VHDL

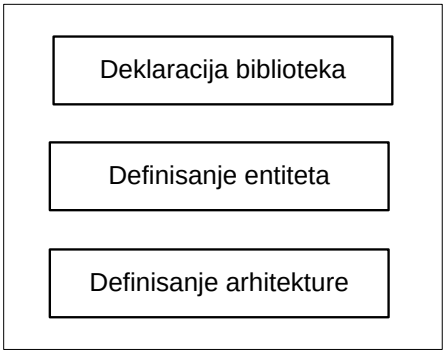
Uvod

VHDL je jezik koji služi za opisivanje funkcionisanja hardvera, kao i za dizajn digitalnih elektronskih sistema, poput osnovnih digitalnih kola, korisnički programabilnih logičkih nizova (engl. Field Programmable Gate Array - FPGA) i kola sa specijalnom namjenom (engl. Application-Specific Integrated Circuit - ASIC). VHDL jezik može da se koristi u svim fazama projektovanja integrisanih kola: opisu, simulaciji i sintezi integrisanih kola. Sistem koji se razvija u VHDL-u, može se fizički implementirati na FPGA, ASIC ili CPLD čipovima. VHDL je skraćenica engleskih riječi *VHSIC Hardware Description Language,* gdje se pod VHSIC (engl. Very High Speed Integrated Circuit) smatra integrisano kolo vrlo velike brzine. Složenost digitalnih sistema koji se modeluju može varirati od jednostavnog logičkog kola pa sve do kompletnog digitalnog sistema. Osim VHDL-a, za opis ponašanja digitalnog elektronskog kola (sistema) mogu se koristiti i ABEL, AHDL, Verilog i slično.

VHDL je u suštini paralelan jezik, a ne samo strukturalan kao što je recimo Pascal ili objektno orijentisan kao što je C++. Pod pojmom paralelan se podrazumijeva da se elementi VHDL programa u izvršavaju paralelno (istovremeno), za razliku od strukturalnih programskih jezika gdje se program izvršava po principu “naredba po naredba”. Naredbe u okviru procesa, funkcija i procedura u VHDL-u se izvršavaju sekvencijalno, ali će o tome biti riječi kasnije.

Struktura VHDL fajla

Osnovni djelovi svakog fajla koji sadrži VHDL kod kojim se opisuje neko digitalno kolo, sastoji se od entiteta i arhitekture. Entitet opisuje osnovne ulaze i izlaze u/iz kola, dok arhitektura opisuje funkcionalnost datog kola, tj. pretvaranje datih ulaza u odgovarajuće izlaze. Pored ova dva dijela, postoji i područje za deklarisanje biblioteka, koje sadrže skup često korišćenih funkcija. Ukoliko se neka funkcija smjesti u biblioteku, omogućeno je njeno ponovno korišćenje neposrednim pozivanjem funkcije. Opšta struktura VHDL fajla prikazana je na Slici 1.



Slika 1. Opšta struktura VHDL fajla

Deklaracija biblioteka

Biblioteke u VHDL-u predstavljaju skupove naredbi, funkcija, definisanih tipova podataka i operacija koje omogućavaju jednostavniju upotrebu jezika za projektovanje digitalnih kola i sistema.

Za deklaraciju biblioteka potrebne su dvije linije koda: prva koja služi za naziv biblioteke, a druga koja pomoću USE naredbe omogućava korišćenje sadržaja navedene biblioteke. Opšta struktura za pozivanje i uključivanje neke biblioteke data je sljedećim kodom:

```
LIBRARY naziv_biblioteke;
USE naziv_biblioteke.naziv_paketa.dio_paketa;
```

Većina biblioteka sastoji se iz velikog broja paketa, tako da je potrebno dodatno specificirati koji paket ili koji dio paketa se koristi.

Tri najčešće korišćene biblioteke su **ieee**, **standard** i **work** biblioteka. Biblioteke **standard** i **work** su automatski uključene u svakom VHDL projektu, tako da se ne moraju eksplicitno naznačiti. Biblioteka **ieee** se koristi samo onda kada se koristi STD_LOGIC vrsta podataka.

Primjeri deklarisanja tri najčešće korišćene biblioteke dati su sljedećim kodovima:

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

LIBRARY std;
USE std.standard.all;

LIBRARY work;
USE work.all;
```

Primijetimo da su za biblioteke **ieee** i **std** specificirani paketi **std_logic_1164** i **standard**, respektivno. S obzirom da je **work** radna biblioteka (sadrži sve elemente sa kojima se radi u okviru jednog projekta), u okviru nje su uključeni svi prateći paketi.

Definisanje entiteta

Entitet predstavlja spoljašnji dio kola koje se projektuje, odnosno definisanje entiteta služi za specificiranje ulaznih i izlaznih signala kola koje se projektuje. Opšti oblik deklaracije entiteta prikazan je u nastavku.

```
ENTITY ime_entiteta IS

      PORT (ime_signala: [režim rada] ime_tipa ;
            ime_signala: [režim rada] ime_tipa  );

END ime_entiteta;
```

Ključna riječ **ENTITY** označava da je započeto deklarisanje entiteta, zatim slijedi naziv entiteta (koji korisnik sam zadaje), dok ključna riječ **IS** označava početak definisanja ulaznih i izlaznih signala za dati entitet. Pomoću ključne riječi **PORT** definišu se ulazni i izlazni portovi signala tako što se najprije zada ime signala (koje može bilo koje validno VHDL ime, odnosno identifikator), zatim se zadaje režim rada porta i na kraju tip podatka koje taj port koristi.

Pravila za davanje imena signala i ostalih promjenljivih u VHDL kodu, kao i tipovi podataka koji mogu da se koriste u VHDL-u opisani su u narednom poglavlju.

Režim rada porta određuje njegovu ulogu u funkcionisanju kola i sistema koji se projektuje. U zavisnosti od namjene režim rada porta može biti: **ulazni, izlazni, ulazno-izlazni** i **bufer**. Karakteristike navedenih režima rada portova date su u Tabeli 1.

TABELA 1. REŽIMI RADA PORTOVA

Režim rada	**Namjena**
IN	Označava signal koji je ulaz u dato kolo.
OUT	Koristi se da označi signal koji je izlaz iz kola. Vrijednost ovog signala se ne može koristiti unutar entiteta, što znači da se u izrazima za dodjelu vrijednosti ovaj signal može naći samo sa lijeve strane operatora dodjele <=.
INOUT	Označava signal koji istovremeno može biti i ulaz i izlaz datog kola.
BUFFER	Predstavlja izlazni signal kola, ali se njegove vrijednosti mogu koristiti unutar kola. To znači da se u izrazima dodjele može pojaviti sa obje strane operatora <=.

Primjer: Napisati deklaraciju eniteta za logičko ILI kolo prikazano na Slici 2a. Kolo ima dva ulaza i jedan izlaz. Svi ulazni signali su tipa BIT.

a)

```
entity ili2 is

            port ( A, B    : in        BIT;
                 Z         :      out   BIT );
end ili2;
```

b)

Slika 2. a) Logičko ILI kolo, b) Deklaracija entiteta za logičko ILI kolo

Komentar:

- Naziv datog kola (entiteta) je "ili2", što je definisano pomoću ključne riječi **entity**.
- Pomoću ključne rijeći **port** deklarisani su ulazni portovi (**in**) A i B, i izlazni port Z (**out**). Svi portovi su tipa **BIT**, što znači da mogu imati samo jednu i to binarnu vrijednost.
- Definisanje entiteta završava se ključnom riječju **end** i nazivom entiteta.

Definisanje arhitekture

Arhitektura kola opisaje način rada i definiše funkciju samog kola, odnosno vrši funkcionalno povezivanje ulaza i izlaza kola. Opšta struktura definisanja arhitekture data je u nastavku:

```
ARCHITECTURE ime_arhitekture OF ime_entiteta IS
     [SIGNAL deklaracije]
     [CONSTANT deklaracije]
     [TYPE deklaracije]
     [COMPONENT deklaracije]
     [ATTRIBUTE specifikacije]
BEGIN
     {COMPONENT iskazi;}
     {CONCURENT ASSIGNMNET iskazi;}
     {PROCESS iskazi;}
     {GENERATE iskazi;}
END [ime_arhitekture];
```

Pomoću ključne riječi **ARCHITECTURE** najprije se zadaje ime arhitekture za odgovarajući entitet (**OF** ime_entiteta), a zatim se nakon ključne riječi **IS** opciono vrše deklaracije. Ključnom riječju **BEGIN** počinje definisanje tijela arhitekture, a definisanje arhitekture završava se ključnom riječju **END** nakon koje slijedi naziv arhitekture.

Definisanje arhitekture se sastoji iz dva glavna dijela: **oblast deklaracije** i **tijelo arhitekture**. Oblast deklaracije može se opciono koristiti i njeno postojanje nije obavezno pri definisanju arhitekture. Oblast deklaracije javlja se prije ključne riječi BEGIN i može se koristiti za deklarisanje pomoćnih signala (SIGNAL), konstanti (CONSTANT), korisnički definisanih tipova podataka (TYPE), komponenti (COMPONENT) i atributa (ATTRIBUTE).

Ključna riječ BEGIN označava početak tijela arhitekture, odnosno funkcionalnog dijela samog kola. U okviru tijela athitekture definišu se iskazi na osnovu kojih se izvršavaju određene logičke ili aritmetičke operacije. U prethodno navedenom kodu je dato više načina za definisanje tijela arhitekture.

Primjer: Za logičko ILI kolo prikazano na Slici 2a definisati arhitekturu koja će obavljati logičku ili funkciju $Z = A + B$.

Rješenje:

```
architecture ili2_arh of ili2 is
begin
            Z <= A or B;
end ili2_arh;
```

Komentar:

- U navedenom kodu naziv arhitekture (ključna riječ **architecture**) je "ili2_arh" i odnosi se na entitet "ili2" (**of** ili2 **is**).
- Oblast deklaracije nije korišćena.
- Tijelo arhitekture (počinje ključnom riječju **begin**) sastoji se od jednog iskaza kojim je definisana operacija logičko ILI (Z <= A or B).
- Definisanje arhitekture završava je naredbom **end** iza koje slijedi naziv arhitekture.

Osnovni elementi VHDL jezika

Osnovni elementi VHDL jezika su:

- Identifikatori,
- Objekti podataka,
- Tipovi podataka,
- Operatori.

Identifikatori

U VHDL jeziku identifikator se sastoji od niza jednog ili više karaktera. Dozvoljeni karakteri su:

- velika i mala slova (od A do Z i od a do z),
- brojevi od 0 do 9,
- podvlaka (_).

Prvi karakter mora biti slovo, a zadnji ne može biti podvlaka. VHDL jezik **NIJE *case sensitive***, a to znači da ne pravi razliku između velikih i malih slova u identifikatoru. Npr.:

- ADDER
- Adder
- addER

su isti identifikator. Također, dvije podvlake ne mogu stajati uzastopno. Nekoliko primjera pravilnih identifikatora:

- FULL_ADDER
- Mux2_4
- My_Integer
- RD16

Komentari u opisu moraju biti naznačeni sa dvije uzastopne crte (--) i oni se navode na kraju linije. Ukoliko se komentar nastavlja u drugoj liniji neophodno je navesti dvije uzastopne crte prije nastavka.

Primjer komentara:

```
signal X: BIT; -- signal X je deklarisan kao jednobitni signal tipa BIT
```

U VHDL-u postoji niz rezervisanih riječi, takozvanih *keywords*, koje imaju posebno značenje u jeziku pa zbog toga ne mogu biti upotrijebljeni kao identifikatori.

Objekti podataka

Postoje tri vrste objekata koje se mogu definisati u VHDL-u:

- Konstante - imenuju određene vrijednosti.
- Promjenjive - koriste se za lokalno smještanje privremenih podataka.
- Signali - povezuju portove/pinove komponenti.

Konstante

Konstante sadrže samo jednu vrijednost datog tipa podataka i ne mogu se mijenjati tokom izvršavanja koda. Konstante se upotrebljavaju u programu ili modelu kada se jedna vrijednost koristi više puta. Mijenjanje neke vrijednosti u programu je tako dosta olakšano, jer se vrijednost mijenja na jednom mjestu.

Konstante se deklarišu na sljedeći način:

```
CONSTANT naziv_konstante : vrsta_konstante [:= vrijednost];
```

Primjer deklaracije konstante : `CONSTANT a : INTEGER := 5 ;`

Promjenjive

Promjenjive se deklarišu na sljedeći način:

```
VARIABLE naziv_varijable , naziv_varijable : vrsta_promjenjive [:=
vrijednost];
```

Primjeri deklaracije promjenljivih:

```
VARIABLE a,b : INTEGER ;
VARIABLE a,b,c : INTEGER := 0;
```

Nakon ključne riječi **VARIABLE** moguće je definisati više promjenjivih razdvojenih zarezom. Vrsta promjenjive kao što i ime kaže definiše vrstu vrijednosti definisanu promjenjivom. Moguće vrijednosti u polju "vrsta promjenjive" biće predstavljene u dijelu «tipovi podataka » koji slijedi.

Signali

Opšta naredba za deklarisanje signala data je:

```
SIGNAL naziv_signala : vrsta_signala [:= inicijalna_vrijednost];
```

Nakon ključne riječi SIGNAL može se definisati više signala razdvojenih zarezom. Vrsta signala definiše tip podataka koji se koristi za navedeni signal.

Signali se mogu definisati prilikom definisanja entiteta, arhitekture i paketa. Ukoliko se vrijednost signala promijeni u toku procesa, signal će tu vrijednost poprimiti tek po završetku procesa, što znači da se u toku trenutnog procesa ne može koristiti, za razliku od varijable, koja datu vrijednost poprima odmah.

Primjeri deklaracije konstanti, promjenljivih i signala:

```
constant B: INTEGER :=7; -- deklarisana je cjelobrojna konstanta B koja ima vrijednost 7

variable SUM: BIT_VECTOR(0 to 7); -- deklarisana je promjenljiva SUM koja je tipa BIT_VECTOR od 8 elemenata

signal W: BIT_VECTOR(3 downto 0); -- deklarisan je signal W koji je tipa BIT_VECTOR, od 4 elementa
```

Tipovi podataka

Da bi napisali VHDL kod potrebno je znati koji su dozvoljeni tipovi podataka, kako ih specificirati i koristiti. Svaki objekat podataka u VHDL-u sadrži vrijednost koja pripada nekom nizu vrijednosti, a taj niz je specificiran korišćenjem deklaracije tipa. Određeni tipovi i operacije koje se mogu sprovesti nad objektom su definisani u samom jeziku. Deklaracija unaprijed definisanih tipova podataka nalazi se u paketu STANDARD i operatori dozvoljeni nad ovim tipovima su definisani u samom jeziku. Deklaracija unaprijed definisanih tipova podataka nalazi se u bibliotekama poput biblioteke **std** u paketu standard koja definišu tipove podataka: **BIT, BOOLEAN, INTEGER** i **REAL**, biblioteke **ieee** u paketu std_logic_1164, koja definiše tipove **STD_LOGIC** i **STD_LOGIC_VECTOR**.

STD_LOGIC ima sljedeće moguće vrijednosti:

'U' --> Uninitialized - - ako nema specificirane početne vrijednosti
'X' --> Forcing Unknown - - 'X' javlja se kada se ne može donijeti odluka da li je “0” ili ”1”
'0' --> Forcing 0;
'1' --> Forcing 1;
'Z' --> High Impedance
'W' --> Weak Unknown
'L' --> Weak 0;
'H' --> Weak 1;
'-' --> Don't care;

Primjeri tipova podataka i dodjele vrijednosti:

- Dodjeljivanje cjelobrojne, tj. INTEGER vrijednosti:

  `VARIABLE x: INTEGER;` - definisana je promjenljiva *x,* koja uzima cjelobrojne vrijednosti

  `x := 5;` - može joj se dodijeliti pozitivna vrijednost

  `x := -6;` - može joj se dodijeliti negativna vrijednost

  `x := 2.55;` - greška, broj koji je dodijeljen je realni broj, a promjenljiva može uzeti samo vrijednost iz skupa cijelih brojeva INTEGER

- Dodjeljivanje realne, tj. REAL vrijednosti:

  `SIGNAL x: REAL;` - definisan je signal (može biti definisana i promjenljiva) *x*, koji uzima vrijednosti iz skupa realnih brojeva - REAL

  ```
  x <= 5.0;
  x <= -6; -- greška, dodijeljena je cjelobrojna vrijednost, a signalu se može dodijeliti samo realna vrijednost
  x <= 3.0E10;
  ```

```
x <= 7.5E-20;
x <= 12.0 ns; -- greška
```

Jedina predefinisana vrijednost sa mjernom jedinicom u VHDL-u je vrijeme (TIME). Deklaracija promjenljive tipa mjernih jedinica i dozvoljenih vrijednosti su date u nastavku:

```
TYPE TIME IS RANGE <implementation defined>
UNITS
ps = 1000 fs; -- pikosekund
ns = 1000 ps; -- nanosekund
us = 1000 ns; -- mikrosekund
ms = 1000 us; -- milisekund
sec = 1000 ms; -- sekund
min = 60 sec; -- minut
hr = 60 min; -- sat
END UNITS;
```

U VHDL jeziku postoji mogućnost definisanja **novog tipa podataka**, kao i **uvođenja ograničenja** za postojeće tipove podataka, kao na primjer:

type L **is** ('U','0','1','Z'); -L je objekat deklarisan kao niz uređenih vrijednosti: 'U','0','1' i 'Z'

type INDEX **is** range 0 **to** 15; -- INDEX je tipa integer koji uključuje vrijednosti od 0 do 15

type ADDRESS_WORD **is array** (0 **to** 63) **of** BIT; -- ADDRESS_WORD je jednodimenzoni niz objekata koji sadrži 64 elementa koji su tipa BIT.

Operatori

U VHDL-u postoji nekoliko predefinisanih operatora:

1. Operatori dodjele
2. Logički operatori
3. Aritmetički operatori
4. Relacioni operatori
5. Operatori pomjeranja bita
6. Operatori spajanja

1. Operatori dodjele

Operatori dodjele se koriste za dodjeljivanje vrijednosti signalima, varijablama i konstantama.

TABELA 2. OPERATORI DODJELE VRIJEDNOSTI

<=	dodjeljivanje vrijednosti signalima (SIGNAL).
:=	dodjeljivanje vrijednosti promjenjivim (VARIABLE), konstantama (CONSTANT) i GENERIC.
=>	dodjeljivanje vrijednosti pojedinačnim elementima vektora ili OTHERS.

Primjer: Na osnovu datih deklaracija moguće je koristiti sljedeće izraze:

```
SIGNAL x: STD_LOGIC;
VARIABLE y: STD_LOGIC_VECTOR(3 downto 0);
SIGNAL w: STD_LOGIC_VECTOR(7 downto 0);

x <= '1';
y := "0000";
w <= "10000000" -- najznačajniji bit je 1, ostali 0
w <= (0 => '1', OTHERS => '0'); -- najznačajniji bit je 1, ostali 0
```

Primijetimo da se vrijednosti koje se dodjeljuju nizovima navode pod dvostrukim znacima navoda.

2. Logički operatori

Logički operatori se mogu koristiti nad sljedećim vrstama podataka: BIT i STD_LOGIC, kao i njihovim produžecima: BIT_VECTOR, STD_LOGIC_VECTOR.

TABELA 3: LOGIČKI OPERATORI

NOT	NE
AND	I
OR	ILI
NAND	NI
NOR	NILI
XOR	ekskluzivno ILI
XNOR	ekskluzivno NILI

Primjeri upotrebe logičkih operatora:

```
y <= NOT a AND b;
y <= NOT (a AND b);
y <= a NAND b;
```

3. Aritmetički operatori

Aritmetički operatori se mogu koristiti nad sljedećim vrstama podataka: INTEGER, SIGNED, UNSIGNED ili REAL.

TABELA 4: ARITMETIČKI OPERATORI

+	Sabiranje
-	oduzimanje
*	Množenje
/	Dijeljenje
**	eksponent
MOD	modul broja
REM	ostatak dijeljenja
ABS	apsolutna vrijednost

Primjeri upotrebe aritmetičkih operatora:

```
sum <= a + b;
x <= A rem B;
```

4. Relacioni operatori

Relacioni operatori se mogu koristiti nad sljedećim vrstama podataka: INTEGER, SIGNED, UNSIGNED ili REAL.

TABELA 5: RELACIONI OPERATORI

=	Jednako
/=	Nije jednako
<	Manje od
>	Veće od
<=	Manje ili jednako
>=	Veće ili jednako

Relacioni operatori najčešće se koriste kod uslovnih naredbi o kojima će biti rijeći kasnije.

5. Operatori pomjeranja bita

Prilikom upotrebe ovih operatora koristi se sljedeća sintaksa:

<lijevi operand> <operacija pomjeranja bita> <desni operand>

Lijevi operand je uvijek BIT VECTOR, dok je desni operand uvijek tipa INTEGER.

TABELA 6: OPERATORI POMJERANJA BITA

sll	Logičko pomjeranje u lijevo
srl	Logičko pomjeranje u desno
sla	Aritmetičko pomjeranje u lijevo
sra	Aritmetičko pomjeranje u desno
rol	Logičko rotiranje u lijevo
ror	Logičko rotiranje u desno

Primjer:

Neka je *x* <= ”01001”, onda y postaje:

```
y <= x sll 2; -- y <= "00100"
y <= x srl 3; -- y <= "00001"
y <= x rol 3; -- y <= "00101"
```

PRIMJERI

1. Napisati VHDL kod koji će obavljati funkciju I kola. Ulazni signali su A i B, tipa STD_LOGIC. Izlazni signal je X, tipa STD_LOGIC.

   Rješenje:

   Na početku treba da definišemo entitet. Svaki VHDL kod mora imati bar jedan entitet - ENTITY, u kojem se definišu ulazne i izlazne promjenljive u kolu, ključnom riječju PORT. Kako realizujemo I kolo, neka ime entiteta bude **i_kolo**. Nakon definisanja entiteta, definiše se arhitektura – ARCHITECTURE u kojoj se definiše funkcija koju kolo obavlja.

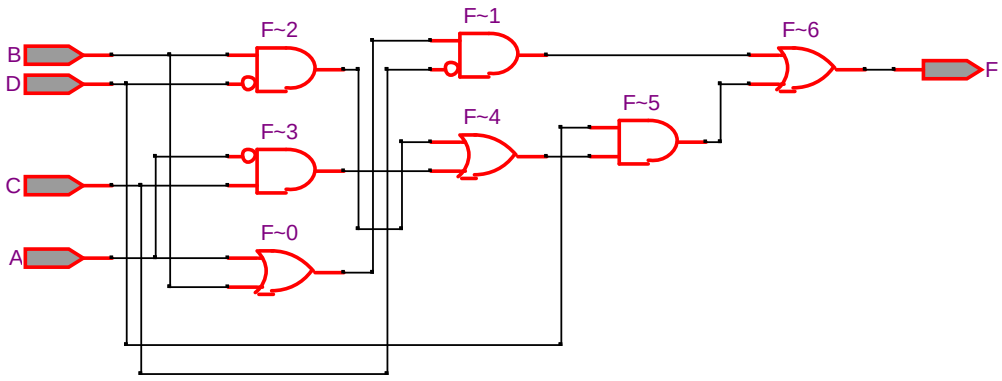
```
library ieee;
use ieee.std_logic_1164.all;

entity primjer1 is
       port(A,B : in std_logic;
              X : out std_logic);

-- primijetimo da iza posljednjeg "bit" nije potrebno pisati
-- ";",vec samo zatvoriti zagradu, a nakon zatvorene zagrade
-- staviti ";" znak

end entity primjer1;

architecture i_kolo_arhitektura of primjer1 is
begin
       X<=A and B;
end architecture i_kolo_arhitektura;
```

2. Projektovati kolo koje će realizovati fukciju: $F = (A+B)\overline{C} + (B\overline{D} + C\overline{A})D$. Svi signali su tipa BIT.

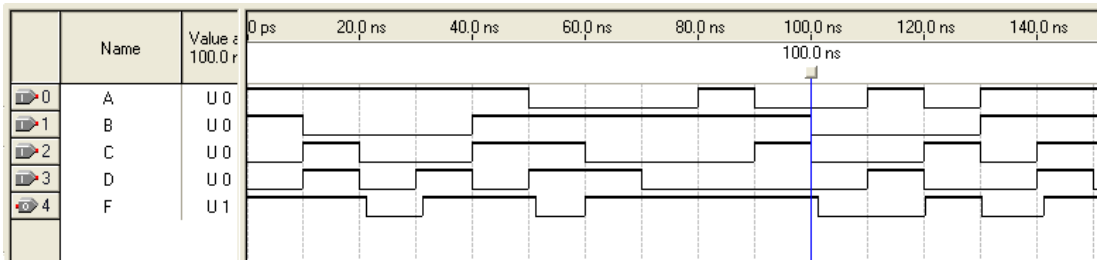
```
entity  funkcija is
port ( A, B, C, D : in  BIT;
          F       : out BIT);
end funkcija;

architecture arh_funkcija of funkcija is
begin
F <= ((A or B) and (not C)) or (((B and (not D)) or (C and
(not A))) and D);
end arh_funkcija;
```

Komentar:

- Definisan je entitet funkcija koji ima 4 ulazna signala (A, B, C i D) i jedan izlazni signal (F) svi tipa BIT
- S obzirom da su signali tipa BIT i da su biblioteke *std* i *work* automatski uključene u svakom VHDL kodu nije protrebno uključivanje dodatnih biblioteka.
- Tijelo arhitektre arh_funkcija sadrži jednu složenu naredbu koja definiše funkciju kola koje se projektuje.

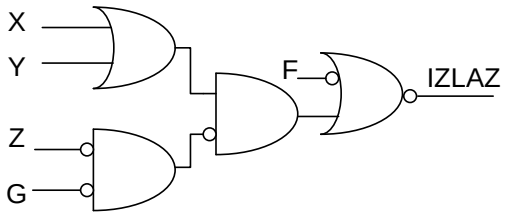


Detaljan prikaz realizacije datog kola



Rezultati simulacije

3. Napisati VHDL kod koji će realizovati funkciju datu na slici. Ulazni signali su tipa STD_LOGIC.



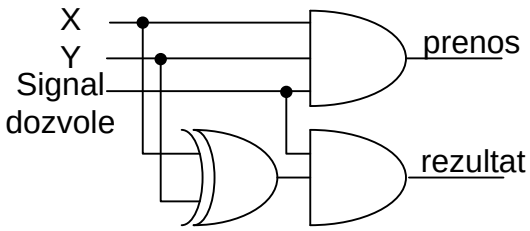
Slika uz primjer 3

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

entity primjer3 is
```

```
      port ( X, Y, Z, G, F : in std_logic;
                        IZLAZ : out std_logic);
end primjer3;

architecture arh_funkcija of primjer3 is
begin
      IZLAZ <= not (((X or Y) and (not (not Z and not G))) or
                  (not F));
end arh_funkcija;
```

Napomena: Uključena je biblioteka *ieee*, jer se radi sa *std_logic* tipom podataka.

4. Projektovati kolo polu-sabirača. Šematski prikaz polu-sabirača dat je na slici, u vidu logičkih kola.



Slika uz primjer 4

```
entity polu_sabirac is
       port( X, Y, signal_dozvole : in bit;
                  prenos, rezultat   : out bit);
end polu_sabirac;

architecture polu_sabirac_b of  polu_sabirac is
begin
       prenos <= signal_dozvole and (X and Y);
       rezultat <= signal_dozvole and (X xor Y);
end polu_sabirac_b;
```

5. Projektovati kolo potpunog sabirača. Ulazni signali su x, y i Cin, dok su izlazni signali s i Cout. Svi signali su jednobitni tipa std_logic. Funkcije kojima se realizuju izlazni signali su:

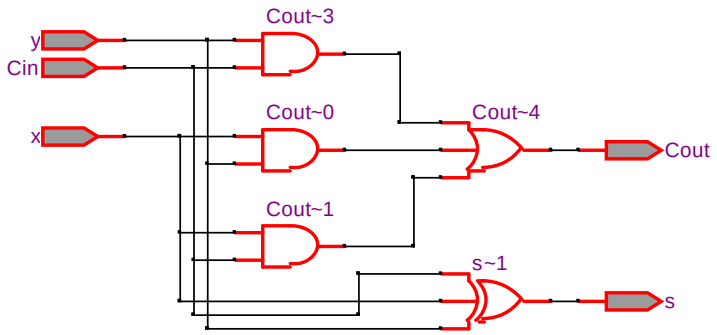
$$s = x \oplus y \oplus Cin$$

$$Cout = x \cdot y + x \cdot Cin + y \cdot Cin$$

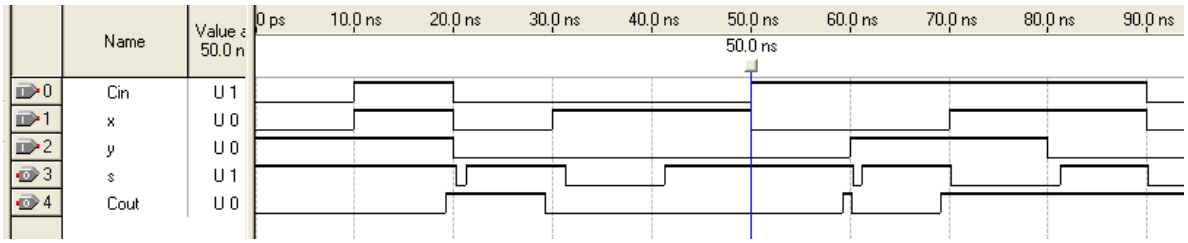
Rješenje:

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

entity potpuni_sabirac is
   port(Cin, x, y   : in  std_logic;
        s, Cout     : out std_logic);
end potpuni_sabirac;

architecture potp_sab of potpuni_sabirac is
begin
    s <= x xor y xor cin;
    Cout <= (x and y) or (x and Cin) or (y and Cin);
end potp_sab;
```

Komentar koda:
Najprije je uključena biblioteka *ieee* i paket *std_logic_1164* koji služi za rad sa STD_LOGIC tipom podataka. Nakon toga je definisan entitet potpuni_sabirac koji ima tri ulazna porta (Cin, x i y) i dva izlazna porta (s i Cout) i svi su tipa STD_LOGIC. Ime arhitekture za dati entitet je potp_sab. Oblast deklarisanja nije korišćena, a tijelo arhitekture sastoji se iz dva iskaza kojima se određuju vrijednosti izlaznih signala.



Detaljan prikaz realizacije datog kola



Rezultati simulacije

Naredbe uslovnog izvršavanja

Kao i svaki drugi programski jezik i VHDL sardži uslovne naredbe, kojima se omogućava izvršavanje dijela koda samo kada je zadovoljen neki unaprijed zadati uslov, ili uslov određen samim signalom. U VHDL-u postoji nekoliko tipova uslovnih naredbi:

- WHEN-ELSE
- WITH-SELECT-WHEN
- IF-THEN-ELSE
- CASE

Uslovna naredba za dodjelu vrijednosti signalu

Ova naredba vrši odabiranje između različitih vrijednosti koje se mogu dodijeliti ciljnom signalu, na oslovu specificiranih uslova. Postoje dvije vrste ove naredbe **WHEN-ELSE** i **WITH-SELECT-WHEN**. Opšta sintaksa ovih naredbi je:

```
signal <= [ vrijednost when uslov else ]
                [vrijednost when uslov else ]
                              . . .
                vrijednost;

      with izraz select
                 signal <= vrijednost when uslov,
                                  vrijednost  when uslov,
                                                …
                                  vrijednost when uslov ;
```

Osnovna razlika među navedenim naredbama je što prilikom upotrebe WHEN-ELSE naredbe mogu da postoje nedefinisana stanja, dok prilikom upotrebe WITH-SELECT-WHEN naredbe sva stanja moraju biti definisana ili je neophodno koristiti ključnu riječ OTHERS koja obuhvata nedefinisane slučajeve.

WHEN-ELSE naredba

Napisati kod za 2 u 4 dekoder korišćenjem WHEN-ELSE naredbe. Dekoder ima jedan dvobitni ulazni signal i jedan četvorobitni izlazni signal. Zavisnost izlaznog od ulaznog signala data je sljedećom tabelom:

Ulaz	Izlaz
00	0001
01	0010
10	0100
11	1000

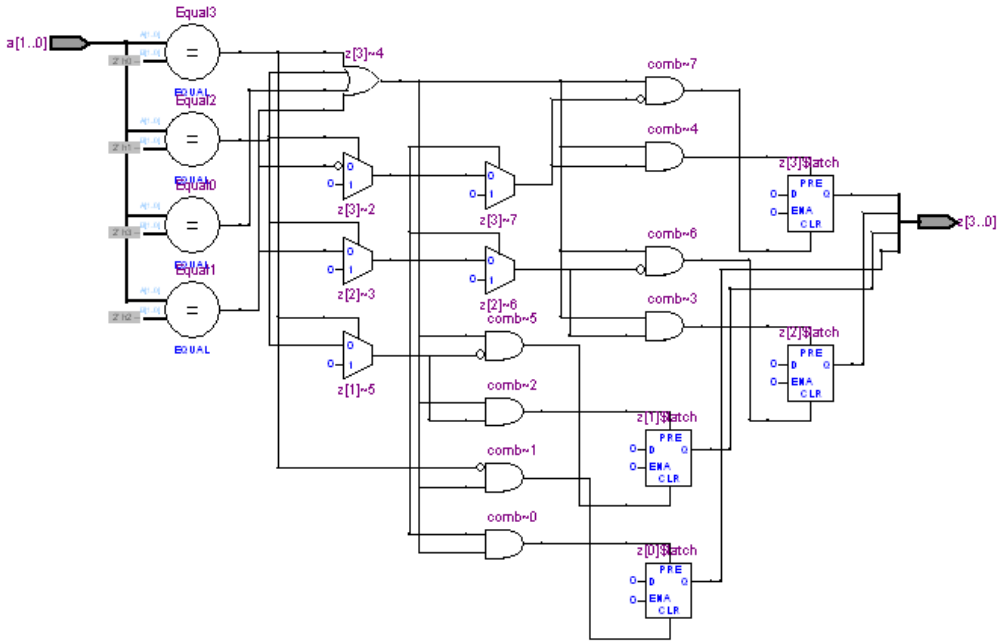
Rješenje:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity decoder2_4 is
      port( a : in     std_logic_vector(1 downto 0);
                z : out   std_logic_vector(3 downto 0));
end entity decoder2_4;

architecture decoder_when_else of decoder2_4 is
begin
z <=”0001” when a = “00” else
”0010” when a = “01” else
”0100” when a = “10” else
”1000” when a = “11”;
end architecture decoder_when_else;
```

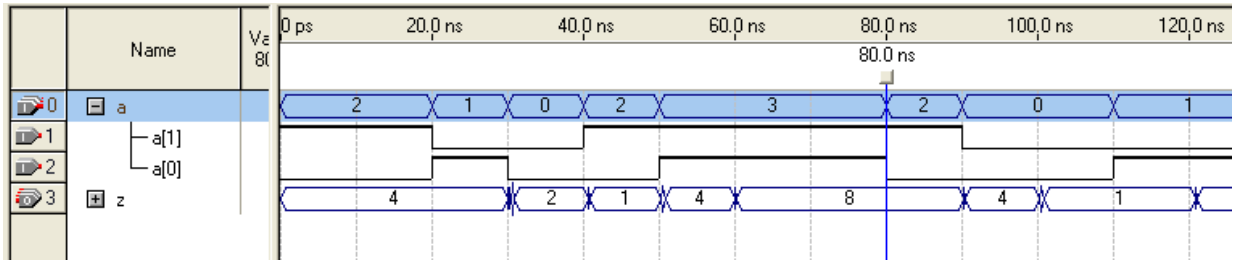
Komentar:

Definisan je entitet dekoder2_4 koji ima jedan ulazni i jedan izlazni signal. Ulazni signal je definisan kao objekat tipa STD_LOGIC_VECTOR, a izraz u zagradi (1 downto 0) označava da se radi o dvobitnom signalu, kod kojeg je prvi bit sa desne strane bit najveće važnosti (MSB). Slično je i izlazni signal definisan kao četvorobitni signal tipa STD_LOGIC_VECTOR (3 downto 0). Imajući u vidu tip ulaznog i izlaznog signala uključena je biblioteka *ieee* i paket *std_logic_1164*.

Tijelo arhitekture opisuje način rada dekodera. Izlazni signal se selektuje u zavisnosti od vrijednosti ulaznog signala. Kada se prvi put bude ispunjen jedan uslov izvršavanje koda se prekida dok se ne promijeni vrijednost ulaznog signala.



Detaljan prikaz realizacije datog kola



Rezultati simulacije

WITH-SELECT-WHEN naredba

Napisati kod za dekoder koji ima četvorobitni ulaz i sedmobitni izlaz. Kod je potrebno napisati korišćenjem naredbe WITH-SELECT-WHEN. Zavisnost izlaznih od ulaznih signala data je sljedećom tabelom. Za nedefinisane slučajeve koristiti ključnu riječ OTHERS. Uzeti da su ova signala tipa *std_logic_vector* tipa.

Input	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	OTHERS
Output	0000001	0000010	00000110	0000111	0001000	0001001	0001111	0010000	0100000	1000000	0000000

Rješenje:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity dekoder is
      port( a : in std_logic_vector(3 downto 0);
            z : out std_logic_vector(6 downto 0));
end entity dekoder;

architecture with_select of dekoder is
begin
with a select
z<=”0000001” when “0000” ,
”0000010” when “0001” ,
”0000110” when “0010” ,
”0000111” when “0011” ,
”0001000” when “0100” ,
”0001001” when “0101” ,
”0001111” when “0110” ,
”0010000” when “0111” ,
”0100000” when “1000” ,
”1000000” when “1001” ,
”0000000” when others;
end architecture with_select;
```

IF-ELSE naredba

Kao i u većini programskih jezika IF-ELSE naredba služi za sekvencijalno izršavanje naredbi kada je zadovoljen zadati uslov. Opšta sintaksa IF-ELSE naredbe je:

IF *uslov* THEN
skup naredbi;
ELSE
skup naredbi;
END IF;

Veoma često se uz IF-ELSE naredbu koristi naredba PROCESS koja služi za ponavljanje niza naredbi u okviru procesa (tijela procesa). Proces sadrži jedan ili više argumenata koji mogu biti neki od signala u kolu. Sa svakom promjenom argumenta naredbe PROCESS izvršava se niz naredbi zadat u okviru tijela procesa. Opšta struktura PROCESS naredbe je:

PROCESS (*argument*)

BEGIN
skup naredbi i iskaza;
END PROCESS;

```
If  A = '0' then --- Ako je A=0
        C<=B;  --- onda se u C
               --- upisuje B
End if;
```

```
If  A = '0' then      --- Ako je A=0
      C<=B;           --- onda se u C
Elsif  A = '1' then   --- upisuje B, a ako je A=1,
      C<=A;           --- onda se u C upisuje A
End if;
```

Primjer:

Napisati kod za D flip flop korišćenjem naredbe IF-ELSE. Ulazni signali su *d*, *clk* (clock signal) i *rst* (reset signal), dok je izlazni signal *q*. Svi signali su tipa *std_logic*. Na promjenu vrijednosti izlaznog signala utiču signali *rst* i *clk*, sa čijom promjenom može doći do promjene izlaznog signala. Funkcionisanje D flip flopa može se opisati kao:

$$q = \begin{cases} 0 & rst = 1 \\ d & rst = 0 \text{ i } clk = 1 \text{ (pri promjeni } clk \text{ signala)} \end{cases}$$

Napomena: Ispitivanje promjene u *clk* signalu vrši se upotrebom atributa *clk'event* kojim se provjerava da li se desio neki događaj, odnosno promjena u signalu.
Rješenje:

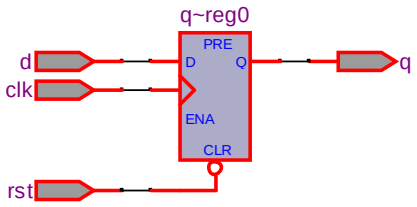
```
library ieee;
use ieee.std_logic_1164.all;

entity dff_1 is
port (d, clk, rst: in std_logic;
        q: out std_logic);
end dff_1;

architecture dff_arh of dff_1 is
begin
    process (rst, clk)
    begin
if (rst = '1') then
      q <= '0';
else if (clk'event and clk='1') then
      q <=d;
                    end if;
end if;
    end process;
end dff_arh;
```

Komentar:

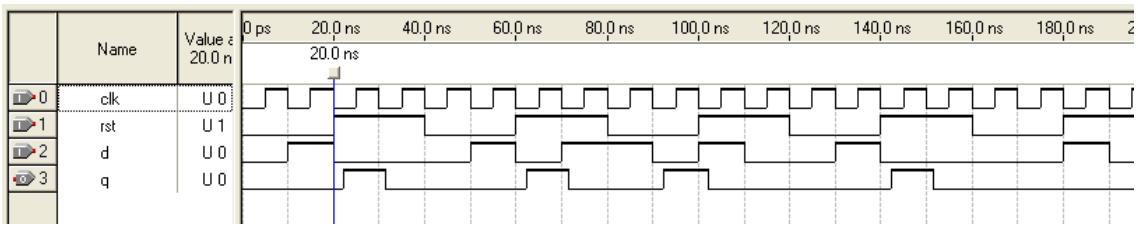
Definisan je enitet dff_1 koji obavlja funkciju D flip flopa. Ovaj entitet ima tri ulazna i jedan izlazni signal. Svi signali su jednobitni i tipa STD_LOGIC, te je stoga i uključena biblioteka *ieee.std_logic_1164*.

Tijelo arhitekture definisano je pomoću IF-ELSE naredbe. Vidimo da se uvijek nakon IF naredbe javlja naredba THEN koja ukazuje koji izkazi treba da se izvrše.

Primijetimo da tijelo arhitekture sadrži i PROCESS naredbu. Ova naredba služi za sekvencijalno izvršavanje programa, odnosno naredbe u okviru procesa se ponavljaju svakom promjenom iskaza nevedenog u zagradama (u navedenom primjeru promjenom signala *clk* i *rst*). Dakle, kada se promijeni stanje clock ili reset signala ispituje se uslov zadat IF naredbom i ukoliko je on zadovoljen izvršavaju se navedene naredbe.



Simbol D flip flopa



Rezultati simulacije

FOR naredba

For petlja omogućava da se izvršavanje određenog dijela koda ponovi tačno zadati broj puta.
Opšta sintaksa FOR naredbe je:

```
FOR promjenljiva IN opseg LOOP
        {Skup naredbi};
END FOR;
```

Primjer:
Napisati kod za sedmobitni Carry Look Ahead sabirač. Princip funkcionisanja sabirača može se opisati relacijama:
Izlazna vrijednost $S = A \oplus B \oplus C$;

Prenos $C_{i+1} = G_i + P_i C_i$, gdje je G = AB , a P = A+B , pri čemu su A i B sedmobitni ulazni signali, dok je C jednobitni ulazni signal koji označava početni prenos. Izlazni signal S je sedmobitni, dok je izlazni signal C_out jednobitni i označava izlazni prenos.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
ENTITY sabirac IS
PORT(
      A : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
      B : IN STD_LOGIC_VECTOR(6 DOWNTO 0);
      C_in : IN STD_LOGIC;
      S : OUT STD_LOGIC_VECTOR(6 DOWNTO 0);
      C_out : OUT STD_LOGIC);
END sabirac;

ARCHITECTURE arh_sabirac OF sabirac IS
      SIGNAL sum : STD_LOGIC_VECTOR(6 DOWNTO 0);
      SIGNAL G : STD_LOGIC_VECTOR(6 DOWNTO 0);
      SIGNAL P : STD_LOGIC_VECTOR(6 DOWNTO 0);
      SIGNAL C_in_p : STD_LOGIC_VECTOR(6 DOWNTO 1);
BEGIN
      sum <= A XOR B;
      G <= A AND B;
      P <= A OR B;
PROCESS (G, P, C_in_p)
BEGIN
C_in_p(1) <= G(0) OR (P(0) AND C_in); -- Nije u okviru FOR petlje jer
                                      --C_in_p(0) ne
                                      -- postoji
      FOR i IN 1 TO 5 LOOP
            C_in_p(i+1) <= G(i) OR (P(i) AND C_in_p(i));
      END LOOP;
            C_out <= G(6) OR (P(6) AND C_in_p(6));
      END PROCESS;
            S(0) <= sum(0) XOR C_in;
            S(6 DOWNTO 1) <= sum(6 DOWNTO 1) XOR C_in_p(6 DOWNTO 1);
END arh_sabirac;
```

CASE naredba

CASE naredba se koristi za uslovno izvršavanje naredbi kada je zadovoljen unaprijed zadati uslov. Opšta stuktura CASE naredbe je:

case *ime_signala* **is**
    when vrijednost => iskaz (ili skup iskaza);
    when vrijednost => iskaz (ili skup iskaza);

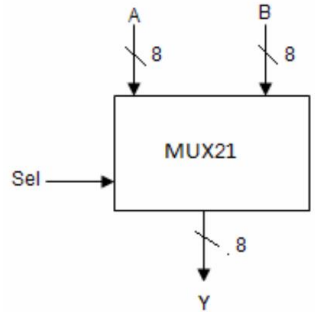
    when vrijednost => iskaz (ili skup iskaza);
end case

PRIMJERI

1. Koristeći naredbu CASE, realizovati kolo koje ima ulaze A, B i SEL, i izlaz Z. SEL signal je dvobitni a A i B su tipa STD_LOGIC. Ukoliko je SEL 01 na izlaz se proslijeđuje signal A, a ako je SEL 10, na izlaz se proslijeđuje signal B. Ukoliko je na SEL ulazu neka druga kombinacija, na izlaz se proslijeđuje 0.

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

 entity primjer is
 port (A, B: in std_logic;
        SEL : out std_logic_vector(0 to 1);
          Z: out std_logic);
 end primjer;

 architecture arh1 of primjer is
 begin

        case SEL is
                 when "01" => Z <= A;
                 when "10" => Z <= B;
                 when others => Z <= '0';
        end case;
end arh1;
```

2. Napisati VHDL kod za multiplekser 2 u 1. Ulazni signali su A i B, tipa STD_LOGIC_VECTOR i dužine 8 bita i Sel signal tipa STD_LOGIC. Izlazni signal je Y, tipa STD_LOGIC_VECTOR i dužine 8 bita. Ukoliko je Sel signal jednak '1' na izlaz se proslijeđuje B, dok za Sel vrijednost '0', na izlazu imamo A.



Slika uz zadatak 2

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

entity MUX2to1 is

port(A, B: in std_logic_vector(7 downto 0);
    Sel: in std_logic;
    Y: out std_logic_vector(7 downto 0));
end MUX2to1;

architecture behavior of MUX2to1 is
begin
process (Sel, A, B)
begin
 if (Sel = '1') then
      Y <= B;
 else
      Y <= A;
 end if;
 end process;
end behavior;
```

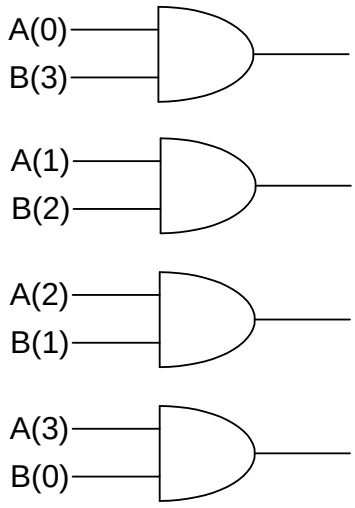
3. Prethodni primjer uraditi korišćenjem naredbe CASE.

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

entity MUX2to1 is
      port (A, B: in std_logic_vector(7 downto 0);
              Sel: in std_logic;
              Y: out std_logic_vector(7 downto 0));
end MUX2to1;

architecture behavior of MUX2to1 is
begin
 process (Sel, A, B)
begin
       case Sel is
              when '0' => Y <= A;
              when '1' =>  Y <=B;
       end case;
 end process;
end behavior;
```

4. Napisati VHDL kod koji će obavljati funkciju 4-robitnog komparatora. Ulazni signali su 4-robitni, tipa std_logic_vector, dok su izlazni signali tipa std_logic. Komparator treba da obavlja operacije =, /=,> i <.

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

entity komparator is
port (
        A, B: in STD_LOGIC_VECTOR (3 downto 0);
        S1, S2, S3, S4: out STD_LOGIC );
end komparator;

architecture komparator_arch of komparator is
begin
        process (A, B)
        begin
        S1<='0';
        S2<='0';
        S3<='0';
        S4<='0';
                if A=B then S1<='1'; end if;
                if A/=B then S2<='1'; end if;
                if A>B then S3<='1'; end if;
                if A>=B then S4<='1'; end if;
        end process;
end komparator_arch;
```

5. Napisati VHDL kod kojim se realizuje funkcija kola datog na slici. Signali su četvorobitni, tipa BIT_VECTOR.



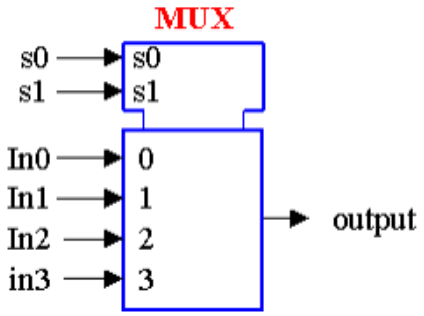
Slika uz zadatak 5

VHDL KOD:

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

entity primjer5 is
      port (A, B: in bit_vector(3 downto 0);
               F: out bit_vector(3 downto 0));
end primjer5;

architecture arhitektura of primjer5 is

begin
      for i in 0 to 3 loop
            F(i) <= A(i) and B(3-i);
      end loop;
end arhitektura;
```

6. Napisati VHDL kod kojim se realizuje multiplekser 4 u 1. Kod napisati korišćenjem:

1) IF;
2) CASE;
3) WITH-SELECT-WHEN i
4) WHEN-ELSE naredbi.



Slika uz zadatak 6

1) Naredba IF:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;
```

```
ENTITY  mux4_1 IS
   PORT (s0               : IN  STD_LOGIC;
         s1               : IN  STD_LOGIC;
         in0              : IN  STD_LOGIC;
         in1              : IN  STD_LOGIC;
         in2              : IN  STD_LOGIC;
         in3              : IN  STD_LOGIC;
         output           : OUT STD_LOGIC
        );
END mux4_1;

ARCHITECTURE if_naredba OF mux4_1 IS
BEGIN

PROCESS(s0, s1, in0, in1, in2, in3)
BEGIN

  IF    (s0='0' AND s1='0') THEN
    output <= in0;
  ELSIF (s0='1' AND s1='0') THEN
    output <= in1;
  ELSIF (s0='0' AND s1='1') THEN
    output <= in2;
  ELSIF (s0='1' AND s1='1') THEN
    output <= in3;
  ELSE        -- (s0 ili s1 nijesu 0 ili 1, već imaju neke druge vrijednosti)
    output <= 'X';
  END IF;

END PROCESS;

END if_naredba;
```

2) Naredba CASE:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY  mux4_1 IS
   PORT (s0               : IN  STD_LOGIC;
         s1               : IN  STD_LOGIC;
         in0              : IN  STD_LOGIC;
         in1              : IN  STD_LOGIC;
         in2              : IN  STD_LOGIC;
         in3              : IN  STD_LOGIC;
         output           : OUT STD_LOGIC
```

```
        );
END mux4_1;


ARCHITECTURE case_PRIMJER OF mux4_1 IS

BEGIN

PROCESS(s0, s1, in0, in1, in2, in3)
VARIABLE  sel  :  STD_LOGIC_VECTOR(1 DOWNTO 0);

-- Uvedena je pomoćna promjenljiva sel, koja nadovezuje signale s1 i s0, u
-- cilju istovremenog ispitvanja slučajeva koji se mogu pojaviti na ulazu

BEGIN

  sel := s1 & s0; -- nadovezujemo signale s1 i s0 u jedan signal - sel

  CASE sel IS
    WHEN  "00"  =>  output <= in0;
    WHEN  "01"  =>  output <= in1;
    WHEN  "10"  =>  output <= in2;
    WHEN  "11"  =>  output <= in3;
    WHEN OTHERS =>  output <= 'X'; -- ukoliko sel uzme neku vrijednost koja -
                                   -- nije "00", "01", "10", ili "11", na
                                   -- izlazu ćemo imati 'X', odnosno
                                    -- nedefinisano stanje
  END CASE;

END PROCESS;

END case_primjer;
```

3) Naredba WITH-SELECT-WHEN:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY  mux4_1 IS
   PORT (s0              : IN  STD_LOGIC;
         s1              : IN  STD_LOGIC;
         in0             : IN  STD_LOGIC;
         in1             : IN  STD_LOGIC;
         in2             : IN  STD_LOGIC;
         in3             : IN  STD_LOGIC;
         output          : OUT STD_LOGIC
```

```
        );
END mux4_1;


ARCHITECTURE with_primjer OF mux4_1 IS

SIGNAL  sel  :  STD_LOGIC_VECTOR(1 DOWNTO 0);

BEGIN
  sel <= s1 & s0;
  WITH sel  SELECT
    output <= in0 WHEN "00",
              in1 WHEN "01",
              in2 WHEN "10",
              in3 WHEN "11",
              'X' WHEN OTHERS;

END with_primjer;
```

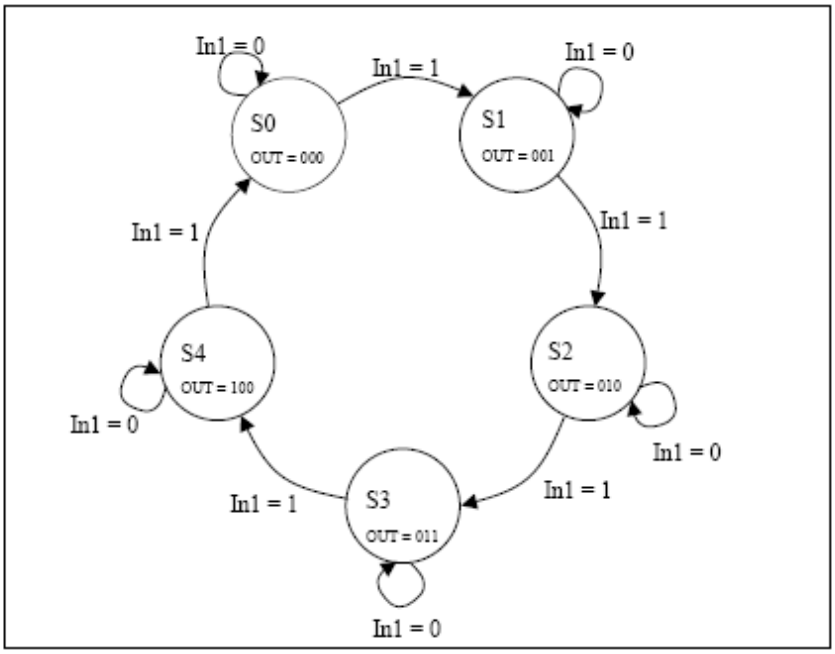
4) Naredba WHEN-ELSE:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_unsigned.ALL;

ENTITY  mux4_1 IS
   PORT (s0               : IN  STD_LOGIC;
         s1               : IN  STD_LOGIC;
         in0              : IN  STD_LOGIC;
         in1              : IN  STD_LOGIC;
         in2              : IN  STD_LOGIC;
         in3              : IN  STD_LOGIC;
         output           : OUT STD_LOGIC
        );
END mux4_1;


ARCHITECTURE when_primjer OF mux4_1 IS

BEGIN

    output <= in0 WHEN (s1 & s0)="00" ELSE
              in1 WHEN (s1 & s0)="01" ELSE
              in2 WHEN (s1 & s0)="10" ELSE
              in3 WHEN (s1 & s0)="11" ELSE
```

```
            'X';

END when_primjer;
```

NAPOMENA: VHDL jezik ne pravi razliku između velikih i malih slova. U prethodnim primjerima neke naredbe su pisane velikim slovima samo u cilju bolje preglednosti.

7. Napisati kod za automat konačnih stanja prikazan na slici. Kolo je upravljano clk i reset i In1 signalima. Za rst=1 automat se vraća u početno stanje s0. Prelazak na naredno stanje izvršava se na pozitivnoj ivici klok signala (rising_edge(clk)) i zavisi od In1 signala (kao na slici). Izlazni signal je out tipa std_logic_vector.

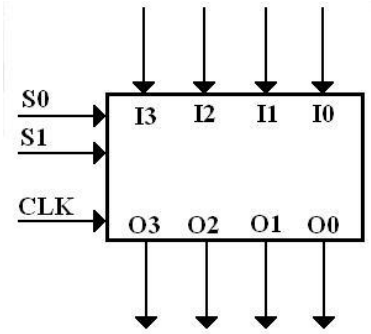



```
library ieee;
use ieee.std_logic_1164.all;

entity FSM1 is
port(clk, in1, rst: in std_logic;
      out1: out std_logic_vector(2 downto 0));
end FSM1;

architecture RTL of FSM1 is

type state_values is (s0, s1, s2, s3, s4);
signal state, next_state: state_values;
```

```
begin

process (clk, rst)
begin

if rst = '1' then
      state <= s0;
elsif rising_edge(clk) then
      state <= next_state;
end if;
end process;

process (state, in1)
begin

case state is
when s0 =>
      out1 <="000";
      if in1 = '0' then
            next_state <= s0;
      else
            next_state <= s1;
      end if;
when s1 =>
      out1 <="001";
      if in1 = '0' then
            next_state <= s1;
      else
            next_state <= s2;
      end if;
when s2 =>
      out1 <="010";
      if in1 = '0' then
            next_state <= s2;
      else
            next_state <= s3;
      end if;
when s3 =>
      out1 <="011";
      if in1 = '0' then
            next_state <= s3;
      else
            next_state <= s4;
      end if;
when s4 =>
      out1 <="100";
      if in1 = '0' then
            next_state <= s4;
      else
            next_state <= s0;
      end if;
when others =>
      out1 <= "000";
```

```
            next_state <= s0;
end case;
end process;
end RTL;
```

8. Na slici je prikazana blok šema sinhronog pomjeračkog bidirekcionog registra. Pomjeranje se vrši cirkularno ili u lijevo ili u desno. Pomjeranje ulijevo znači da se izbacuje izlazni bit najveće važnosti, a ostali izlazni biti se pomjeraju u lijevo, dok se na mjestu izlaznog bita najmanje težine upisuje najznačajniji bit ulaznog signala. Sličan princip važi i za pomijeranje u desno. Naime, odbacuje se LSB izlazni bit, dok se ostali izlazni biti pomjeraju u desno, a na mjestu MSB izlaznog bita upisuje se LSB bit ulaznog signala.
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 (*rising_edge(clk)*). Za odgovarajući režim rada registra koriste se signali S0 i S1.
U tabeli na slici 2 dati su režimi rada registra u zavisnosti od signala S0 i S1. Implementirati ovaj registar u VHDL-u.



S_1	S_0	Režim rada
0	0	Zadržava se postojeće stanje
0	1	Pomjeranje u lijevo
1	0	Pomjeranje u desno
1	1	Paralelni upis

```
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); -- bitovi na ulazu
            s : in std_logic_vector (1 downto 0); -- kontrolni signali
            o : out std_logic_vector (3 downto 0) -- bitovi na izlazu
            );
end shifter;
architecture structure of shifter is
    signal internal : std_logic_vector(3 downto 0); --interna promjenjiva
begin
    process (clk)
```

```
    begin
      if rising_edge(clk) then
            if s="00" then   -- za S1S0=00 ostaje u sadasnjem stanju
                  internal <= internal;
            elsif s=    "01" then  -- ako je S1S0=01 pomjera u lijevo
             internal (0) <= i (3);
             internal (1) <= internal (0);
             internal (2) <= internal(1);
             internal (3) <= internal (2);
            elsif s=    "10" then  -- ako je S1S0=10 pomjera u desno
             internal (0) <= internal (1);
             internal (1) <= internal (2);
             internal (2) <= internal (3);
             internal (3) <= i (0);
             elsif s=   "11" then  -- ako je S1S0=11 paralelni upis
             internal <= i;
            end if;
      end if;
    end process;
    o <= internal; -- interna promjenjiva se sada salje na izlaz
end structure;
```

9. Napisati kod za automat koji će obavljati funkciju paljenja pokazivača pravca na automobilu. Ukoliko je aktivan ulazni signal LEFT uključuju se tri lijeva pokazivača, tako što se najprije uključuje jedan, zatim dva i na kraju sva tri pozivača. Ukoliko je aktivan ulazni signal RIGHT uključuju se tri desna pokazivača, tako što se najprije uključuje jedan, zatim dva i na kraju sva tri. Ukoliko je aktivan ulazni signal SVI uključuje se svih šest pokazivača istovremeno. Kada se uključe svi pokazivači, ili sva tri lijeva ili sva tri desna automat se vraća u početno stanje kada nijedan pokazivač nije uključen. U svakom trenutku treba provjeriti da li je aktivan signal SVI. Vrijednosti izlaznog signala za svako od stanja dati su u narednoj tabeli.

STANJA	Izlazni signal
P	000
S	001
L1	010
L2	011
L3	100
R1	101
R2	110
R3	111

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

entity Automat is
port(clk, LEFT, RIGHT, SVI: in std_logic;
      out1: out std_logic_vector(2 downto 0));
```

```
end Automat;

architecture ARCH of Automat is

type state_values is (P, S, L1, L2, L3, R1, R2, R3);
signal state, next_state: state_values;

begin

process (clk)
begin

if clk = '1' then
      state <= next_state;
end if;
end process;

process (state, LEFT, RIGHT, SVI)
begin

case state is
when P =>
      out1 <="000";
      if SVI = '1' then
            next_state <= S;
      elsif LEFT ='1' then
            next_state <= L1;
      elsif RIGHT ='1' then
            next_state <= R1;
      else
            next_state <= P;
      end if;
when S =>
      out1 <="001";
      next_state <= P;
when L1 =>
      out1 <="010";
      if SVI = '1' then
            next_state <= S;
      else
            next_state <= L2;
      end if;
when L2 =>
      out1 <="011";
      if SVI = '1' then
            next_state <= S;
      else
            next_state <= L3;
      end if;
when L3 =>
      out1 <="100";
            next_state <= P;
when R1 =>
      out1 <="101";
```

```
      if SVI = '1' then
            next_state <= S;
      else
            next_state <= R2;
      end if;
when R2 =>
      out1 <="110";
      if SVI = '1' then
            next_state <= S;
      else
            next_state <= R3;
      end if;
when R3 =>
      out1 <="111";
      next_state <= P;
end case;
end process;
end ARCH;
```