variable variable-names: variable-type;
```

VHDL proces se ili nalazi u stanju izvršavanja (aktivan) ili je suspendovan. Lista signala u okviru definicije procesa (sensitivity list), određuje kada se proces izvršava. Inicijalno je proces suspendovan, a kada bilo koji signal iz sensitivity list-e promeni vrednost, proces se izvršava polazeći od prve sekvencijalne linije i nastavlja sve do poslednje. Ako neki signal iz liste osetljivosti u toku izvršavanja procesa promeni vrednost, proces se po poslednjoj liniji opet izvršava. Ovo se nastavlja sve dok ni jedan signal ne promeni vrednost. Korektno napisan proces će se uvek nekad zaustaviti. Nekorektan se nikad ne zaustavlja.
Primer:

```
X<=not X;  sa listom osetljivosti (X)
```

Ovaj proces se stalno izvršava. U praksi postoje ograničenja za ovo neželjeno ponašanje tako što se proces prekida posle npr 1000 izvršenja.

Lista osetljivosti je opciona, proces bez liste osetljivosti počinje sa izvršavanjem u t=0 u okviru simulacije.

VHDL ima nekoliko vrsta sekvencijalnih linija. Prva je **sequential signal-assignment statement**, sintaksa joj je ista kao i konkurentnoj liniji **signal-name<=expression** s tim što se ona dešava u okviru tela procesa a ne arhitekture. Analogno za varijable postoji **variable-assignment statement** sa sintaksom **variable-name:=expression;** (:= umesto <=).

Primer detektora prostih brojeva.

```
architecture prime6_arch of prime is
begin
 process(N)
    variable  N3L_N0, N3L_N2L_N1, N2L_N1_N0, N2_N1L_N0: STD_LOGIC;
 begin
   N3L_N0          := not N(3)                                  and N(0);
   N3L_N2L_N1  := not N(3)   and not N(2) and      N(1)              ;
   N2L_N1_N0    :=                  not N(2) and      N(1)  and N(0);
   N2_N1L_N0    :=                       N(2) and not N(1)  and N(0);
   F <= N3L_N0  or N3L_N2L_N1 or N2L_N1_N0 or  N2_N1L_N0;
 end process;
end prime6_arch;
```

Ovde u okviru arhitekture (prime6_arch) imamo samo jednu konkurentnu liniju koja je u stvari sam proces. Lista osetljivosti sadrži samo **N,** izlazi logičkih kola moraju biti definisani kao varijable obzirom da definicija signala nije dozvoljena.

IF STATEMENT sintaksa

```
if boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
else sequential-statement
end if;

if boolean-expression then sequential-statement
elseif  boolean-expression then sequential-statement
…….
elseif  boolean-expression then sequential-statement
end if;

if boolean-expression then sequential-statement
elseif  boolean-expression then sequential-statement
…….
elseif  boolean-expression then sequential-statement
else sequential-statement
end if;
```

U VHDL opisu ponašanja nekog sistema često se koriste **if-then-else** strukture. Za kreiranje ugneždenih **if-then-else** struktura VHDL koristi specijalnu reč **elsif**. Sekvencijalna linija se izvršava ako boolean izraz ima vrednost **true**, i svi boolean izrazi koji prethode imaju vrednost **false**. Opciona poslednja **else** sekvencijalna linija se izvršava ako su svi prethodni boolean izrazi bili false.

Primer detektora prostih brojeva

```
architecture prime7_arch of prime is
begin
  process(N)
     variable  NI: integer;
  begin
      NI:=CONV_INTEGER(N);
       If NI=1 or NI=2 then F<='1';
       elsif NI=3 or NI=5 or NI=7 or NI=11 or NI=13 then F<='1';
       else F<='0';
       end if;
  end process;
end prime7_arch;
```

NI lokalna promenjiva sadrži konvertovanu vrednost ulaznog signala N (4 bita). Bulovi izrazi se ne preklapaju pa je samo jedan uslov ispunjen.

CASE STATMENT sintaksa

```
case expression is
   when choices => sequential-statement
   ……….
   when choices => sequential-statement
end case;
```

Case naredba se koristi kada vrednost jednog signala ili izraza određuje nekoliko različitih sekvencijalnih koraka koje treba izvršiti. Prvo se izračunava **expression**, nalazi se vrednost u **choices** koja se poklapa sa izračunatom vrednosti i izvrši odgovarajuća sekvencijalna linija. Za svaku **choices** može postojati jedna ili više sekvencijalnih linija. **Choices** mogu biti u formi **single** ili **multiple** vrednosti odvojene vertikalnim linijama. Izbori moraju biti međusobno isključivi i obuhvatiti sve moguće vrednosti izraza. Izraz **others** može stajati umesto poslednjeg **choices** da odredi sve one slučajeve koji nisu prethodno bili obuhvaćeni.

Primer detektora prostih brojeva

```
architecture prime8_arch of prime is
begin
  process(N)
  begin
       case CONV_INTEGER(N) is
          when 1 => F<='1';
          when 2 => F<='1';
          when 3|5|7|11|13 => F<='1';
          when others => F<='0';
       end case;
  end process;
end prime8_arch;
```

LOOP STATEMENT sintaksa

```
loop
   sequential-statement
  ……….
   sequential-statement
end loop;
```

Sledeću važnu sekvencijalnu naredbu predstavlja naredba loop. U gornjem primeru je prikazana sintaksa za loop petlju koja je beskonačna, sa sekvencijalnim linijama u okviru petlje. Retko se koristi beskonačna petlja, korisna je za hardversko modelovanje.

FOR LOOP sintaksa

```
for identifier in range loop
   sequential-statement
  ……….
   sequential-statement
end loop;
```

Promenjiva **identifier** je deklarisana implicitno i ima isti tip kao i **range.** Ova promenjiva se može koristiti u okviru sekvencijalnih linija, i ona prolazi kroz sve vrednosti u okviru **range**-a od leva na desno, jedanput u toku iteracije.

Dve korisne sekvencijalne linije koje mogu biti u okviru petlje su **exit** i **next**. **Exit** prebacuje kontrolu na liniju koja se nalazi odmah posle loop end. **Next** uzrokuje da se preskoče preostale sekvencijalne linije i da se uđe u narednu iteraciju (sledeće izvršavanje petlje).
Primer detektora prostih brojeva

```
library IEEE;
use IEEE.std_logic_1164.all;
entity prime9 is
   port (N : in STD_LOGIC_VECTOR (15 downto 0);
         F : out STD_LOGIC);
end prime9;
architecture prime9_arch of prime9 is
begin
  process (N)
  variable NI:integer ;
  variable prime: boolean;
  begin
    NI:=CONV_INTEGER(N);
    prime:=true;
    if NI=1 or NI=2 then null;
    else for i in  2 to 253 loop
            if NI mod i =0 then
             prime:=false;exit;
            end if;
         end loop;
     end if;
     if prime then F<='1'; else F<='0'; end if;
   end process;
end prime9_arch;
```

WHILE LOOP sintaksa

```
while boolean-expression loop
   sequential-statement
  ……….
   sequential-statement
end loop;
```

Pre svake iteracije se testira bulov izraz i petlja se izvršava samo ako je bulov izraz true.

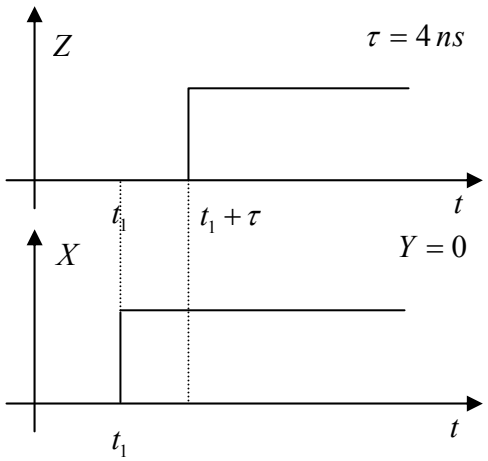
Simulacija i vreme

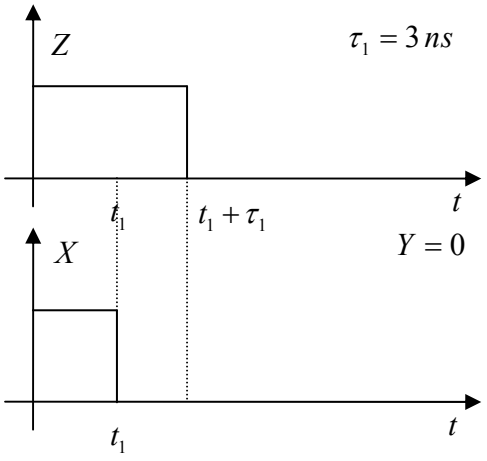
U svim do sada navedenim primerima vrme je bilo izostavljeno kao parametar, podrazumevalo se da se sve operacije trenutno izvršavaju. Međutim, VHDL ima dobre mogućnosti za modelovanje ponašanja sistema u vremenu, pa se može uzeti da je i to jedna od prednosti jezika.

VHDL dozvoljava specificiranje vremenskog kašnjenja upotrebom **after** službene reči u bilo kojoj liniji gde se dodeljuje vrednost signalu. Primer:

```
Z<='1' after 4ns when X='1' and Y='0'
else '0' after 3ns;
```

Ovo nam omogućuje da modelujemo logičko kolo sa fizičkim parametrima u vremenskoj dimenziji.

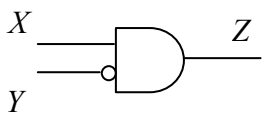




Drugi način uvođena vremenske dimenzije je upotreba službene reči **wait**. Ona se može koristiti da suspenduje proces na neko određeno vreme.

Važno je naglasiti da se pomenute naredbe koriste uglavnom za pisanje test bench-a, posebne vrste VHDL koda koji služi za proveru rada neke komponente, i da se na osnovu takvog koda ne može izvršiti logička sinteza.

Primer: Kreiranje talasnog oblika (signala) za četiri različite ulazne kombinacije u intervalima od 10ns u test bench-u za proveru rada dvoulazničog gejta (komponenta Inhibit).



```
entity InhibitTestBench is
end  InhibitTestBench;
acrhitecture InhibitTB_arch of InhibitTestBench is
component Inhibit port (X,Y : in BIT; Z: out BIT); end component;
signal XT, YT, ZT: BIT;
begin
  U1 : Inhibit port map (XT, YT, ZT);
  process
    begin
    XT<='0'; YT<='0';
    wait for 10ns;
    XT<='0'; YT<='1';
    wait for 10ns;
    XT<='1'; YT<='0';
    wait for 10ns;
    XT<='1'; YT<='1';
    wait; -- suspenduje proces na neodređeno
   end process;
end InhibitTB_arch;
```

Simulacija počinje u trenutku t=0, u to vreme simulator inicijalizuje sve signale na **default** vrednosti (koje su nezavisne od nas), takođe inicijalizuje one signale i promenljive koje su eksplicitno navedene da dobijaju vrednost. Sledeće što se dešava je početak izvršavanja svih procesa i konkurentnih linija. U početnom trenutku svi procesi su uzeti u razmatranje za izvršavanje, i jedan od njih se selektuje. Sve njegove sekvencijalne linije se izvršavaju, uključujući i petlje. Kada se završi izvršavanje ovog procesa, drugi proces se selektuje itd....sve dok se svi procesi ne završe. Ovim se kompletira jedan simulacioni ciklus.

Za vreme svog izvršavanja, proces može dodeliti nove vrednosti signalima. Međutim, nove vrednosti signala se ne dodeljuju momentalno, nego se smeštaju u listu događaja i nadgledaju da postanu efektivne u određeno vreme. Ako je navedeno kad se dodeljuje vrednost signalu (pomoću **delay**, ili **after**) onda se nadgleda lista događaja i u naznačenom vremenu izvrši dodela vrednosti signalima. Ako to nije navedeno, dodela se vrši "odmah" po završetku simulacionog ciklusa (mora da postoji "δ kašnjenje" koje je definisano u simulatoru). Kada se završi ciklus simulacije, lista događaja se skenira, gleda se koji su se signali menjali, donosi se odluka o novim vrednostima signala (u slučaju da se vrednost jednog signala menjala u nekoliko procesa u tom ciklusu) i nakon δ vremena se dodeljuju nove vrednosti tim signalima. Neki od tek izvršenih procesa mogu biti osetljivi na promenu tih signala, i svi oni se uzimaju u obzir za izvršavanje u narednom ciklusu simulacije koji počinje odmah.

Ovakav **(event-list)** mehanizam omogućuje simulaciju konkurentskih procesa bez obzira što se simulator izvršava na jednom procesoru u jednoj niti. Na taj način se dobija simulacija koja odgovara realnom radu hardvera gde je uobičajeno da više uređaja radi istovremeno.

Aldec Active-HDL simulacija

Za većinu primera koji slede je data i vremenska simulacija koja je urađena u Aldecovom Active-HDL-u (verzija 6.3). U nastavku će ukratko biti opisani koraci u procesu formiranja novog projekta, kompilacije, odnosno vremenske simulacije hardvera koji se opisuje.

Formiranje novog projekta

Selektovanjem opcije *File → New → Workspace* pokreće se wizard kojim će se formirati novi projekat (grupa projekata u okviru workspace-a). U prvom prozoru koji se pojavljuje potrebno je u okviru text boxa *Type the workspace name* ukucati ime grupe projekta npr. *New Projects*, dok u okviru text boxa *Select the location of the workspace folder* ukucati putanju do željenog direktoriju (npr. *c:\program files\aldec\active-hdl 6.3\my_designs*), gde će se smestiti fajlovi koji nastaju u kasnijim fazama. Nakon ovog koraka napravljen je direktorijum *c:\Program Files\Aldec\Active-HDL 6.3\My_Designs\New_Projects* sa potrebnim fajlovima

Drugi prozor koji se pojavljuje nudi mogućnost importovanja postojećih fajlova u dizajn, ili formiranje praznog projekta. Selektovati npr. poslednju opciju *Create an empty design*

Treći prozor koji se pojavljuje nudi izbor alata koji bi se koristili u procesu implementacije budućeg VHDL opisa na ciljnoj komponenti. Ako je cilj samo izvršiti logičku simulacija opisanog hardvera u projektu koji se kreira ova polja mogu ostati prazna.

U četvrtom prozoru u okviru text boxa *Type the design name* potrebno je ukucati ime novog projekta npr. *New_Project.* Nakon ovoga završeno je kreiranje neophodnih direktorijuma i fajlova koji će se koristiti u kasnijim fazama

Pisanje VHDL koda počinje kreiranjem fajla (jednog ili više njih) koji će kasnije sadržati opis hardvera. Izborom *File → New → VHDL Source* pokreće se wizard za kreiranje novog VHDL source fajla. Prvi prozor nudi mogućnost importovanja već napisanog VHDL fajla. U drugom prozoru u okviru text boxa *Type the name of the source file to create* potrebno je ukucati ime novog VHDL fajla, ili dugmetom *Browse* selektovati postojeći VHDL fajl ako je u predhodnom prozoru izabrana opcija za importovanje napisanog fajla u dizajn. Ostala dva text boxa su opciona (*Type the name of the entity, Type the name of the architecture body)*, mogu ostati prazni ili ako se ispune odgovarajući deo koda (template) se automatski generiše za sadržaj entity-ja i architecture.

Naredni prozor nudi mogućnost specificiranja portova, ako se već unapred zna koje i koliko portova će hardver imati Ovo je takođe opciono i olakšava pisanje koda jer ako se specificiraju portovi automatski će se generisati deo koda sa adekvatnom sintaksom koja opisuje te portove. Selektovanjem *Finish* dugmeta završeno je generisanje VHDL fajla, ostaje još da se on ispuni.

Vremenska simulacija

Pre pokretanja vremenske simulacije, potrebno je izvršiti kompilaciju projekta. Selektovanjem *Design → Compile All* , pokreće se proces kompilacije, koji treba da potvrdi ispravnost napisanog koda. Ukoliko postoje sintaksne greške u kodu, one će se pojaviti u okviru *Console* prozora. Ako je kompilacija prošla bez grešaka, pojaviće se poruka npr. # Compile success 0 Errors 0 Warnings Analysis time : 0.5 [s]. Nakon ovoga moguće je izvršiti simulaciju. Selektovanjem *File → New → Waveform* kreira se površina (Waveform Editora) sa vremenskom podelom duž x ose, dok se na y osi prikazuju odgovarajući signali.

Selektovanjem *Waveform* → *Waveform properties* i izborom odgovarajućih opcija moguće je izabrati oblik u kom će se pojavljivati signali koji se ispituju u vremenu. Signali koji se prate u vremenu se dodaju na površinu selektovanjem *Waveform* → *Add Signals* , gde se po izvršenom selektovanju otvara prozor sa svim signalima (promenjivama) koji su definisani u procesu pisanja VHDL koda. Mogu se pojedinačno selektovati oni signali koji se žele pratiti u vremenu, odnosno zadavati im se test vrednosti u nekom vremenskom intervalu.

Zadavanje vrednosti ulaznim signalima, za koje se želi pratiti dešavanje unutar hardvera ili na njegovim izlazima u vremenu, vrši se selektovanjem odgovarajućeg signala u okviru Waveform Editora a zatim selektovanjem opcije *Waveform* → *Stimulators* . Nakon ovoga se otvara prozor koji nudi različite mogućnosti za dodelu vrednosti nekom signalu koji treba da bude test signal, neke od njih su *Clock, Formula, Value,....* prva mogućnost je da dati signal bude signal takta (mada ne mora, može i neki drugi signal periodično menjati vrednosti), druga mogućnost je da se dati signal definiše u vremenu npr. 01001110 0ns,10001110 50ns,11001110 100ns ..... ako je signal bio osmobitni vektor, treća mogućnost je da se forsira konstantna vrednost tokom celog vremenskog intervala itd.

Kada su definisani test signali izborom opcije *Simulation* → *Run* ili *Simulation* → *RunUntil* ili *Simulation* → *Run For* pokreće se odgovarajući mod simulacije, u prvom slučaju simulacija je bez vremenskog ograničenja (traje onoliko koliko je određeno trajanjem test signala), u drugom slučaju simulacija traje do nekog vremena koje se specificira, dok u trećem slučaju postoji neki fiksni korak u kojem se simulira rad kola. Korak simulacije u trećem slućaju se može promeniti izmenom vrednosti u odgovarajućoj kućici Waveform Editora.

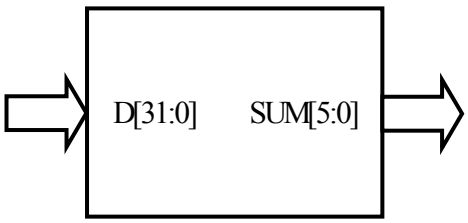
Zadatak 12.1.

Koristeći se metodom dizajniranja na bazi opisa ponašanja sistema, napraviti VHDL model digitalnog sistema koji broji ukupan broj jedinica u ulaznom vektoru. Ulazni vektor D se sastoji od 32 bita, ukupan broj jedinica se dobija na izlazu iz sistema kao vektor SUM (binarno kodovan broj jedinica). Vektori D i SUM su dati u smeru MSB>LSB (potrebno je izračunati koliki je broj bita u vektoru SUM i usvojiti minimalnu potrebnu širinu tog vektora). Nacrtati i blok šemu sistema. Operacija sabiranja objekta tipa STD_LOGIC_VECTOR i objekta tipa INTEGER je definisana u okviru paketa std_logic_unsigned koji se nalazi u IEEE biblioteci.

Rešenje:

Maksimalan broj jedinica koji se može pojaviti na ulazu sistema je 32, prema tome minimalna potrebna širina vektora S je 6 jer je $2^6 - 1 > 32 > 2^5 - 1$

Blok šema sistema je prikazana na slici 12.2.



Slika 12.2.

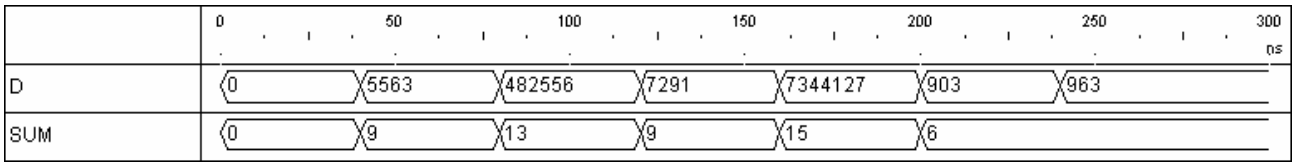
Na početku se usvaja promenjiva S koja služi za smeštanje trenutnog broja jedinica (njen tip je isti kao i tip izlaznog vektora, dakle STD_LOGIC_VECTOR). Glavni deo programa se sastoji od petlje u kojoj se prolazi kroz svih 32 bita ulaznog vektora D i proverava da li je odgovarajući bit u trenutnoj iteraciji 1 ili 0, ako je 1 promenjiva S se inkrementira, ako nije njena vrednost se ne menja. Pošto se sabira promenjiva S koja je tipa STD_LOGIC_VECTOR sa jedinicom koja je tipa INTEGER u slučaju inkrementiranja potrebno je na početku opisa uključiti std_logic_unsigned paket kako bi se ostvarila ova operacija korektno. Na kraju se vektoru SUM dodeljuje promenjiva S.

Kompletan opis kola dat je kao:

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

entity Vcnt1s is
   port ( D: in   STD_LOGIC_VECTOR (31 downto 0);
             SUM: out STD_LOGIC_VECTOR (5 downto 0));
end  Vcnt1s;

architecture Vcnt1s_arch of Vcnt1s is
begin
 process (D)
 variable S:  STD_LOGIC_VECTOR (5 downto 0);
 begin
  S:=”000000”;
  for i in 0 to 31 loop
     if D(i) = ‘1’ then S:= S + “000001”; end if;
```

```
  end loop;
  SUM <= S;
 end process;
end  Vcnt1s_arch;
```

Simulacija rada brojača jedinica u ulaznom vektoru je data na slici 12.3. Pošto je u pitanju kombinaciona mreža izlazi se menjaju odmah po promeni stanja na ulazu (ukoliko naredni broj na ulazu ima izmenjen broj jedinica).



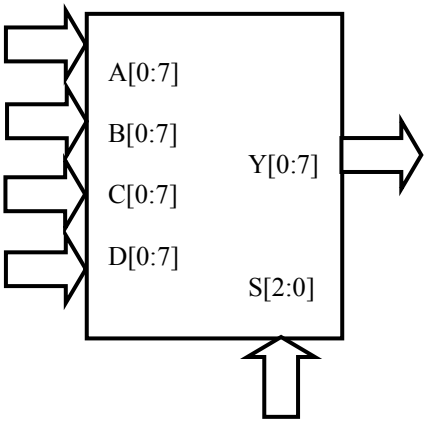
Slika 12.3.

Zadatak 12.2.

Koristeći se metodom dizajniranja na bazi opisa ponašanja sistema, napraviti VHDL model specijalizovanog multipleksera sa četiri osmobitna ulaza. Ulazni vektori su A, B, C, D, dok je izlazni vektor Y, upravljački vektor je S sa tri bita u smeru MSB>LSB, svi ostali vektori su u smeru LSB>MSB. Za vrednosti upravljačkih signala 000, 001, 010, 011 ..... 111 na izlazu se pojavljuju redom ulazi A, B, A, C, A, D, A, B. Kolo radi u pozitivnoj logici. Nacrtati i blok dijagram sistema.

Rešenje:

Blok dijagram sistema prikazan je na slici 12.4.



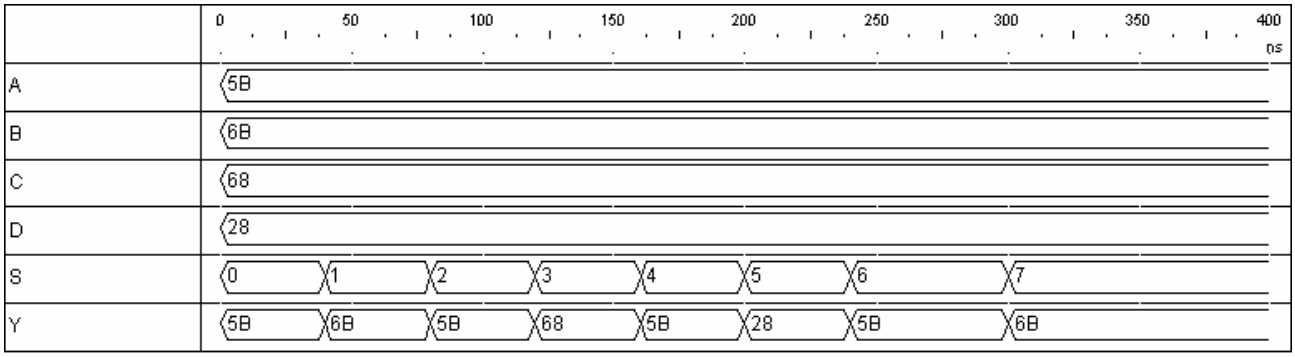
Slika 12.4.

Kompletan opis dat je kao:

```
library  IEEE;
use IEEE.std_logic_1164.all;

entity mux4in8b is
   port ( A, B, C, D: in STD_LOGIC_VECTOR  (0 to 7);
              S: in   STD_LOGIC_VECTOR (2 downto 0);
```

```
            Y: out STD_LOGIC_VECTOR  (0 to 7));
end  mux4in8b

architecture mux4in8p of mux4in8b is
begin
process (A, B, C, D, S)
  begin
    case S is
       when “000” | “010” | ”100” | “110”=> Y <=A;
       when “001” | “111” => Y <=B;
       when “011” => Y <=C;
       when “101” => Y <=D;
       when others => Y <= (others => ‘U’); -- kreira 8 bitni vektor od
    end case;                               -- ‘U’ stanja i pridružuje
  end process;                              -- ih vektoru Y
end  mux4in8p;
```

Simulacija rada specijalizovanog multipleksera je prikazana na slici 12.5. Vektori A, B, C, D imaju vrednosti koje se u vremenu koliko traje simulacija ne menjaju. Menja se samo vrednost vektora S u koracima, 000, 001, 010, 011 ..... 111. Analizom izlaza Y može se zaključiti da je VHDL model korektan, jer se na izlazu redom pojavljuju vrednosti koje odgovaraju vektorima A, B, C, D u sekvenci A, B, A, C, A, D, A, B.



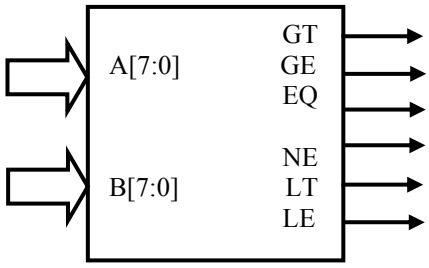
Slika 12.5.

Zadatak 12.3.

Koristeći se metodom dizajniranja na bazi opisa ponašanja sistema, napraviti VHDL model 8 bitnog komparatora. Na ulaz komparatora se dovode 8 bitni vektori A i B. Izlazi komparatora su sledeći signali EQ (EQ=1 za A=B inače EQ=0); NE (NE=1 za $A \neq B$ inače NE=0); GT (GT=1 za A>B inače GT=0); GE (GE=1 za $A \geq B$ inače GE=0); LT (LT=1 za A<B inače LT=0); LE (LE=1 za $A \leq B$ inače LE=0). Kolo radi u pozitivnoj logici, vektori A i B su dati u smeru MSB>LSB.

Rešenje:

Blok šema kola je prikazana na slici 12.6.



Slika 12.6.

VHDL poseduje operatore = ; /= ; > ; < ; >= ; <= koji se mogu koristi za poređenje objekata tipa integer, nabrojivih tipova (kao npr. STD_LOGIC), i jednodimenzione nizove nabrojivih tipova ili integer tipa. Nizovi se uvek porede s leva na desno bez obzira na redosled opsega indeksa („to" ili „downto"). Ako nizovi imaju nejednaku dužinu i svi elementi kraćeg niza su jednaki odgovarajućim elementima dužeg niza, tada se kraći niz smatra manjim prilikom poređenja.

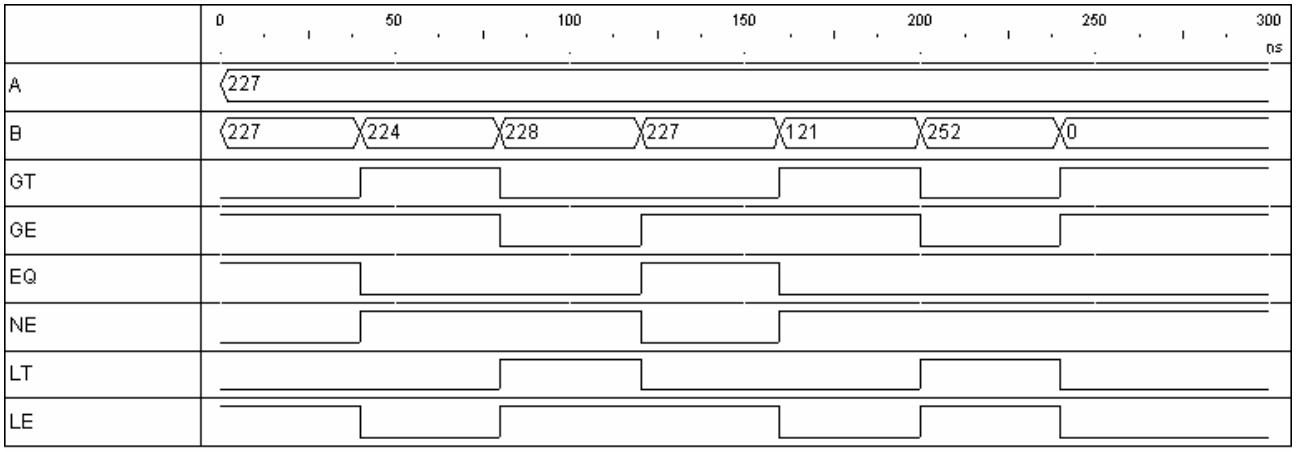
Entity odgovarajuće arhitekture kombinacionog kola će sadržati vektore A i B, kao i signale GT, GE, EQ, NE, LT, LE, pri čemu su A i B ulazni signali dok su GT, GE, EQ, NE, LT, LE izlazni.

Arhitektura će sadržati proces kao osnovni elemenat dizajna u duhu opisa ponašanja sistema. Proces će u listi osetljivosti posedovati vektore A i B. Poređenje se vrši tako što se vektori A i B uzimaju kao binarni brojevi predstavljeni sa 8 bita i porede njihove vrednosti.

Kompletan VHDL opis kola dat je kao:

```
library  IEEE;
use IEEE.std_logic_1164.all;

entity vcompare is
   port ( A, B: in   STD_LOGIC_VECTOR (7 downto 0);
          GT, GE, EQ, NE, LT, LE: out STD_LOGIC );
end  vcompare;

architecture vcompare_arch of vcompare is
begin
     process (A, B)
       begin
         EQ<='0'; NE<='0'; GT<='0'; GE<='0'; LT<='0'; LE<='0';
           If  A=B   then EQ<='1'; end if;
           If  A/=B  then NE<='1'; end if;
           If  A>B   then GT<='1'; end if;
           If  A>=B  then GE<='1'; end if;
           If  A<B   then LT<='1'; end if;
           If  A<=B  then LE<='1'; end if;
     end process;
end  vcompare_arch;
```

Simulacija rada osmobitnog komparatora je prikazana na slici 12.7. Ulaz A je konstantan u vremenu dok se ulaz B menja i posmatraju signali GT, GE, EQ, NE, LT, LE koji treba da označe odgovarajući odnos između A i B.



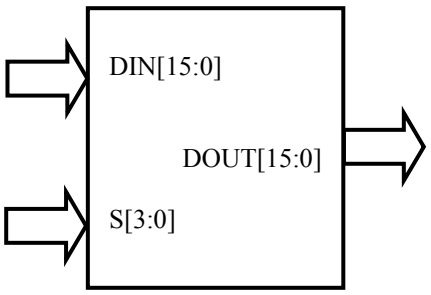
Slika 12.7.

Zadatak 12.4.

Koristeći se metodom dizajniranja na bazi opisa ponašanja sistema, napraviti VHDL model kružnog pomerača u levo. Kružni pomerač u levo (rol16) je kombinaciono kolo koje ima na ulazu vektor DIN [15:0], upravljački vektor S[3:0] i izlazni vektor DOUT[15:0]. Upravljački vektor S[3:0] određuje broj pomeraja ulevo npr. za S=”0010” DOUT će biti DIN pomeren za dva mesta u levo, s tim što će biti DOUT(1)=DIN(15) i DOUT(0)=DIN(14), za ostale bite važi DOUT(n)=DIN(n-2).

Rešenje:

Blok šema kola je prikazana na slici 12.8.



Slika 12.8.

U okviru entity-ja odgovarajuće arhitekture kao ulazni signali nalaze se DIN i S, dok je jedini izlaz vektor DOUT. Kompletno rešenje je:

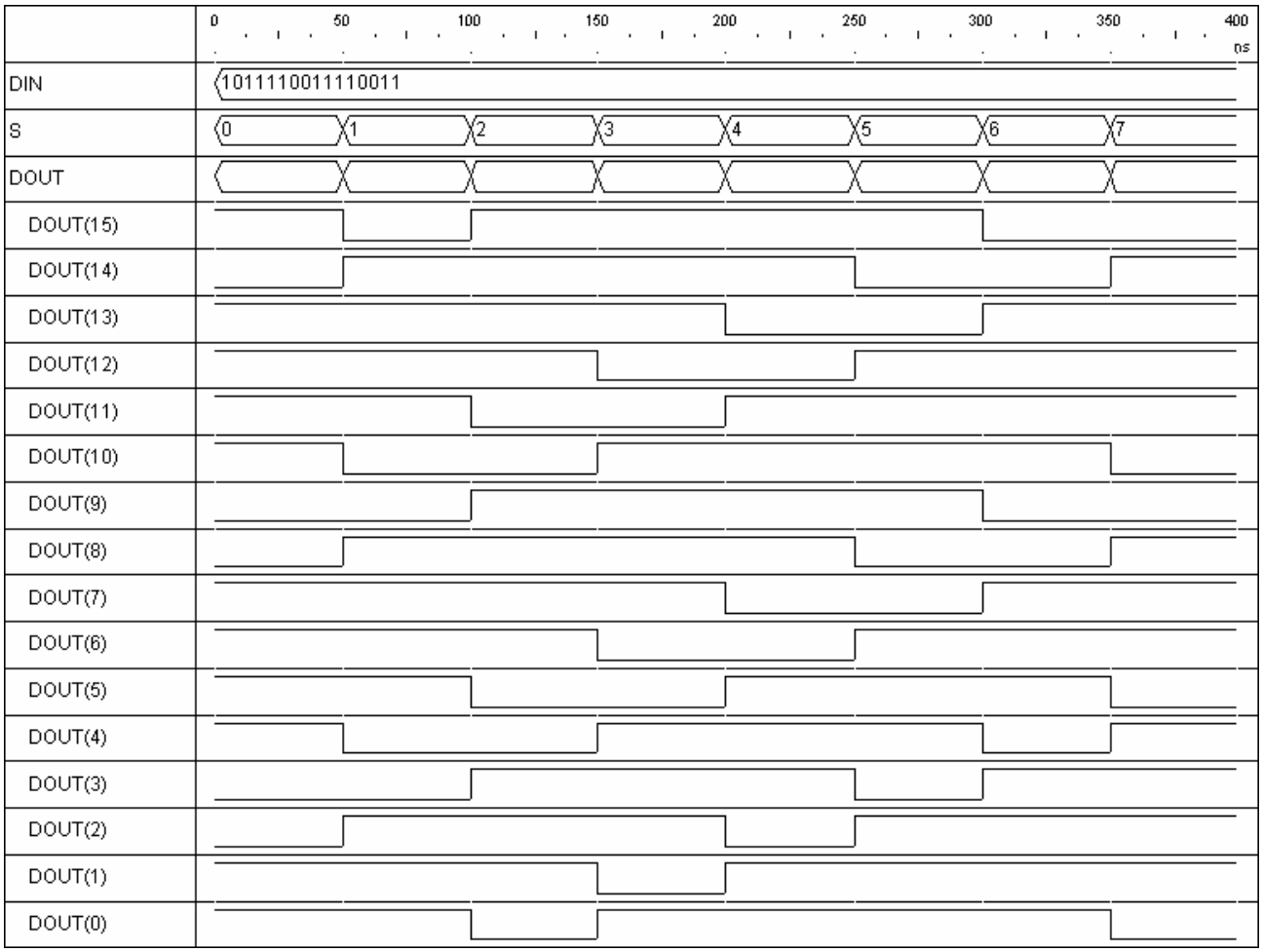
```
use IEEE.std_logic_1164.all;

entity rol16 is
   port (   DIN: in   STD_LOGIC_VECTOR (15 downto 0);
            S: in   STD_LOGIC_VECTOR (3 downto 0);
            DOUT: out   STD_LOGIC_VECTOR (15 downto 0));
end  rol16;
```

```
architecture rol16_arch of rol16 is
begin
     process (DIN, S)
      variable X, Z, Y: STD_LOGIC_VECTOR (15 downto 0);
       begin
           If  S(0)='1'  then
                X:= DIN (14 downto 0) & DIN(15);
           else X:= DIN;
           end if;
           If  S(1)='1'  then
                Y:= X (13 downto 0) & X (15 downto 14);
           else Y:= X;
           end if;
           If  S(2)='1'  then
                Z:= Y (11 downto 0) & Y (15 downto 12);
           else Z:= Y;
           end if;
           If  S(3)='1'  then
                DOUT<=Z (7 downto 0) & Z (15 downto 8);
           else DOUT<=Z;
           end if;
     end process;
end  rol16_arch;
```

Arhitekura definiše tri promenjive X, Z, Y koje se koriste radi lakšeg ostvarivanja funkcije pomeranja. U prvoj if then liniji se proverava da li je S(0)=1 ako jeste znači da postoji pomeranje sigurno za jedan u levo, to se ostvaruje i međurezultat se čuva u promenjivoj X (u slučaju da treba izvršiti još pomeranja ulevo), ako je S(0)=0 nema pomeranja i X postaje DIN. U drugoj liniji se ispituje da li je S(1)=1, ako jeste znači da postoji još i pomeranje za dva u levo, što se onda vrši nad promenjivom X koja je ili pomeren ulaz (ako je S(0) bilo 1) ili originalan ulaz ako nije bilo pomeranja (ako je S(0) bilo 0). U trećoj if then liniji se ispituje da li postoji još četiri pomeranja ulevo i to se izvršava ako je S(2)=1. Četvrta if then linija ispituje da li ima još osam pomeranja, dakle logika je ista kao i u predhodnim linijama samo što se u svakoj narednoj vrši pomeranje za 2x više mesta jer svaki S(i) vredi dva puta više od S(i-1) i=1, 2, 3.

Operator „ & ” spaja nizove, npr od dva niza A i B dužine n i m nastaje niz C dužine n+m ako se napiše C=A&B. Pri čemu je C(m+n-1)=A(n-1), C(m+n-2)=A(n-2), ... C(0)=B(0).

Simulacija rada kružnog pomerača ulevo je prikazana na slici 12.9. Ulaz DIN se ne menja u vremenu i ima vrednost 1011110011110011. Upravljački vektor S na svakih $t = 50$ ns promeni vrednost, tako da je nova vrednost za jednu veća od prethodne. Prema tome vektor S uzima vrednosti 000,001,010,011,100,101,110,111. Na slici su prikazani pojedinačni biti vektora DOUT i njihove promene.



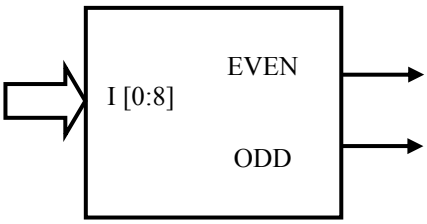
Slika 12.9.

Zadatak 12.5.

U nekom digitalnom sistemu se podaci prenose po 8 bitnoj magistrali. Radi provere korektnosti prenosa informacija dodaje se još i bit parnosti, tako da podatak pored osnovnih 8 bita poseduje još jedan bit koji se generiše tako da ukupan zbir jedinica (podatak plus bit parnosti) bude ili paran ili neparan (što se usvaja po uspostavljanju protokola za prenos informacija). Koristeći se metodom dizajniranja na bazi opisa ponašanja sistema, napraviti VHDL model digitalnog kola koje treba da vrši detekciju parnosti primljene informacije, ukoliko je ukupan broj jedinica u paketu (8 bita podatka plus bit parnosti) paran treba postaviti signal EVEN na logičku jedinicu (inače je nula), ako je ukupan broj jedinica neparan potrebno je postaviti izlazni signal ODD na logičku jedinicu. Kolo radi u pozitivnoj logici. Ulazni vektor je I (devet bita u redosledu LSB>MSB).

Rešenje:

Blok šema kola je prikazana na slici 12.10.



Slika 12.10.

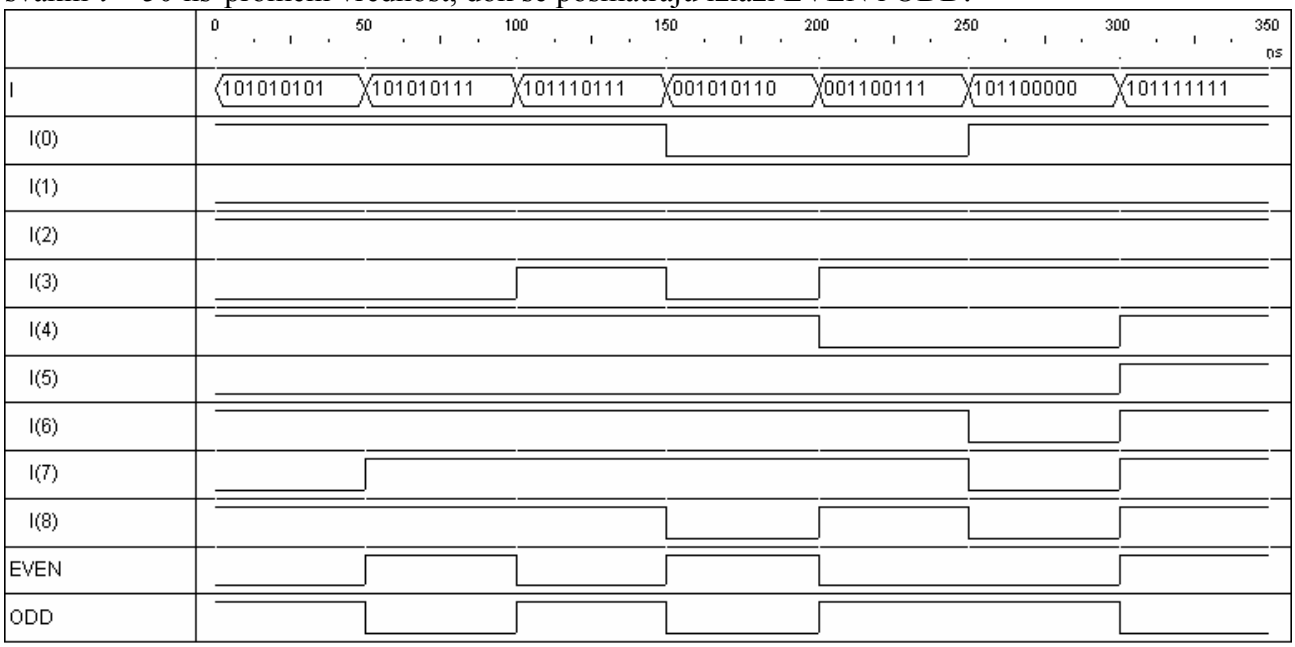
Entity odgovarajuće arhitekture poseduje ulazni vektor I [0:8] i izlazne signale EVEN i ODD.

Arhitektura će sadržati proces kao osnovni elemenat dizajna u duhu opisa ponašanja sistema. Proces će u listi osetljivosti posedovati vektor I [0:8].

Arhitektura definiše promenjivu p koja sadrži trenutnu parnost. Na početku joj se dodeljuje vrednost LSB bita ulaznog vektora. U petlji se redom proveravaju ostali biti u ulaznom vektoru, kada se desi da neki od ovih bita ima vrednost logičke jedinice parnost se promeni (promenjiva p dobija suprotnu vrednost u logičkom smislu, jer pojava jedinice znači i promenu parnosti u odnosu na predhodno izračunatu parnost). Na kraju se signalima ODD i EVEN dodeljuju vrednosti promenjive p, kada se prođe kroz svih devet bita.

```
library  IEEE;
use IEEE.std_logic_1164.all;

entity parity9 is
   port ( I: in   STD_LOGIC_VECTOR (0 to 8);
          EVEN, ODD: out   STD_LOGIC);
end  parity9;

architecture parity9_arch of parity9 is
begin
process (I)
 variable p : STD_LOGIC;
  begin
     p:= I(0);
     for j in 1 to 8 loop
        if  I(j) = '1' then p:= not p; end if;
     end loop;
     ODD <=p;
     EVEN<= not p;
  end process;
end  parity9_arch;
```

Simulacija rada digitalnog kola za proveru parnosti je prikazana na slici 12.11. Ulazni vektor I na svakih $t = 50$ ns promeni vrednost, dok se posmatraju izlazi EVEN i ODD.



Slika 12.11.

Zadatak 12.6.

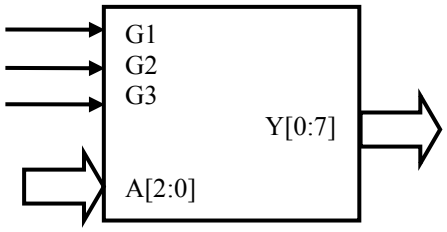
Koristeći se metodom dizajniranja na bazi opisa ponašanja sistema, napraviti VHDL model 3 u 8 dekodera. Ulazni vektor A je dat u smeru MSB>LSB, dok je izlazni vektor Y u smeru LSB>MSB. Kolo radi u pozitivnoj logici (za ulaz A=’’001’’ izlaz je Y=’’01000000’’). Dekoder poseduje još i tri kontrolna ulaza (G1, G2, G3) gde se preslikavanje (funkcija) dekodera ulaz>izlaz ostvaruje samo ako su kontrolni ulazi svi na logičkoj jedinici, inače se na izlazu pojavljuje vektor Y=’00000000’. Kolo radi u pozitivnoj logici (za ulaz A=’’001 ’’ izlaz je Y=’’01000000’’), vektor A je dat u smeru MSB>LSB, dok je vektor Y u smeru LSB>MSB.

Rešenje:

Osnovni element u opisu ponašanja sistema u VHDL dizajnu je **proces**. **Proces** je kolekcija sekvencijalnih koraka koji se izvršavaju konkurentno (paralelno) sa ostalim konkuretntin koracima i ostalim procesima. Upotrebom procesa mogu se specificirati kompleksne interakcije signala i događaja onako kako bi se one odvijale u simulaciji sistema (bez vremenske dimenzije).

Dekoder je kombinaciono kolo koje ima (u ovom slučaju) 3 ulaza i 8 izlaza. Pored ovih osnovnih ulaza (koji se dekoduju i samo jedan od 8 izlaza je aktivan u odnosu na odgovarajući ulaz), dekoder poseduje i 3 kontrolna ulaza koji omogućuju funkciju dekodera samo ako su svi na logičkoj jedinici. Prema tome u entity-ju opisa ovog kola pojaviće se osim ulaznih signala koji se dekoduju (adrese A[2]-A[0]) i signali G1, G2, G3 kao kontrolni, dok će izlazi biti Y[0]-Y[7].

Blok šema kola je prikazana na slici 12.12.



Slika 12.12.

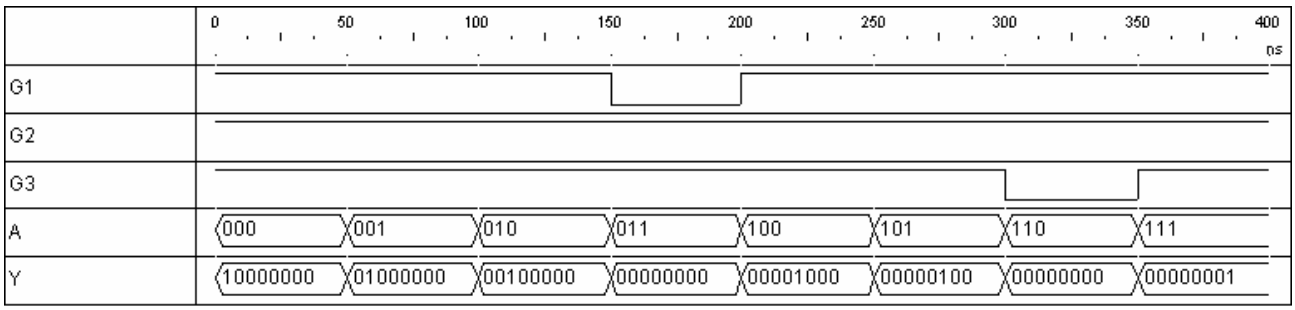
Kompletan VHDL opis je dat u nastavku,

```
library  IEEE;
use IEEE.std_logic_1164.all;

entity V3to8dec is
   port (G1, G2, G3: in STD_LOGIC;
         A: in   STD_LOGIC_VECTOR (2 downto 0);
         Y: out STD_LOGIC_VECTOR  (0 to 7));
end  V3to8dec;

architecture V3to8dec_b of V3to8dec is
  signal Y_s: STD_LOGIC_VECTOR  (0 to 7);
begin
process (A, G1, G2, G3, Y_s)
  begin
    case A is
       when "000" => Y_s <="10000000";
```

```
        when "001" => Y_s <="01000000";
        when "010" => Y_s <="00100000";
        when "011" => Y_s <="00010000";
        when "100" => Y_s <="00001000";
        when "101" => Y_s <="00000100";
        when "110" => Y_s <="00000010";
        when "111" => Y_s <="00000001";
        when others => Y_s <="00000000";
    end case;
    if (G1 and G2 and G3)='1' then Y<=Y_s;
    else Y <="00000000";
    end if;
  end process;
end  V3to8dec_b;
```

U okviru arhitekture uveden je interni signal Y_s, kojem se prvo dodeljuje vrednost u odnosu na vektor A, bez obzira na signale G1, G2, G3. Ako su ovi signali pozitivni, tek onda se vrši dodela izlaznom vektoru Y. Opšti princip je da signali koji se deklarišu u okviru procesa (ovde je to Y_s) učestvuju u njegovoj listi osetljivosti.

Simulacija rada dekodera 3 u 8 je prikazana na slici 12.13. Ulazni vektor A na svakih $t = 50$ ns promeni vrednost, dok se posmatraju izlazi vektora Y. U nekim trenucima G1 i G3 postanu neaktivni čime se proverava ispravnost funkcionisanja dekodera u odnosu na upravljačke signale.



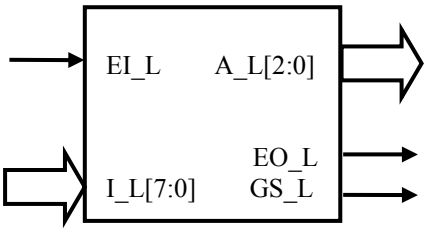
Slika 12.13.

Zadatak 12.7.

Koristeći se metodom dizajniranja na bazi opisa ponašanja sistema, napraviti VHDL model 8 u 3 kodera prioriteta (74x148). Ulazni vektor I_L i izlazni A su dati u smeru MSB>LSB. Kolo radi u negativnoj logici (za ulaz I_L=''0xxxxxxx '' izlaz je A_L=''000''). Koder poseduje još kontrolni ulaz EI_L gde se preslikavanje (funkcija) dekodera ulaz>izlaz ostvaruje samo ako je kontrolni ulaz EI_L na logičkoj nuli (EI_L='0'), inače se na izlazu pojavljuje vektor A_L='111', GS_L='1', EO_L='1'. Kolo radi u negativnoj logici (za ulaz I_L=''0xxxxxxx '' izlaz je A_L=''000''), vektori I_L i A su dati u smeru MSB>LSB. Kolo poseduje još i dva izlaza GS_L i EO_L koji se koriste prilikom kaskadnog vezivanja više kodera (npr 16-bitni koder bi se dobio vezivanjem dva 8-bitna kodera). Izlaz GS_L je na logičkoj jedinici (GS_L='1') u slučaju I_L=''11111111'', EI_L='0' odnosno I_L=''xxxxxxxx'', EI_L='1', dok je u svim ostalim slučajevima GS_L='0'. Izlaz EO_L je na logičkoj nuli (EO_L='0') samo ako je I_L=''11111111'', EI_L='0', inače je u svim ostalim slučajevima EO_L='1'. Koristiti CONV_STD_LOGIC_VECTOR (n,j) funkciju koja pretvara broj n (INTEGER) u STD_LOGIC_VECTOR dužine j. Ova funkcija se nalazi u okviru IEEE biblioteke u std_logic_arith paketu.

Rešenje:

Isto kao i u predhodnom slučaju polazi se od blok dijagrama sistema, slika 12.14.



Slika 12.14.

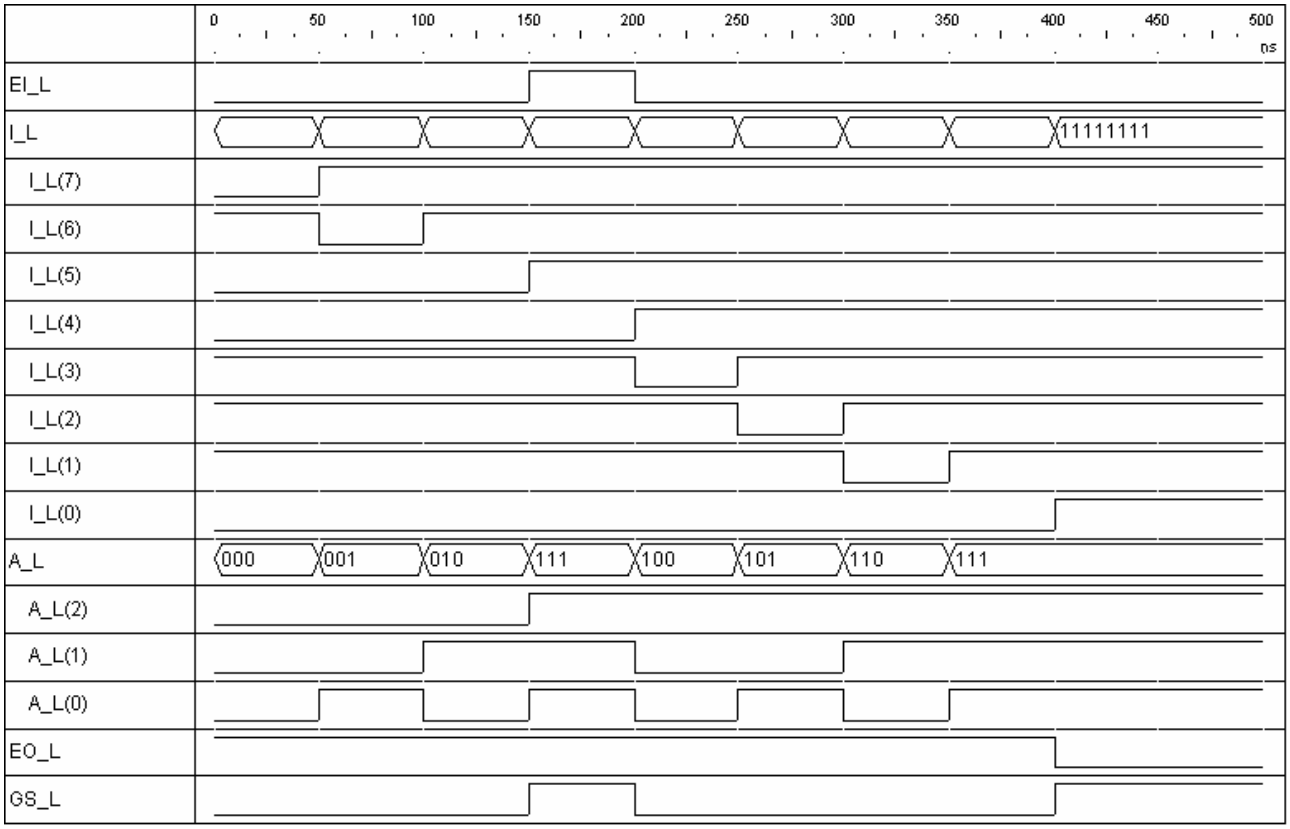
Kompletan VHDL opis kodera prioriteta je dat u nastavku.

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

entity V74x148 is
   port ( EI_L:       in STD_LOGIC;
          A_L:        out   STD_LOGIC_VECTOR (2 downto 0);
          I_L:        in STD_LOGIC_VECTOR  (7 downto 0);
          EO_L, GS_L: out STD_LOGIC);
end  V74x148;
architecture V74x148_b of V74x148 is
  signal EI:  STD_LOGIC;            -- ekvivalent u pozitivnoj logici
  signal A:   STD_LOGIC_VECTOR (2 downto 0); -- ekvivalent u p.logici
  signal I:   STD_LOGIC_VECTOR  (7 downto 0) -- ekvivalent u p.logici
  signal EO, GS:  STD_LOGIC;                 -- ekvivalent u p.logici
begin
  process (EI_L, I_L, EI, A, I, EO, GS)
  variable j: INTEGER range 7 downto 0;
  begin
    EI <= not  EI_L;           -- konverzija ulaza u pozitivnu logiku
     I <= not   I_L;           -- konverzija ulaza u pozitivnu logiku
    EO <= '1'; GS<='0'; A<="000";
    If (EI)='0' then EO<='0';
    else for j in 7 downto 0 loop
         if I(j)='1' then
            GS<='1'; EO<='0'; A<= CONV_STD_LOGIC_VECTOR(j, 3);
            exit;
         end if;
       end loop;
    end if;
    EO_L<= not EO;              -- konverzija izlaza u negativnu
logiku
    GS_L<= not GS;              -- konverzija izlaza u negativnu
logiku
    A_L <= not A;               -- konverzija izlaza u negativnu
logiku
  end process;
end  V74x148_b;
```

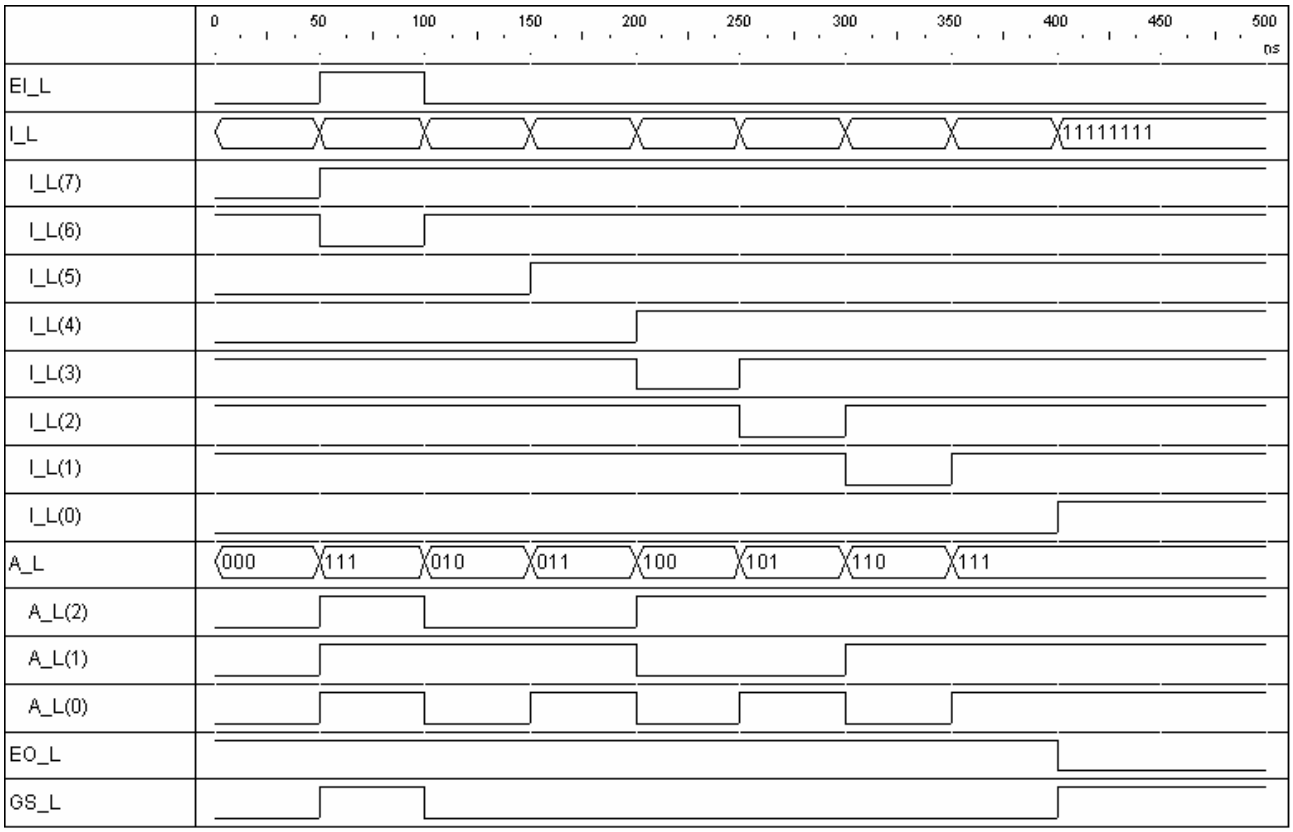
U arhitekturi se koristi funkcija CONV_STD_LOGIC_VECTOR(j, 3), koja pretvara broj „j" u STD_LOGIC_VECTOR dužine 3 (pošto je to dužina vektora A). Zbog ove funkcije je bilo potrebno pretvarati ulaze u pozitivnu logiku, u pozitivnoj logici rešiti funkciju kola a potom se vratiti u negativnu logiku transformacijama na kraju arhitekture.

Prikazani pristup je više u duhu „opisa ponašanja sistema" u odnosu na prethodni zadatak gde se eksplicitno vršila dodela vrednosti vektoru Y_s (pandan izlaznom vektoru Y). U petlji (loop) se polazi od najstarijeg bita (j=7) i proverava da li je on '1' ako nije, promenjiva j se dekrementira i proverava se naredni bit ulaznog vektora I_L (njegovog pandana u pozitivnoj logici), ako je I(j)= '1' za neko j, petlja se prekida i postavljaju se interni signali GS, EO, A, u pozitivnoj logici, koji se kasnije transformišu u pandane u negativnoj logici, jer je rečeno u postavci zadatka da kolo radi u negativnoj logici. Da to nije rečeno ne bi bilo potrebno vršiti ove transformacije, takođe da funkcija CONV_STD_LOGIC_VECTOR radi u negativnoj logici ne bi bilo potrebe za ovim transformacijama.

Simluacija rada kodera prioriteta je prikazana na slikama 12.15 i 12.16, gde je uzeto da se upravljački signal EI_L, menja u dva različita trenutka, dok se ulazni vektor I_L menja tako da su prisutne sve moguće vrednosti na izlazu za vektor A_L.



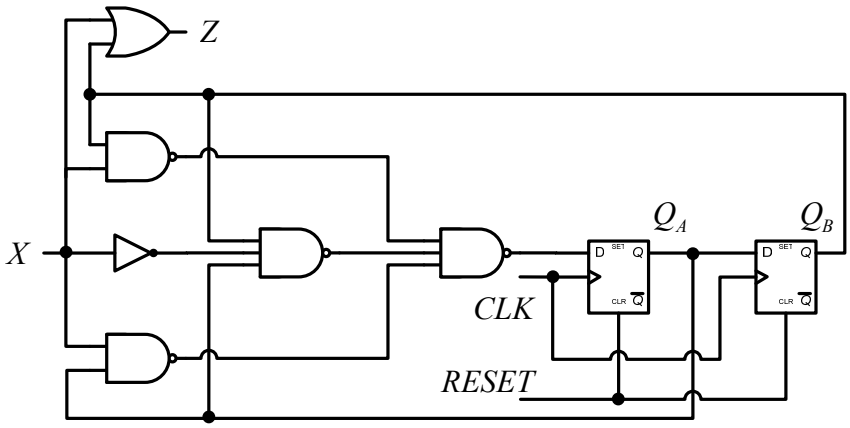
Slika 12.15.



Slika 12.16.

Zadatak 12.8.

Na slici 12.17 je prikazano digitalno kolo čije se ponašanje može opisati preko konačne mašine stanja FSM (Finite State Machine). Ulaz mašine stanja je signal X dok je izlaz signal Z. Pored ulaza i izlaza mašina stanja ima i dve promenljive stanja Q_A i Q_B. Pored signala X na ulaz mašine stanja se dovodi još i signal takta CLK koji omogućava prelazak iz jednog u drugo stanje na svaku uzlaznu ivicu u zavisnosti od ispunjenosti uslova za taj prelazak, kao i signal asinhronog reseta $RESET$, koji svojom kratkotrajnom pojavom prevodi FSM u stanje $Q_A = 0$ i $Q_B = 0$.



Slika 12.17.

a) Napisati tabelu prelaza za ovu FSM.

b) Nacrtati dijagram stanja za ovu FSM.

c) Implementirati opisanu FSM u VHDL-u.

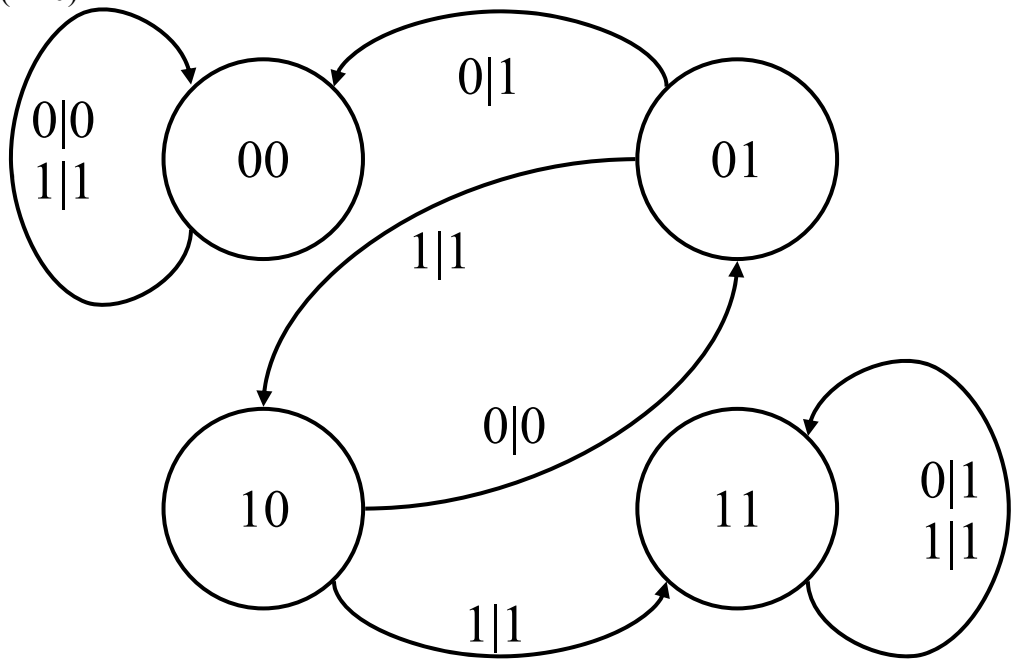
Rešenje

a) Tabela prelaza za FSM je data na slici 12.8. Uzimajući u obzir sva moguća stanja Q_A i Q_B, kao i ulaz X, prikazane su vrednosti kontrolnih signala D flip flopova, D_A i D_B, kao i izlaznog signala Z za svaku kombinaciju Q_A, Q_B i X.

Q_A	Q_B	X	D_A	D_B	Z
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	0	1
0	1	1	1	0	1
1	0	0	0	1	0
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Slika 12.18.

b) Dijagram stanja za FSM je prikazan na slici 12.9. Pored svake strelice koja označava prelaz iz jednog u drugo stanje nalazi se uslov prelaska i vrednost izlaza koja se uspostavlja prelaskom u naredno stanje u formatu *uslov/izlaz*. Npr. za stanje 01 prelazak u stanje 00 je moguće za vrednost kontrolne promenljive X=0, pri čemu je izlaz Z=1. Prelaz se ostvaruje nailaskom prve uzlazne ivice signala takta pri ispunjenom uslovu prelaska (X=0)



Slika 12.19.

c) Implementacija opisane mašine stanja u VHDL-u je data narednim VHDL kodom.

```
library ieee;
use ieee.std_logic_1164.all;
use work.std_arith.all;
entity cd_fsm is port(
     clk      : in std_logic;
     reset    : in std_logic;
     X        : in std_logic;
     Z        : out std_logic;
     stateout : out std_logic_vector(l downto 0));
end cd_fsm;

architecture state_machine of cd_fsm is
     type StateType is (ZeroZero, ZeroOne, OneZero, OneOne);
     signal present_state, next_state : StateType;

begin
     with present_state select
          stateout <=
          "00" when ZeroZero,
          "11" when OneOne,
          "01" when ZeroOne,
          "10" when OneZero,
          "10" when others;

state_comb:process(present_state, X)
begin
     case present_state is
          when ZeroZero =>
               next_state <= ZeroZero;
               Z <= X;
          when ZeroOne =>
               if X = '1' then
                    next_state <= OneZero;
               else
                    next_state <= ZeroZero;
               end if;
               Z <= '1';
          when OneZero =>
               if X = '1' then
                    next_state <= OneOne;
               else
                    next_state <= ZeroOne;
               end if;
               Z <= X;
          when OneOne =>
               next_state <= OneOne;
               Z <= '1';
          when others =>
               next_state <= OneZero;
               Z <= '0' ;
     end case;
end process;

state_clocked:process(clk, reset)
```

```
begin
     if (reset = '1') then
          present_state <= ZeroZero;
     elsif rising_edge(clk) then
          present_state <= next_state;
     end if;
end process state_clocked;

end state_machine;
```

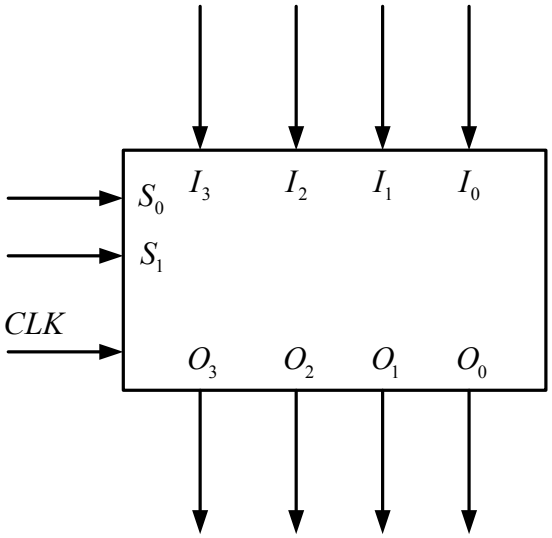
Definisana su dva procesa:

a) state_comb, za promenu izlaznog signala Z, kao i određivanje narednog stanja FSM, koji je kombinacionog tipa, dakle u opštem slučaju ne zavisi od signala takta, i postaje aktivan sa promenom ulaznog signala X, odnosno sadašnjeg stanja FSM.
b) state_clocked, koji određuje prelazak u naredno stanje, koje se dešava pojavom uzlazne ivice signala takta - CLK, odnosno pojavom asinhronog reseta - $RESET$ gde se FSM prevodi u stanje $Q_A = 0$, $Q_B = 0$.

Na početku je definisan tip StateType, zbog jednostavnije notifikacije stanja FSM.

Zadatak 12.9.

Na slici 12.20 je prikazana blok šema sinhronog pomeračkog bidirekcionog registra. Pomeranje se vrši cirkularno ili u levo ili u desno, osim ovih funkcija registar može vršiti paralelni upis, odnosno zadržati postojeće stanje. Svaka od pomenutih aktivnosti se odvija u odnosu na uzlaznu ivicu signala takta CLK. Za odgovarajući režim rada registra koriste se signali S_0 i S_1. U tabeli na slici 12.21 dati su režimi rada registra u zavisnosti od signala S_0 i S_1. Implementirati ovaj registar u VHDL-u.



Slika 12.20.

S_1	S_0	Režim rada
0	0	Zadržava se postojeće stanje
0	1	Pomeranje u levo
1	0	Pomeranje u desno
1	1	Paralelni upis

Slika 12.21.

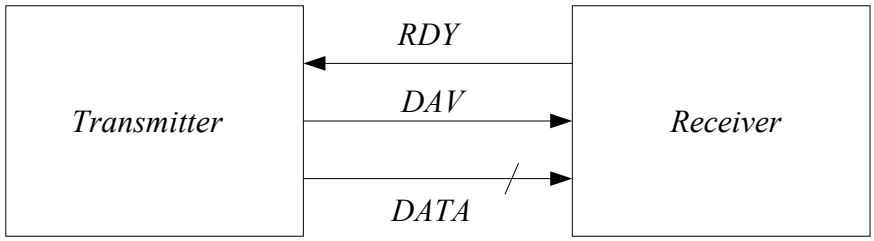
Rešenje:

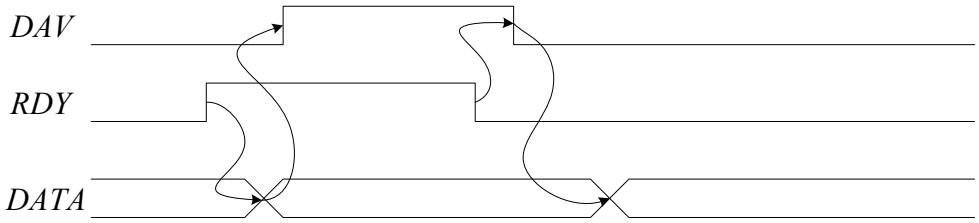
```
library ieee;
use ieee.std_logic_1164.all;
entity shifter is
     port (
          clk : in std_logic; -- signal takta
          i : in std_logic_vector(3 downto 0); -- biti na ulazu
```

```
          s : in std_logic_vector(1 downto 0); -- kontrolni signali
          o : out std_logic_vector(3 downto 0); -- biti na izlazu
end shifter;
architecture structure of shifter is
     signal internal : std_logic_vector(3 downto 0);--interna
     promenjiva
begin
     stuff: process (clk)
     begin
          if rising_edge(clk) then
               if s=”00” then –-za S1So=00,ostaje u sadašnjem stanju
                    internal <= internal;
               elsif s=”01” then –- ako je S1So=01, pomera ulevo
                    internal(0) <= i(3);
                    internal(1) <= i(0);
                    internal(2) <= i(1);
                    internal(3) <= i(2);
               elsif s=”10” then -- ako je S1So=10, pomera u desno
                    internal(0) <= i(1);
                    internal(1) <= i(2);
                    internal(2) <= i(3);
                    internal(3) <= i(0);
               elsif s=”11” then -- ako je S1So=11, paralelni upis
                    internal <= i;
               else
                    internal <= i;
               end if;
          end if;
     end process stuff;
     o<= internal; -- interna promenjiva se sada šalje na izlaz
end structure;
```

Zadatak 12.10.

U okviru UART-a (Universal Asynchronous Receiver Transmitter) za komunikaciju u procesu asinhrone razmene podataka koristi se tehnika „handshake”. Blok dijagram učesnika u „handshake” protokolu sa vremenskim dijagramima karakterističnih signala kojim se ovaj protokol odvija je prikazan na slici 12.22.





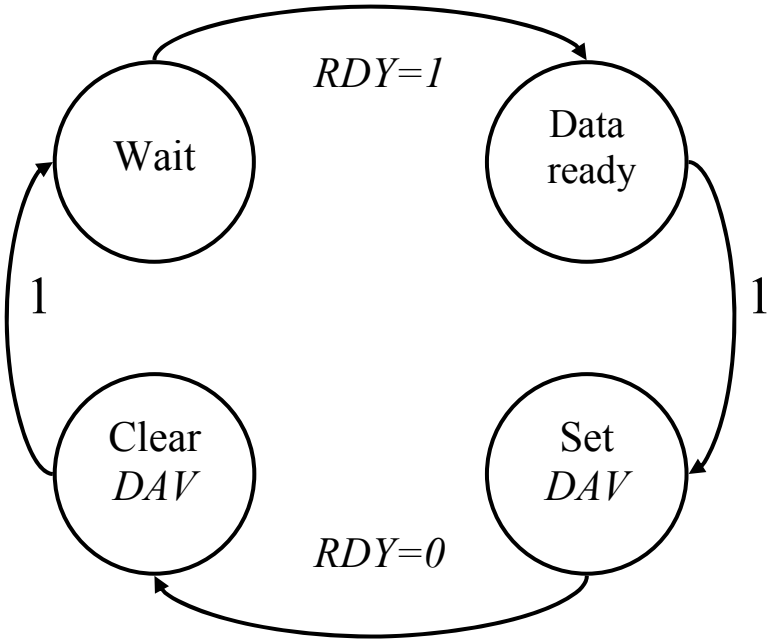
Slika 12.22.

- Prijemnik (receiver) kada je spreman da prihvati podatak signalizira postavljanjem signala *RDY* na logičku jedinicu
- Predajnik (transmitter) tada postavlja podatke *DATA*, i nakon nekog vremena po uspostavljanju stabilnog stanja na linijama magistrale podataka (uzeti da je ona četvorobitna i da se može proizvoljno menjati), signalizira da su podaci važeći postavljanjem signala *DAV* na logičku jedinicu
- Prijemnik po očitavanju podataka postavlja signal *RDY* na logičku nulu signalizirajući time svoju zauzetost
- Predajnik na poslednju akciju prijemnika odgovara postavljanjem signala *DAV* na logičku nulu, čime se završava jedan ciklus razmene podataka.

Opisati u VHDL-u predajnik (transmitter) i prijemnik (receiver) u procesu razmene podataka „handshake" tehnikom.

Rešenje:

Predajnik (transmitter) i prijemnik (receiver) u procesu razmene podataka „handshake" tehnikom se mogu opisati kao konačne mašine stanja. Pošto je komunikacija između predajnika i prijemnika asinhrona, pretpostavka je da predajnik i prijemnik rade sa različitim signalima takta, predajnik sa clk, prijemnik sa rclk. Dijagram stanja za FSM predajnika je prikazan na slici 12.23.



Slika 12.23.

FSM predajnika ima četiri stanja koja su usvojena u odnosu na režime rada u kojima se može naći predajnik (Wait-čeka se prijemnik za signal kada je spreman da otpočne prijem itd..). Prelazak iz stanja Data ready u stanje Set DAV je bezuslovan, odnosno dešava se na prvu sledeću rastuću ivicu signala takta bez obzira na vrednosti ostalih signala u kolu. Takav prelaz se označava jedinicom pored strelice prelaza.

Prelazak iz stanja Data ready u stanje Set DAV se odvija bezuslovno, kao i prelazak iz stanja Clear DAV u stanje Wait.

VHDL kod za ovu FSM predajnika je dat u nastavku

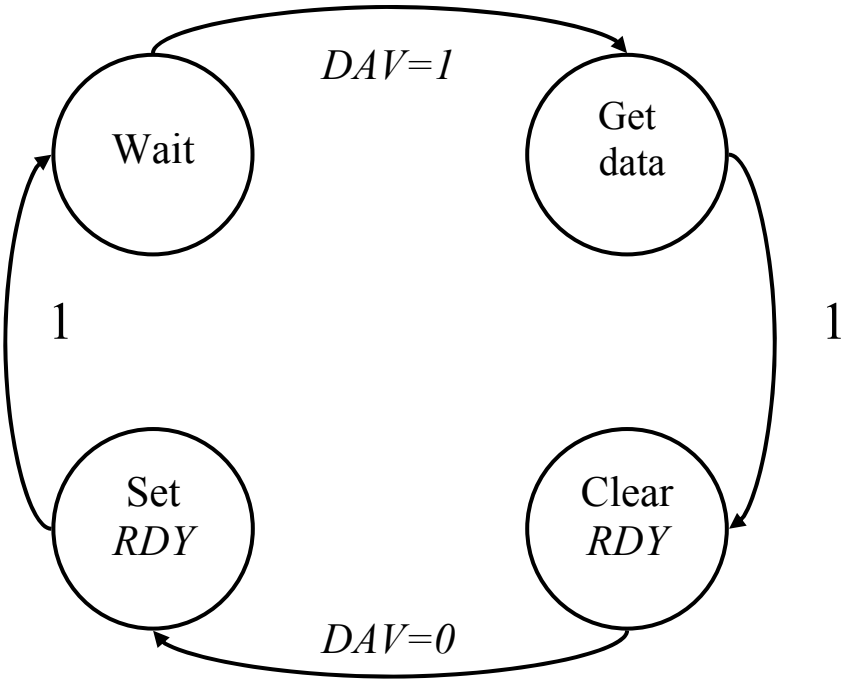
```
library ieee;
use ieee.std_logic_1164.all;

entity fullsend is
      generic (size: integer := 4);
      port (rdy, clk : in std_logic;
            datin : in std_logic_vector(size-1 downto 0);
            dav : out std_logic;
            datout : out std_logic_vector(size - 1 downto 0));
end fullsend;

architecture behavioral of fullsend is
      type StateType is (wt, dat, d_av, r_dy);
      attribute enum_encoding of StateType: type is "00 01 11 10";
      signal state : StateType;
begin
      dav <= '1' when (state = r_dy) else '0';
      handshake : process(clk)
      begin
            if rising_edge(clk) then
                  case state is
                        when wt =>
                              if rdy = '1' then
                                    state <= dat;
                              else
                                    state <= wt;
                              end if;
                        when dat =>
                              datout <= datin;
                              state <= r_dy;
                        when r_dy =>
                              if rdy = '0' then
                                    state <= d_av;
                              else
                                    state <= r_dy;
                              end if;
                        when d_av =>
                              state <= wt;
                  end case;
            end if;
      end process handshake;
end;
```

Stanjima sa dijagrama stanja za FSM predajnika (slika 12.23) odgovaraju sledeće oznake u kodu:

- Wait - wt
- Data ready - dat
- Set DAV - r_dy
- Clear DAV - d_av

Uzeto je da je magistrala podataka četvorobitna, a njena širina se može proizvoljno menjati zadavanjem vrednosti generika *size*. Vremenski interval između postavljanja podataka na magistralu podataka i podizanja signala *DAV* je uzet da bude jedna perioda taktnog intervala (uvođenjem i dekrementiranjem pomoćne promenljive taj interval se može proširiti).

Dijagram stanja za FSM prijemnika je prikazan na slici 12.24.



Slika 12.24.

Prelazak iz stanja Get data u stanje Clear RDY se odvija bezuslovno, kao i prelazak iz stanja Set RDY u stanje Wait.

VHDL kod za ovu FSM prijemnika je dat u nastavku.

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

entity fullrecv is
      generic (size: integer := 4);
      port (dav, rclk : in std_logic;
            datin : in std_logic_vector(size-1 downto 0);
            rdy : out std_logic;
            datout : out std_logic_vector(size - 1 downto 0));
end fullrecv;

architecture behavioral of fullrecv is
      type StateType is (w_dav, datav, r_rdy, wt_ndav);
      attribute enum_encoding of StateType: type is "00 01 11 10";
      signal state : StateType;
begin
      rdy <= '1' when (state=w_dav) or (state=r_rdy) or (state=datav)
      else '0';
      handshake : process(rclk)
      begin
            if rising_edge(rclk) then
                  case state is
                        when w_dav =>
```

```
                              if dav = '1' then
                                    state <= datav;
                              else
                                    state <= w_dav;
                              end if;
                        when datav =>
                              datout <= datin;
                              state <= wt_ndav;
                        when wt_ndav =>
                              if dav = '0' then
                                    state <= w_dav;
                              else
                                    state <= wt_ndav;
                              end if;
                        when r_rdy =>
                              state <= w_dav;
                  end case;
            end if;
      end process handshake;
end;
```

Prijemnik kada je spreman da prihvati podatak, signalizira postavljanjem signala *RDY* na logičku jedinicu, iz stanja **Set RDY** (r_rdy) prelazi u stanje **Wait** (w_dav) gde čeka da predajnik izbaci podatke na magistralu i postavi signal *DAV* na logičku jedinicu. Kad je predajnik postavio podatke na magistralu i podigao signal *DAV*, prijemnik prelazi u stanje **Get data** (datav) gde očitava podatke i u narednom taktnom ciklusu prelazi u stanje **Clear RDY** (wt_ndav) u kojem postavlja signal *RDY* na nulu i čeka da predajnik obori signal *DAV* na nulu. Kada se to desi prijemnik je spreman za novi ciklus prijema podataka.

Zadatak 12.11.

U nastavku je dat VHDL kod u okviru fajla Odd.vhd koji predstavlja implementaciju nekog hardverskog uređaja. Potrebno je:

a) Opisati funkciju hardverskog uređaja čija je implementacija data kodom u okviru fajla odd.vhd

b) Nacrtati blok dijagram koji opisuje strukturu uređaja koji je opisan u Odd.vhd fajlu

Odd.vhd

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

entity unknown is
      port (
            clk, we         :     in std_logic;
            rdlsel, wdlsel  :     in std_logic_vector(l downto 0);
            wdata           :     in std_logic_vector(7 downto 0);
            rdlout          :     out std_logic_vector(7 downto 0));
end unknown;

architecture whatisthis of unknown is
```

```
     signal zero, one, two, three : std_logic_vector(7 downto 0);
begin
     clk_process: process (clk, one, two, zero, three, we, rdlsel,
     wdlsel, wdata)
     begin
          if rising_edge(clk) then
               if we = '1' then
                    if wdlsel = "00" then
                         zero <= wdata;
                    elsif wdlsel = "01" then
                         one <= wdata;
                    elsif wdlsel = "10" then
                         two <= wdata;
                    else
                         three <= wdata;
                    end if;
               end if;
          end if;
          case rdlsel is
               when "00"=> rdlout <= zero;
               when "01" => rdlout <= one;
               when "10" => rdlout <= two;
               when "11" => rdlout <= three;
               when others => rdlout <= "00000000";
          end case;
     end process clk_process;
end whatisthis;
```

Rešenje:

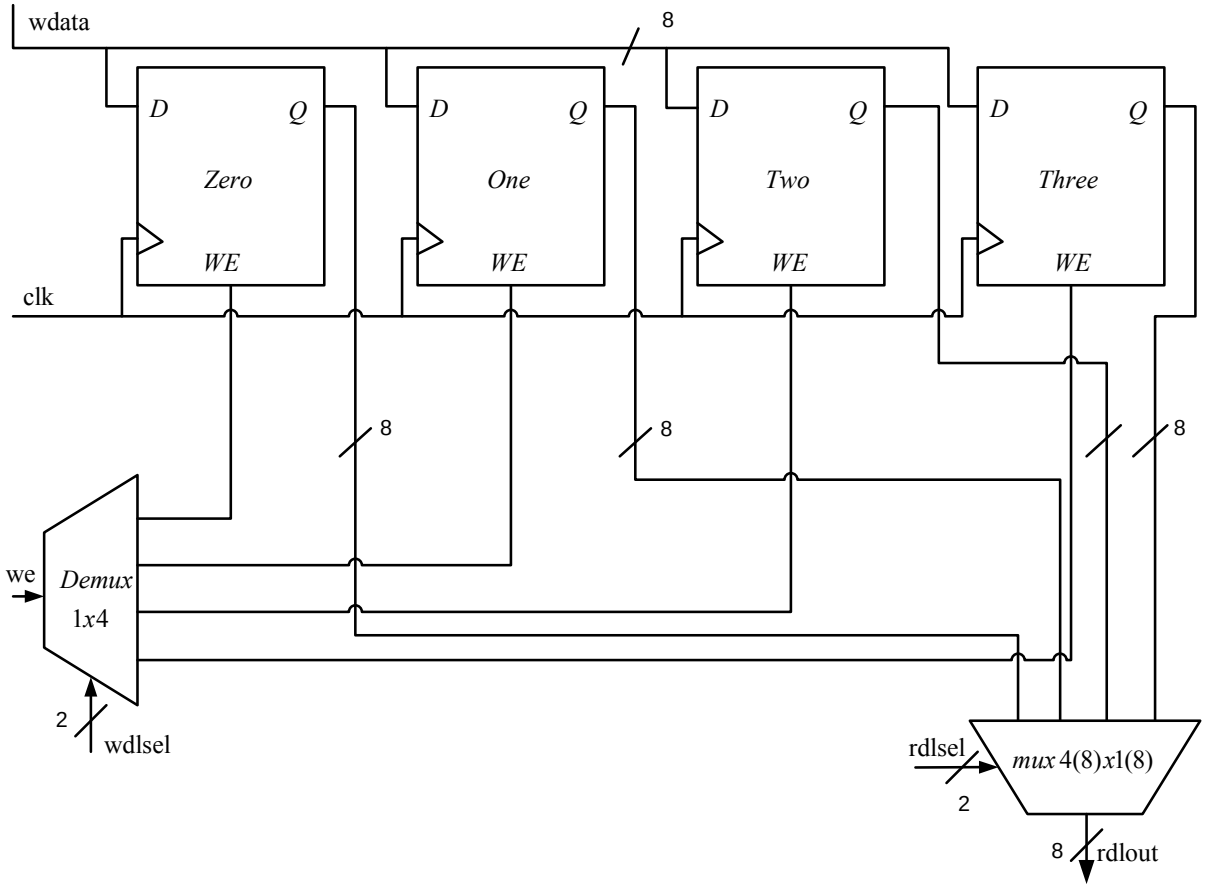
a) Na osnovu signala prisutnih u entity-ju Odd.vhd fajla može se zaključiti da postoje:

- osmobitna magistrala podataka wdata, kojom se podaci dovode na ulaz uređaja
- osmobitna magistrala podataka rdlout, kojom se podaci očitavaju sa izlaza uređaja
- dvobitni kontrolni signali rdlsel, wdlsel
- signal takta clk
- kontrolni signal we

Na osnovu opisa arhitekture whatisthis koja opisuje funkciju nepoznatog uređaja, može se zaključiti sledeće:

- U pitanju je sinhrona sekvencijalna mreža, koja sadrži četiri grupe memorijskih lokacija za upis podataka sa magistrale podataka wdata u zavisnosti od kontrolnog signala wdlsel, kojim se bira grupa memorijskih lokacija u koju se vrši upis. Upis je moguć samo ako je kontrolni signal we aktivan, pri čemu se upis vrši na uzlaznu ivicu signala takta clk. Svaka od četiri grupe memorijskih lokacija (zero, one, two, three) u principu predstavlja registar fajl, grupu od osam D flip-flopova čiji su D ulazi $D_0D_1.....D_7$ vezani za odgovarajuću liniju magistrale wdata. Svaki D flip-flop ima kontrolni ulaz na koji se dovodi signal we preko demultipleksera za odgovarajuću grupu od osam registara, pri čemu se demultiplekserom (1x4) upravlja kontrolnim signalima wdlsel.
- Iz uređaja se mogu očitavati prethodno upisani podaci sa magistrale podataka wdata i u zavisnosti od kontrolnih signala rdlsel, podaci iz odgovarajuće grupe memorijskih lokacija (zero, one, two, three) su prisutni na izlaznoj magistrali podataka rdlout.

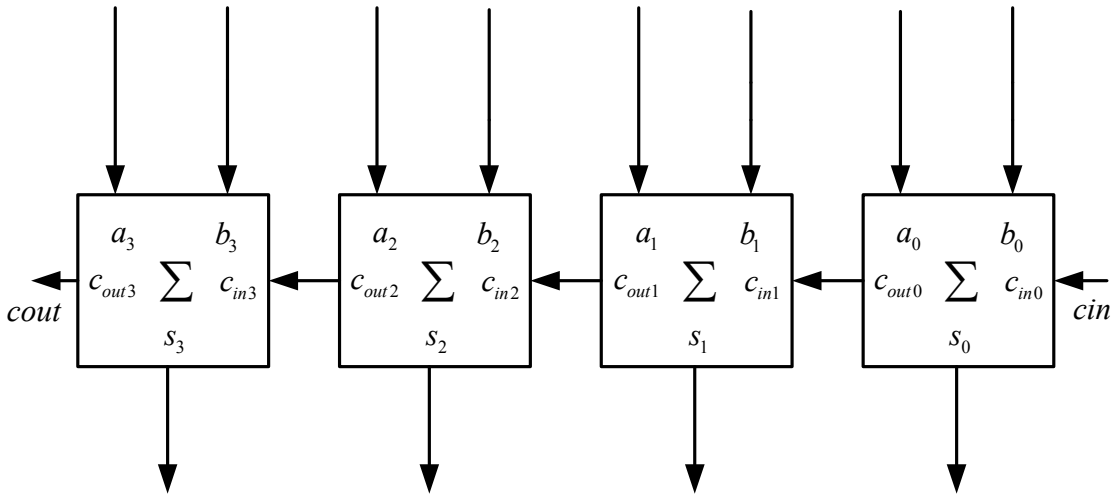
b) Blok dijagram koji opisuje strukturu hardverskog uređaja opisanog u Odd.vhd fajlu je prikazan na slici 12.25



Slika 12.25.

Zadatak 12.12.

„Carry ripple" i „carry look ahead" su dva klasična pristupa u dizajniranju sabirača. „Carry ripple" sabirač je jednostavniji (zahteva manje komponenata) za realizaciju, dok je „carry look ahead" sabirač brži, ali nešto komplikovaniji. Principska šema četvorobitnog „carry ripple" sabirača je prikazana na slici 12.26.



Slika 12.26.

Tabela logičke funkcije jedne ćelije „carry ripple" sabirača je prikazana na slici 12.27.

a_i	b_i	c_{ini}	s_i	c_{outi}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

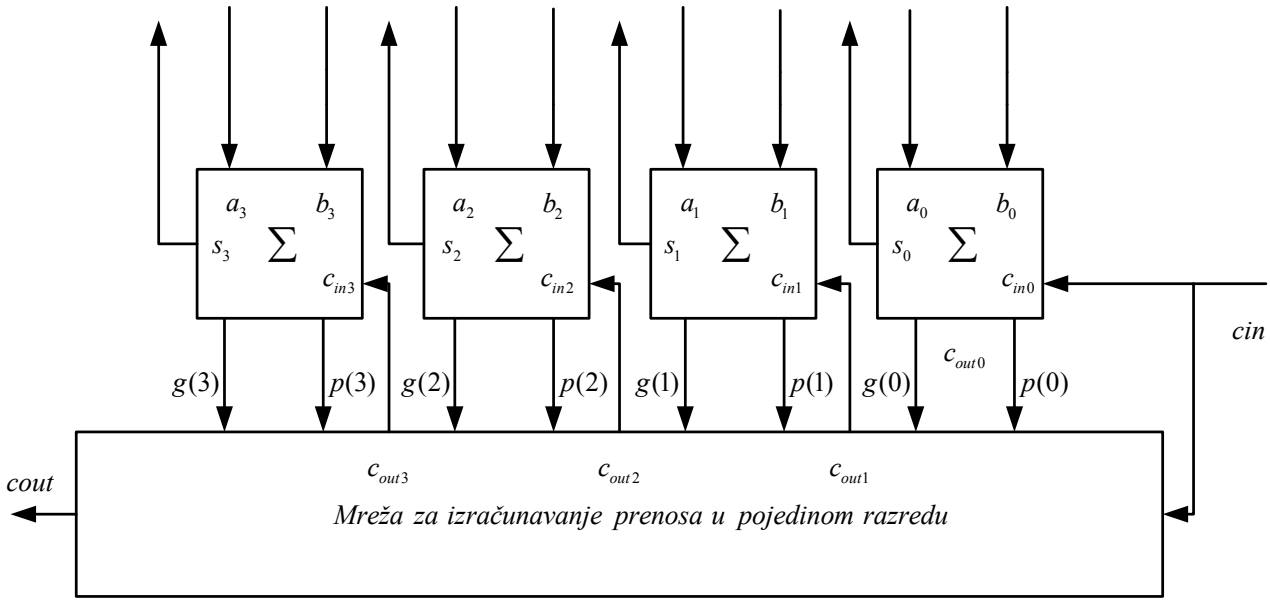
Slika 12.27.

Izrazi za s_i i c_{outi} dobijeni minimizacijom su:

$s_i = a_i \oplus b_i \oplus c_{ini}$

$c_{outi} = c_{ini} b_i + c_{ini} a_i + a_i b_i$

Principska šema četvorobitnog „carry look ahead" sabirača je prikazana na slici 12.28.



Slika 12.28.

Implementacija „carry look ahead" sabirača je bazirana na principu generisanja i propagacije (generate and propagate) prenosa, što omogućava veću brzinu sabiranja u odnosu na „carry ripple" sabirač. Logička funkcija i-te ćelije sabirača ostaje ista kao i u slučaju „carry ripple" sabirača, jedino što su uvedeni signali $g(i)$ i $p(i)$, koji treba da omoguće bržu propagaciju signala prenosa iz razreda u razred. Kod „carry ripple" sabirača izračunavanje zbira u poslednjem razredu je zavisilo od prisustva signala prenosa prethodnog razreda, odnosno tek nakon izračunavanja tog prenosa vrednost na izlazu poslednjeg razreda je bila korektna.

Prema tabeli prikazanoj na slici 12.27, može se zaključiti da se bez obzira na ulazni prenos bit izlaznog prenosa generiše ako su oba bita $a(i), b(i)$ jednaka 1, a ukoliko su biti $a(i), b(i)$ različiti, izlazni prenos će biti jednak ulaznom prenosu, tj. bit ulaznog prenosa se propagira kroz tu ćeliju sabirača. Na osnovu ovakvog razmatranja se mogu definisati signali $g(i)$ i $p(i)$ u i-toj ćeliji sabirača kao:

$$g(i) = a(i)b(i)$$
$$p(i) = a(i) \otimes b(i)$$

Kao što se vidi iz definicije ovih signala, oni ne zavise od signala prenosa i stoga se mogu izračunati na samom početku operacije sabiranja. Izlazni prenos u i-tom razredu će biti 1 ako je u prethodnom razredu bio aktivan signal $g(i\text{-}1)$ ("generiši" prenos) ili ako je u prethodnom razredu bio aktivan signal $p(i\text{-}1)$, a pri tom je ulazni prenos $c_{in}(i\text{-}1)$ bio 1. Dalje se može uspostaviti veza između prenosa u i-ti razred $c_{in\,i}$ i svih ostalih bita $a(j), b(j)$. Ta veza je data kao:

$$c_{in0} = cin$$
$$c_{in1} = cin\, p(0) + g(0)$$
$$c_{in2} = cin\, p(0)p(1) + g(0)p(1) + g(1)$$
$$c_{in3} = cin\, p(0)p(1)p(2) + g(0)p(1)p(2) + g(1)p(2) + g(2)$$
$$cout = cin\, p(0)p(1)p(2)p(3) + g(0)p(1)p(2)p(3) + g(1)p(2)p(3) + g(2)p(3) + g(3)$$

Na osnovu ovih izraza, a pošto signali $g(i)$ i $p(i)$ zavise samo od $a(i), b(i)$ može se zaključiti da se prenos u poslednjem razredu može izračunati nezavisno od računanja prenosa u prethodnim razredima, što predstavlja osnovnu prednost „carry look ahead" sabirača.

a) Izvršiti implementaciju četvorobitnog „carry ripple" sabirača u VHDL-u.

b) Izvršiti implementaciju četvorobitnog „carry look ahead" sabirača u VHDL-u.

Rešenje:

VHDL implementacija četvorobitnog „carry ripple" sabirača je data u nastavku,

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY adder_cripple IS
      GENERIC (n: INTEGER := 4);
      PORT ( a, b: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
             cin: IN STD_LOGIC;
              s: OUT STD_LOGIC_VECTOR (n-1 DOWNTO 0);
              cout: OUT STD_LOGIC);
END adder_cripple;

ARCHITECTURE adder OF adder_cripple IS
      SIGNAL c: STD_LOGIC_VECTOR (n DOWNTO 0);
BEGIN
      c(0) <= cin;
      G1: FOR i IN 0 TO n-1 GENERATE
            s(i) <= a(i) XOR b(i) XOR c(i);
            c(i+1) <=  (a(i) AND b(i)) OR
```

```
                    (a(i) AND c(i)) OR
                    (b(i) AND c(i));
     END GENERATE;
     cout <= c(n);
END adder;
```

Kao što se vidi iz koda, realizacija ovog sabirača je jednostavna i podrazumeva generisanje četiri identične ćelije.

VHDL implementacija četvorobitnog „carry look ahead" sabirača je data u nastavku.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY CLA_Adder IS
     PORT ( a, b: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
            cin: IN STD_LOGIC;
            s: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
            cout: OUT STD_LOGIC);
END CLA_Adder;

ARCHITECTURE CLA_Adder OF CLA_Adder IS
     SIGNAL c: STD_LOGIC_VECTOR (4 DOWNTO 0);
     SIGNAL p: STD_LOGIC_VECTOR (3 DOWNTO 0);
     SIGNAL g: STD_LOGIC_VECTOR (3 DOWNTO 0);
BEGIN
     G1: FOR i IN 0 TO 3 GENERATE
          p(i) <= a(i) XOR b(i);
          g(i) <= a(i) AND b(i);
          s(i) <= p(i) XOR c(i);
     END GENERATE;

     c(0) <= cin;
     c(1) <= (cin AND p(0)) OR
                g(0);
     c(2) <= (cin AND p(0) AND p(1)) OR
                (g(0) AND p(1)) OR
                g(1);
     c(3) <= (cin AND p(0) AND p(1) AND p(2)) OR
                (g(0) AND p(1) AND p(2)) OR
                (g(1) AND p(2)) OR
                g(2);
     c(4) <= (cin AND p(0) AND p(1) AND p(2) AND p(3)) OR
                (g(0) AND p(1) AND p(2) AND p(3)) OR
                (g(1) AND p(2) AND p(3)) OR
                (g(2) AND p(3)) OR
                g(3);
     cout <= c(4);
END CLA_Adder;
```

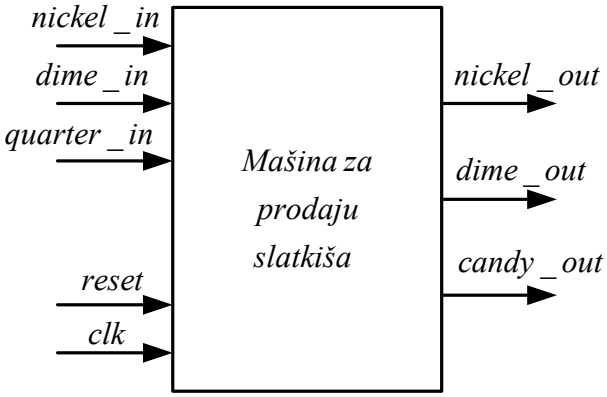
Zadatak 12.13.

Opisati u VHDL-u mašinu za prodavanje slatkiša. Mašina izdaje samo jednu vrstu slatkiša koja košta 25 poena. Mašina poseduje slotove (ulaze) na koje se mogu ubacivati metalni novčići u vrednostima od 5, 10 i 25 poena. Na izlazu mašina može izdati slatkiš (samo jedan komad u jednom ciklusu), odnosno kusur od ubačenog novca do vrednosti slatkiša od 25 poena. Za svaki

slot u koji se ubacuju novčići postoji po jedna interna promenjiva (nickel_in-za 5 poena, dime_in-za 10 poena, quarter_in-za 25 poena), koja je automatski postavljena od strane kontrolera slotova, prilikom ubacivanja novčića. Promenjiva pridružena slotu se postavlja na prvu nailazeću ivicu takta (uzlaznu ili silaznu) od momenta ubacivanja novčića i ima vrednost logičke jedinice u trajanju periode signala takta, posle čega se ponovo resetuje. Novčić trenutno prolazi kroz odgovarajući slot i završava u kasi mašine. Mašina izdaje slatkiš, odnosno kusur tako što postavlja promenljive candy_out odnosno nickel_out, dime_out na logičku jedinicu sa uzlaznom ivicom signala takta, pri čemu trajanje logičke jedinice iznosi jednu periodu signala takta. Smatrati da će mašina da izda slatkiš i odgovarajući kusur čim se u kasi pojavi iznos od bar 25 poena. Mašina poseduje signal reseta koji kada je aktivan $reset = 1$, zabranjuje rad mašine. Pored signala *reset* , unutar mašine se generiše i signal takta *clk* , koji omogućava sekvencijalni režim rada mašine.

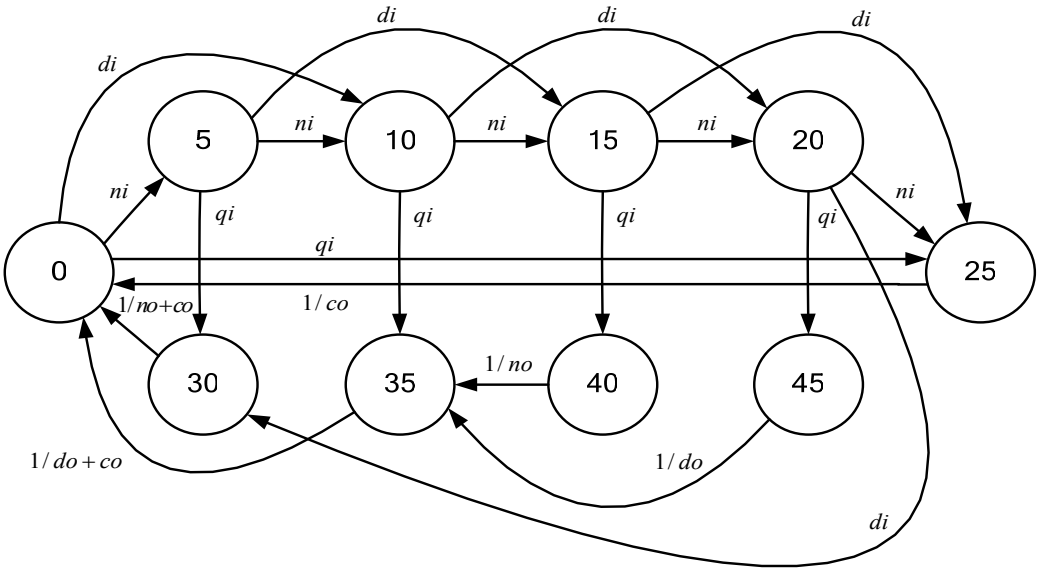
Rešenje:

Mašina za prodavanje slatkiša se može realizovati ako sinhrona sekvencijalna mreža koja se taktuje signalom *clk* . Blok dijagram mašine za prodaju slatkiša je prikazan na slici 12.29.



Slika 12.29.

Dijagram stanja FSM po kojoj funkcioniše mašina za prodaju slatkiša je prikazan na slici 12.30.



Slika 12.30.

Stanja FSM su označena prema ukupnoj sumi novca koja se nalazi u datom momentu u mašini za prodaju slatkiša, a koja je ubačena od strane jedne mušterije u jednom ciklusu izdavanja slatkiša. Prelazak u naredno stanje je uslovljen ubacivanjem odgovarajućeg novčića, a ukoliko na ulazu nema promena, FSM sa narednom pojavom signala takta ostaje u zatečenom stanju sve dok se ne ubaci još neki novčić (ovo važi za stanja 0, 5, 10, 15, 20). Iz stanja 25, 30, 35, 40, 45, u naredna stanja mašina prelazi sa pojavom rastuće ivice signala takta (nezavisno od ulaznih promenljivih), pri čemu se generišu odgovarajući signali za akciju koju mašina treba da izvrši (izdaje slatkiš, vraća kusur u odgovarajućim apoenima).

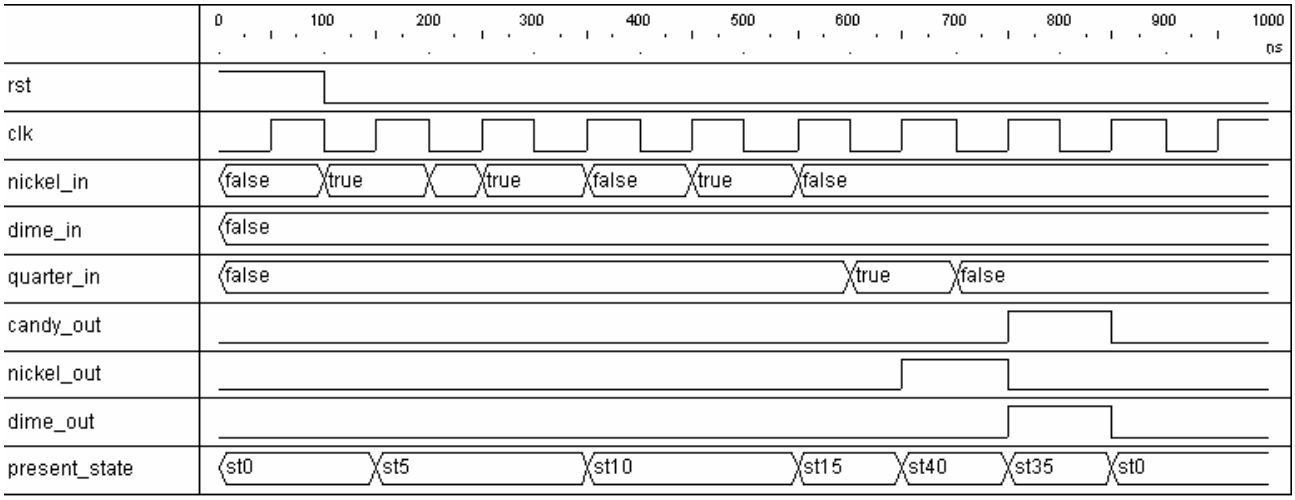
U nastavku je dat VHDL opis mašine za prodaju slatkiša, izveden na osnovu prethodne analize.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY vending_machine IS
      PORT ( clk, rst: IN STD_LOGIC;
               nickel_in, dime_in, quarter_in: IN BOOLEAN;
               candy_out, nickel_out, dime_out: OUT STD_LOGIC);
END vending_machine;

ARCHITECTURE fsm OF vending_machine IS
      TYPE state IS (st0, st5, st10, st15, st20, st25,
            st30, st35, st40, st45);
      SIGNAL present_state, next_state: STATE;
BEGIN

      PROCESS (rst, clk)
      BEGIN
            IF (rst='1') THEN
                  present_state <= st0;
            ELSIF (clk'EVENT AND clk='1') THEN
                  present_state <= next_state;
            END IF;
      END PROCESS;

      PROCESS (present_state, nickel_in, dime_in, quarter_in)
      BEGIN
            CASE present_state IS
                  WHEN st0 =>
                        candy_out <= '0';
                        nickel_out <= '0';
                        dime_out <= '0';
                        IF (nickel_in) THEN next_state <= st5;
                        ELSIF (dime_in) THEN next_state <= st10;
                        ELSIF (quarter_in) THEN next_state <= st25;
                        ELSE next_state <= st0;
                        END IF;
                  WHEN st5 =>
                        candy_out <= '0';
                        nickel_out <= '0';
                        dime_out <= '0';
                        IF (nickel_in) THEN next_state <= st10;
                        ELSIF (dime_in) THEN next_state <= st15;
                        ELSIF (quarter_in) THEN next_state <= st30;
                        ELSE next_state <= st5;
```

```
                    END IF;
                WHEN st10 =>
                    candy_out <= '0';
                    nickel_out <= '0';
                    dime_out <= '0';
                    IF (nickel_in) THEN next_state <= st15;
                    ELSIF (dime_in) THEN next_state <= st20;
                    ELSIF (quarter_in) THEN next_state <= st35;
                    ELSE next_state <= st10;
                    END IF;
                WHEN st15 =>
                    candy_out <= '0';
                    nickel_out <= '0';
                    dime_out <= '0';
                    IF (nickel_in) THEN next_state <= st20;
                    ELSIF (dime_in) THEN next_state <= st25;
                    ELSIF (quarter_in) THEN next_state <= st40;
                    ELSE next_state <= st15;
                    END IF;
                WHEN st20 =>
                    candy_out <= '0';
                    nickel_out <= '0';
                    dime_out <= '0';
                    IF (nickel_in) THEN next_state <= st25;
                    ELSIF (dime_in) THEN next_state <= st30;
                    ELSIF (quarter_in) THEN next_state <= st45;
                    ELSE next_state <= st20;
                    END IF;
                WHEN st25 =>
                    candy_out <= '1';
                    nickel_out <= '0';
                    dime_out <= '0';
                    next_state <= st0;
                WHEN st30 =>
                    candy_out <= '1';
                    nickel_out <= '1';
                    dime_out <= '0';
                    next_state <= st0;
                WHEN st35 =>
                    candy_out <= '1';
                    nickel_out <= '0';
                    dime_out <= '1';
                    next_state <= st0;
                WHEN st40 =>
                    candy_out <= '0';
                    nickel_out <= '1';
                    dime_out <= '0';
                    next_state <= st35;
                WHEN st45 =>
                    candy_out <= '0';
                    nickel_out <= '0';
                    dime_out <= '1';
                    next_state <= st35;
            END CASE;
        END PROCESS;
END fsm;
```

Stanja u kojima može da se nađe FSM su imenovana kao st0, st5, st10, st15, st20, st25,st30, st35, st40, st45 kojima odgovaraju trenutne vrednosti sume koja se nalazi u kasi mašine za slatkiše.

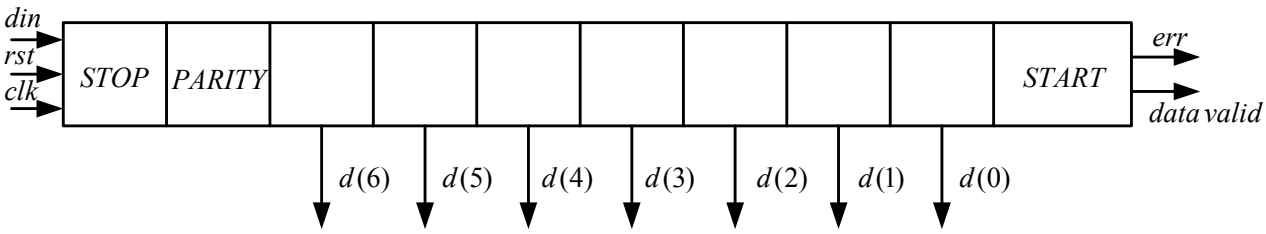
Na slici 12.31 su prikazani vremenski dijagrami simulacije izlaznih signala za jedan scenario signala na ulazu, gde je uzeto da signal takta bude periode $T = 100\,\text{ns}$.



Slika 12.31.

Zadatak 12.14.

Opisati u VHDL-u serijski prijemnik podataka. Blok šema serijskog prijemnika je prikazana na slici 12.32. Prijemnik ima ulaze *din*, *rst*, *clk* i izlaze $d(0)$, $d(1)$,..... $d(6)$ koji predstavljaju bite primljenog podatka i dalje se paralelno prosleđuju. Prijemnik takođe ima i izlaze *err* i *data valid*, koji ukazuju da li je došlo do greške u prijemu. Format poruke koja se šalje prijemniku je takav da se prvo šalje start bit koji kada je na logičkoj jedinici ukazuje prijemniku na početak prijema nove poruke. Sledećih sedam bita su biti podatka koji se šalje, i na kraju sledi bit parnosti koji je ”1” ako je broj jedinica u poslatoj poruci (računa se sedam bita podatka i sam bit parnosti, start i stop biti se ignorišu) neparan, inače je bit parnosti ”0”. Na kraju (posle poslednjeg bita poslatog podatka) šalje se stop bit (ukupno deseti bit od početka prijema), koji mora biti ”1” ako je slanje izvršeno korektno. Po ispravnom prijemu podaci smešteni u internom registru *reg*, se prosleđuju na izlaze $d(0)$, $d(1)$,..... $d(6)$ i promenjiva *data valid* postaje ”1”, dok *err* ostaje ”0”. U slučaju greške u prenosu *data valid* ostaje ”0”, dok *err* postaje ”1”. Kada otpočne prijem poruke (nailaskom start bita), na svaku narednu rastuću ivicu signala takta *clk*, očitava se ulaz *din* i interpretira kao naredni bit u poruci (prvi očitan bit podatka posle start bita je $d(0)$-LSB, poslednji očitan bit podatka je $d(6)$-MSB) sve dok se ne pojavi stop bit kada se okončava prijem poruke. Signal reseta *rst* briše interni registar, promenljive *err* i *data valid* se postavljaju na nulu. Dok je *rst* aktivan (logička jedinica) prijemnik ne prima poruke.



Slika 12.32.

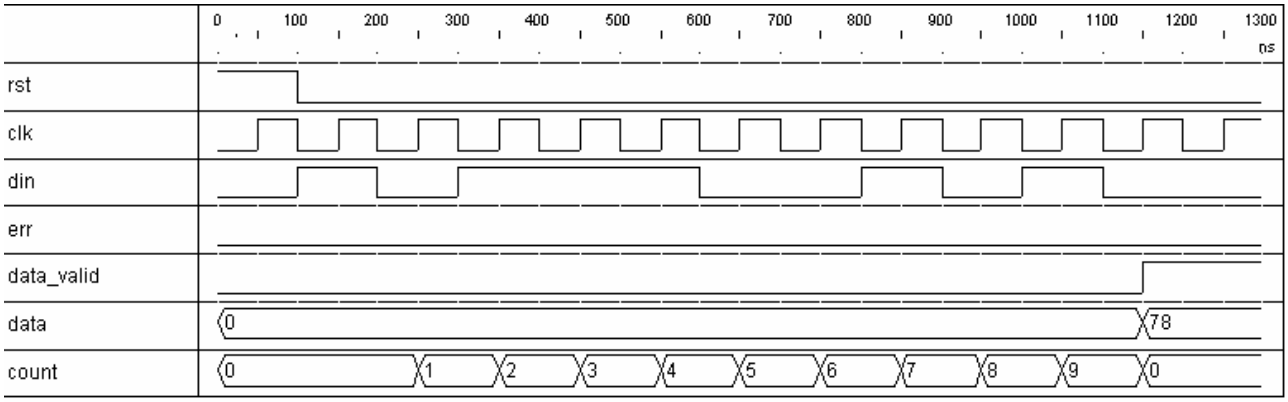
Rešenje:

VHDL kod koji opisuje prijemnik iz postavke zadatka je prikazan u nastavku.

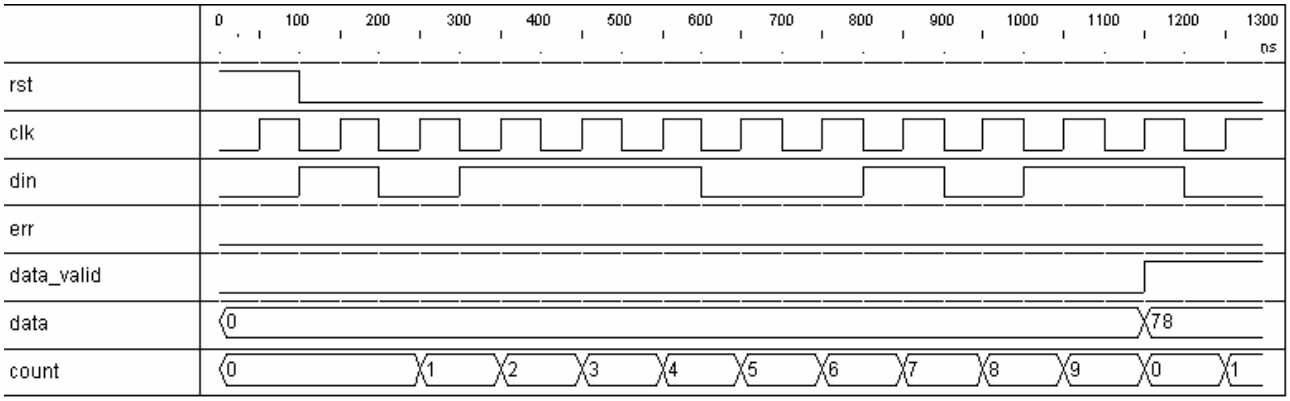
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY receiver IS
      PORT ( din, clk, rst: IN BIT;
             data: OUT BIT_VECTOR (6 DOWNTO 0);
             err, data_valid: OUT BIT);
END receiver;

ARCHITECTURE rtl OF receiver IS
BEGIN
      PROCESS (rst, clk)
            VARIABLE count: INTEGER RANGE 0 TO 10;
            VARIABLE reg: BIT_VECTOR (10 DOWNTO 0);
            VARIABLE temp : BIT;
      BEGIN
            IF (rst='1') THEN
                  count:=0;
                  reg := (reg'RANGE => '0');
                  temp := '0';
                  err <= '0';
                  data_valid <= '0';
            ELSIF (clk'EVENT AND clk='1') THEN
                  IF (reg(0)='0' AND din='1') THEN
                        reg(0) := '1';
                  ELSIF (reg(0)='1') THEN
                        count := count + 1;
                        IF (count < 10) THEN
                              reg(count) := din;
                        ELSIF (count = 10) THEN
                              temp := (reg(1) XOR reg(2) XOR reg(3) XOR
                                       reg(4) XOR reg(5) XOR reg(6) XOR
                                       reg(7) XOR reg(8)) OR NOT reg(9);
                              err <= temp;
                              count := 0;
                              reg(0) := din;
                              IF (temp = '0') THEN
                                    data_valid <= '1';
                                    data <= reg(7 DOWNTO 1);
                              END IF;
                        END IF;
                  END IF;
            END IF;
```

```
    END PROCESS;
END rtl;
```

Na slici 12.33 je prikazana simulacija rada prijemnika za poruku {start =1, din = 0111001, parity = 0, stop = 1}, gde din predstavlja broj 78 decimalno (jer je $d(6)$=1, $d(0)$=0). Na slici 12.34 je prikazana simulacija u slučaju da iza poruke sa brojem 78, odmah sledi naredna poruka (start bit=1). Uzeto je da signal takta bude periode $T = 100\,\text{ns}$.



Slika 12.33.



Slika 12.34.

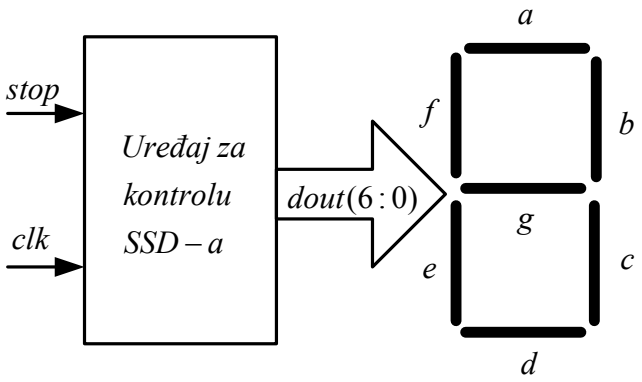
Zadatak 12.15.

Opisati u VHDL-u uređaj za kontrolu sedmosegmentnog LE displeja. Blok dijagram uređaja zajedno sa displejom je prikazan na slici 12.35. Ulazni signali su *clk* i *stop*, dok su izlazni signali $d(0)....d(6)$ koji pobuđuju odgovarajući segment SSD-a (sedmo-segmenti displej). Segmenti displeja treba da se pobuđuju tako da ostave utisak okretanja kazaljke na satu. Prvo se pali segment a, potom se on ugasi pa upali segment b, potom se on ugasi pa upali segment c itd...u krug. Kako bi utisak kruženja bio realističniji potrebno je kratkotrajno imati upaljena dva susedna segmenta, npr. ako je prvo bio upaljen segment a, kratko pre njegovog gašenja pali se i segment b (tako da sijaju zajedno), pa se onda gasi segment a i ostaje da svetli samo segment b. Prema tome sekvenca paljenja odgovarajućih segmenata displeja je sledeća:

$$a \rightarrow ab \rightarrow b \rightarrow bc \rightarrow c \rightarrow cd \rightarrow d \rightarrow de \rightarrow e \rightarrow ef \rightarrow f \rightarrow fa \rightarrow a.$$

Signal takta ima učestanost $f = 100\,\text{Hz}$. U stanjima $a,b,c.....$ sistem ostaje $time1 = 80\,\text{ms}$, dok u kombinovanim stanjima $ab,bc,cd.....$ sistem ostaje $time2 = 30\,\text{ms}$. Ako je signal *stop* aktivan

(na logičkoj jedinici) upaljen je samo segment *a* i sistem ostaje u tom stanju sve dok *stop* ne postane neaktivan (logička nula).

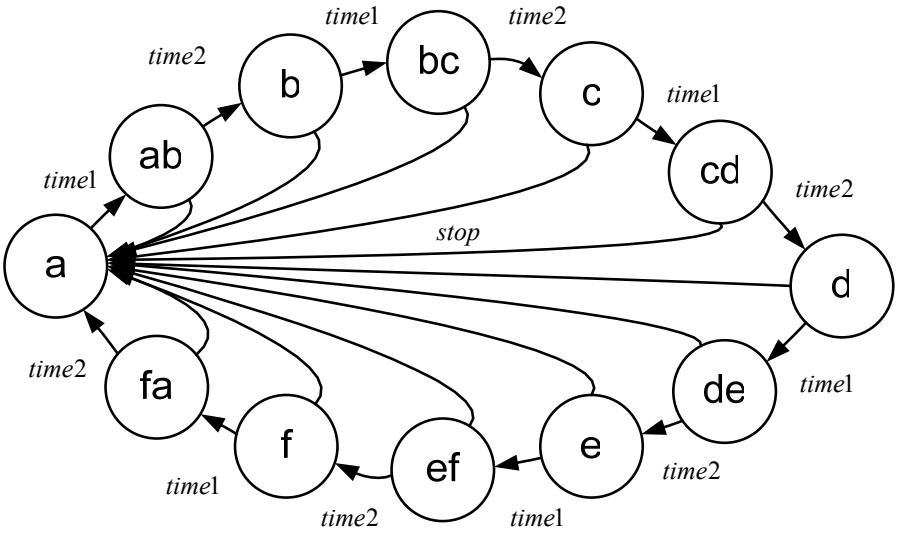


Slika 12.35.

Rešenje:

Pošto je u pitanju sekvencijalno paljenje segmenata SSD-a, kao prirodno rešenje u procesu VHDL dizajna nameće se FSM koncept. Dijagram prelaza FSM pridružene uređaju za kontrolu SSD-a je prikazan na slici 12.36.

VHDL kod je dat u nastavku. Entity uređaja nosi naziv ssd_control, usvojena je promenjiva flip koja govori o tome da li je FSM u stanju koje traje *time*1 ili *time*2. Na slici 12.37 su prikazani rezultati simulacije u slučaju da je promenjiva *stop* sve vreme neaktivna, dok su na slici 12.38 dati rezultati simulacije u slučaju da promenjiva *stop* u toku prelaska FSM iz jednog u drugo stanje kada se naizmenično pale segmenti SSD-a, promeni vrednost i postane $stop = 1$. Slika 12.39 pokazuje rezultate simulacije kada promenjiva *stop* u toku rada kontrolera SSD-a postane aktivna i nakon nekog vremena ponovo uzme vrednost nula. Kao što se vidi sa poslednjeg dijagrama promenjiva count nastavlja da broji tamo gde je zatečena pojavom signala stop. Dodavanjem count <= 0 posle linije present_state <= a, pošto se pojavi promenjiva stop, može se obezbediti da count ponovo počne od nule sa brojanjem.

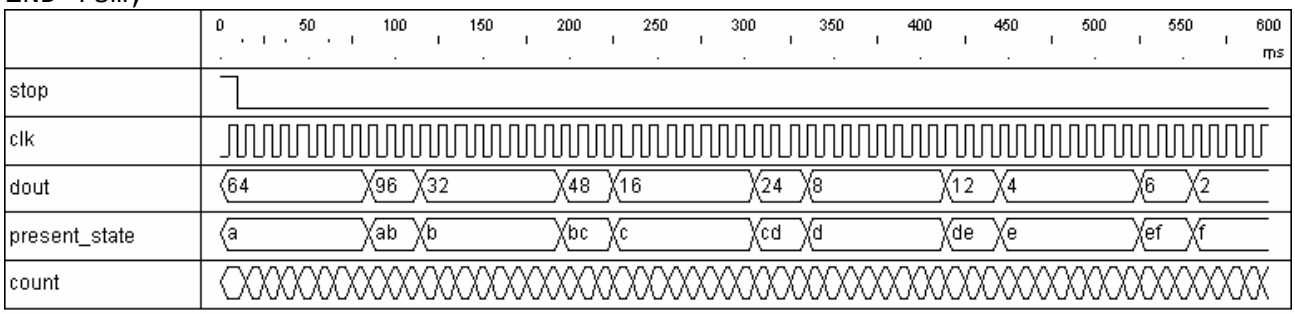


Slika 12.36.

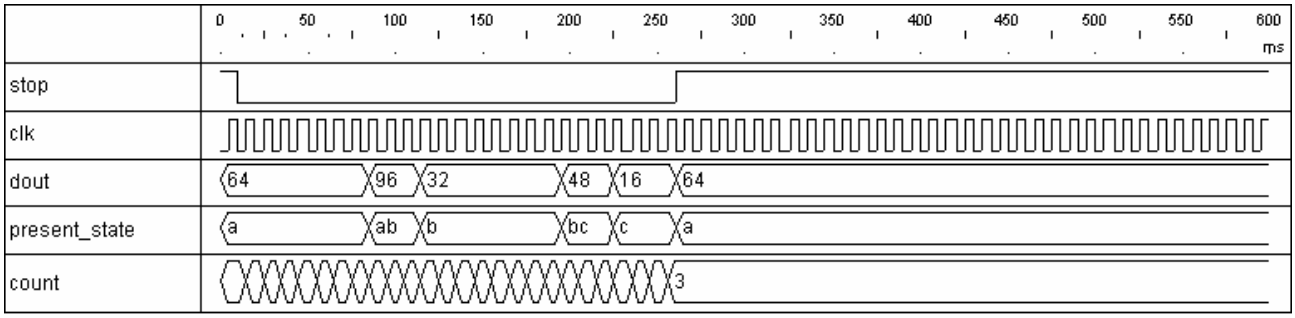
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY ssd_control IS
     PORT ( clk, stop: IN BIT;
```

```
            dout: OUT BIT_VECTOR (6 DOWNTO 0));
END ssd_control;

ARCHITECTURE fsm OF ssd_control IS
     CONSTANT time1: INTEGER := 7; -- vreme 80ms
     CONSTANT time2: INTEGER := 2; -- vreme 30ms
     TYPE states IS (a, ab, b, bc, c, cd, d, de, e, ef, f, fa);
     SIGNAL present_state, next_state: STATES;
     SIGNAL count: INTEGER RANGE 0 TO 7;
     SIGNAL flip: BIT;
BEGIN

     PROCESS (clk, stop)
     BEGIN
          IF (stop='1') THEN
               present_state <= a;
          ELSIF (clk'EVENT AND clk='1') THEN
               IF ((flip='1' AND count=time1) OR
                  (flip='0' AND count=time2)) THEN
                   count <= 0;
                   present_state <= next_state;
               ELSE count <= count + 1;
               END IF;
          END IF;
     END PROCESS;
     PROCESS (present_state)
     BEGIN
          CASE present_state IS
               WHEN a =>
                    dout <= "1000000"; -- Decimalno 64
                    flip<='1';
                    next_state <= ab;
               WHEN ab =>
                    dout <= "1100000"; -- Decimalno 96
                    flip<='0';
                    next_state <= b;
               WHEN b =>
                    dout <= "0100000"; -- Decimalno 32
                    flip<='1';
                    next_state <= bc;
               WHEN bc =>
                    dout <= "0110000"; -- Decimalno 48
                    flip<='0';
                    next_state <= c;
               WHEN c =>
                    dout <= "0010000"; -- Decimalno 16
                    flip<='1';
                    next_state <= cd;
               WHEN cd =>
                    dout <= "0011000"; -- Decimalno 24
                    flip<='0';
                    next_state <= d;
               WHEN d =>
                    dout <= "0001000"; -- Decimalno 8
                    flip<='1';
                    next_state <= de;
               WHEN de =>
                    dout <= "0001100"; -- Decimalno 12
```

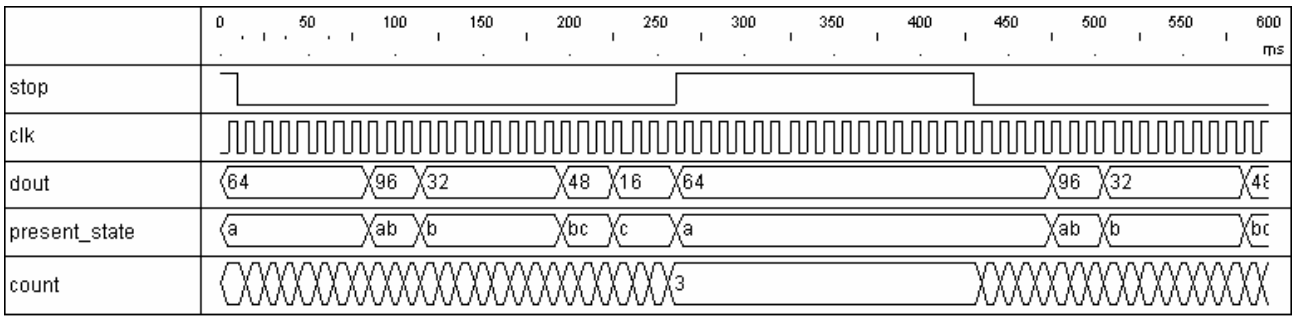
```
                    flip<='0';
                    next_state <= e;
               WHEN e =>
                    dout <= "0000100"; -- Decimalno 4
                    flip<='1';
                    next_state <= ef;
               WHEN ef =>
                    dout <= "0000110"; -- Decimalno 6
                    flip<='0';
                    next_state <= f;
               WHEN f =>
                    dout <= "0000010"; -- Decimalno 2
                    flip<='1';
                    next_state <= fa;
               WHEN fa =>
                    dout <= "1000010"; -- Decimalno 66
                    flip<='0';
                    next_state <= a;
          END CASE;
     END PROCESS;
END fsm;
```



Slika 12.37.



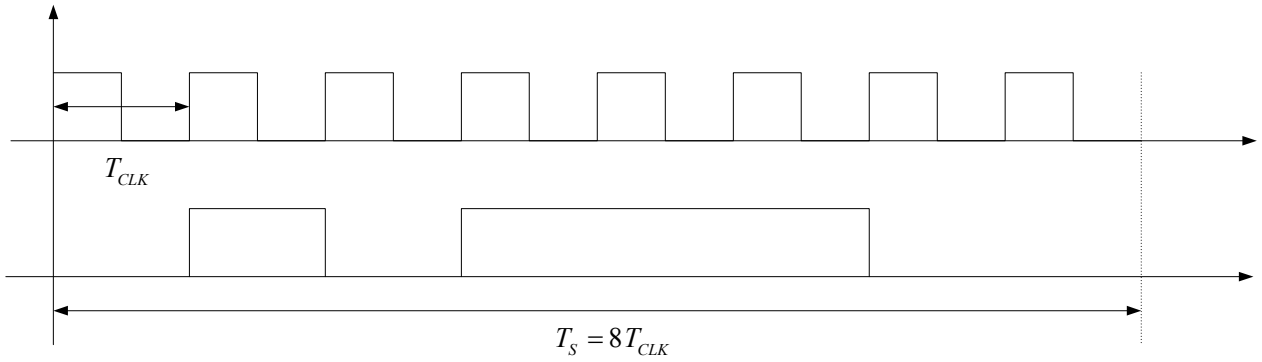
Slika 12.38.



Slika 12.39.

Zadatak 12.16.

Na slici 12.40 prikazan je talasni oblik signala koji se periodično ponavlja sa periodom od $T_S = 8T_{CLK}$. Potrebno je opisati u VHDL-u hardverski uređaj koji generiše posmatrani talasni oblik na svom izlazu. Na ulaz uređaja se dovodi signal takta periode T_{CLK}.



Slika 12.40.

Rešenje:

Jedno rešenje je opisati traženi uređaj kao FSM koja će imati osam stanja i u svakom stanju na izlazu dati odgovarajuću vrednost traženog signala tako da se u vremenu $T_S = 8T_{CLK}$, izlaz uređaja poklopi sa vremenskim dijagramom kojim je zadat signal koji treba da se generiše.

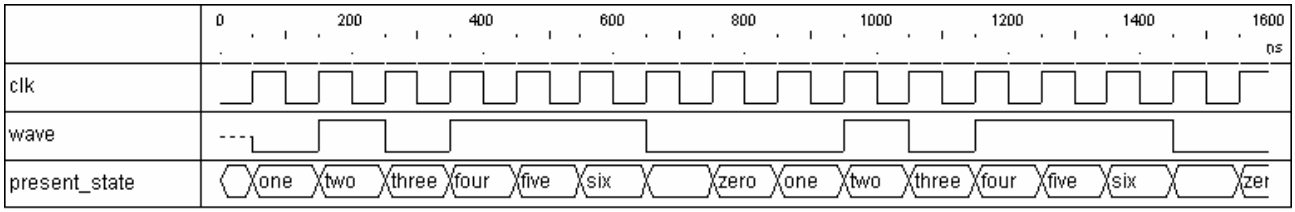
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY signal_gen IS
      PORT (clk: IN STD_LOGIC;
            wave: OUT STD_LOGIC);
END signal_gen;

ARCHITECTURE fsm OF signal_gen IS
      TYPE states IS (zero, one, two, three, four, five, six,
                      seven);
      SIGNAL present_state, next_state: STATES;
      SIGNAL temp: STD_LOGIC;
BEGIN

      PROCESS (clk)
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  present_state <= next_state;
                  wave <= temp;
            END IF;
      END PROCESS;

      PROCESS (present_state)
      BEGIN
      CASE present_state IS
            WHEN zero => temp<='0'; next_state <= one;
            WHEN one => temp<='1'; next_state <= two;
            WHEN two => temp<='0'; next_state <= three;
```

```
            WHEN three => temp<='1'; next_state <= four;
            WHEN four => temp<='1'; next_state <= five;
            WHEN five => temp<='1'; next_state <= six;
            WHEN six => temp<='0'; next_state <= seven;
            WHEN seven => temp<='0'; next_state <= zero;
      END CASE;
      END PROCESS;
END fsm;
```

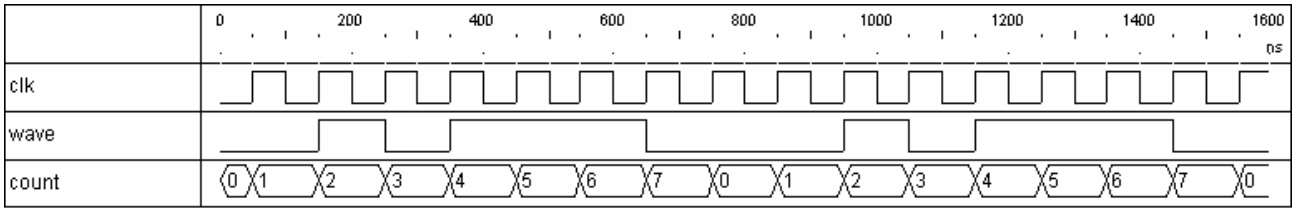
Drugo rešenje je opisati traženi uređaj korišćenjem IF strukture i brojanjem uzlaznih ivica signala takta, kao što je to pokazano u nastavku.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY signal_gen1 IS
      PORT (clk: IN BIT;
            wave: OUT BIT);
END signal_gen1;

ARCHITECTURE arch1 OF signal_gen1 IS
BEGIN
      PROCESS
            VARIABLE count: INTEGER RANGE 0 TO 7;
      BEGIN
            WAIT UNTIL (clk'EVENT AND clk='1');
            CASE count IS
                  WHEN 0 => wave <= '0';
                  WHEN 1 => wave <= '1';
                  WHEN 2 => wave <= '0';
                  WHEN 3 => wave <= '1';
                  WHEN 4 => wave <= '1';
                  WHEN 5 => wave <= '1';
                  WHEN 6 => wave <= '0';
                  WHEN 7 => wave <= '0';
            END CASE;
            IF count=7 THEN
                  count:=0;
            ELSE
                  count := count + 1;
            END IF;
      END PROCESS;
END arch1;
```

Rezultati simulacije u slučaju opisa hardvera kao FSM, dati su na slici 12.41, dok u slučaju korišćenja IF strukture, rezultati su prikazani na slici 12.42. Uzet je signal takta sa periodom $T_{CLK} = 100\,\text{ns}$.



Slika 12.41.


0
200
400
600
800
1000
1200
1400
1600
ns
clk
wave
count
0
1
2
3
4
5
6
7
0
1
2
3
4
5
6
7
0

Slika 12.42.