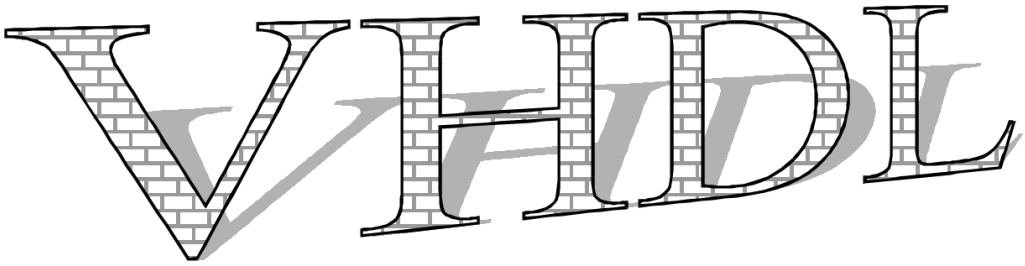
Univerza v Ljubljani
Fakulteta za elektrotehniko

Andrej Trost

Načrtovanje digitalnih vezij v jeziku



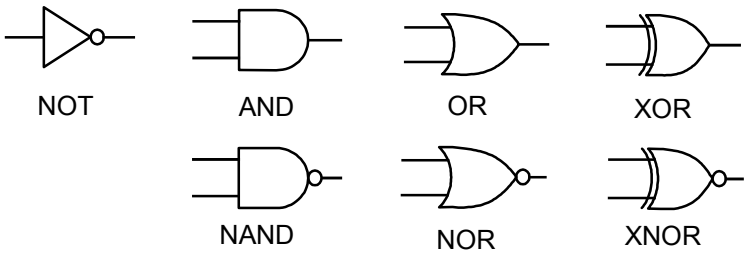
Delovno gradivo za laboratorijske vaje pri predmetu Integrirana vezja

Ljubljana, 2001

1.

Uvod v VHDL

Osnovni elementi digitalnih vezij so logična vrata in flip-flopi. Med seboj jih povezujemo v kompleksnejše gradnike kot so npr. dekodirniki, seštevalniki in registri. Z uporabo osnovnih in kompleksnejših gradnikov sestavljamo in načrtujemo nova digitalna vezja.



a	B	and	or	xor
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Slika 1.1: Nekaj osnovnih logičnih vrat in pravilnostna tabela za operacije and, or in xor

Tradicionalni pristop k načrtovanju digitalnih vezij poteka z risanjem sheme vezja. Shematsko načrtovanje uporabljamo na različnih nivojih opisa vezja: na nivoju tranzistorjev, logičnih vrat, kompleksnih elementov in celotnega sistema. Shema vezja zelo pregledno prikazuje relacije med posameznimi gradniki vezja. Shematsko načrtovanje pa postane neprimerno pri zelo kompleksnih digitalnih vezjih, ki so sestavljena iz velikega števila elementov. V tem primeru je bolj primerna uporaba visokonivojskih jezikov, kot je VHDL, s katerimi opišemo delovanje vezja.

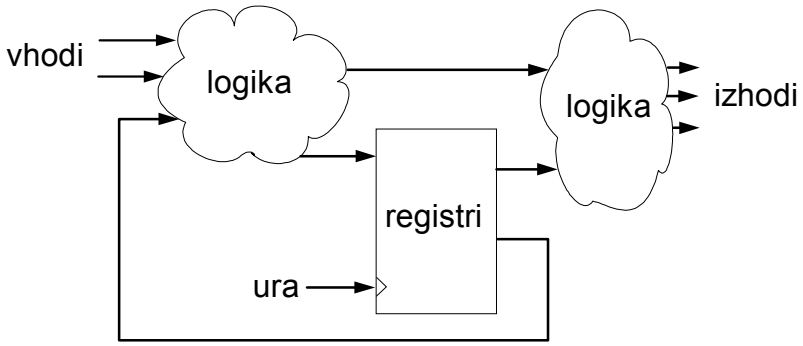
Ko začnemo načrtovati digitalno vezje pogosto poznamo le željeno delovanje vezja, ne pa tudi zgradbo, ki bo omogočala takšno delovanje. Z visokonivojskim opisom vezja v jeziku VHDL lahko na hiter in kompakten način vnesemo vezje v računalnik ter opravimo simulacijo vezja. Računalniški programi za sintezo vezij nam iz VHDL opisa vezja generirajo optimizirano vezje na nivoju logičnih vrat. Datoteka z vezjem na nivoju logičnih vrat pa je osnova za izdelavo vezja v izbrani tehnologiji (npr. monolitna tehnologija standarnih celic, programirljiva vezja...).

Jezik VHDL nam omogoča različne načine opisa digitalnih vezij: z visokonivojskimi stavki (npr. if stavek) opisujemo obnašanje vezja (angl. behavioral description) ali pa povezujemo v naprej pripravljene strukture, podobno kot pri risanju sheme vezja (angl. structural description). VHDL je bil prvotno namenjen za modeliranje in simulacijo digitalnih vezij, zato vsebuje tudi ukaze in strukture, ki jih ne moremo sintetizirati. Primer takšnih struktur so časovne zakasnitve, ki omogočajo zelo natančno simulacijo,

ne moremo pa sintetizirati vezja s poljubnimi časovnimi zakasnitvami. Primer ukazov, ki niso smiselni pri sintezi vezij, pa so ukazi za delo z datotekami, izpis na zaslon itd.

Digitalna vezja v splošnem delimo na kombinacijska in sekvenčna vezja. Kombinacijska vezja so sestavljena iz logičnih vrat in jih lahko opišemo s preslikavo med vhodi in izhodi vezja. Vsak izhodni signal je določen z vrednostmi enega ali večih vhodnih signalov. Za opis kombinacijskih vezij uporabljamo v jeziku VHDL logične operatorje ali pa visokonivojske stavke. S sintezo kombinacijskih vezij običajno nimamo težav.

Sekvenčna vezja vsebujejo poleg logičnih vrat tudi pomnilne elemente (flip-flope, povratne zanke itd.). Pri obravnavi sekvenčnih vezij lahko ločimo del, ki ga sestavljajo pomnilni elementi, od kombinacijskega dela vezja, kot prikazuje slika 1.2. Takšne strukture smo spoznali pri teoriji avtomatov. Kot pomnilne elemente uporabimo flip-flope in registre, kombinacijski del pa sestavimo iz logičnih vrat.



Slika 1.2: Sekvenčno vezje

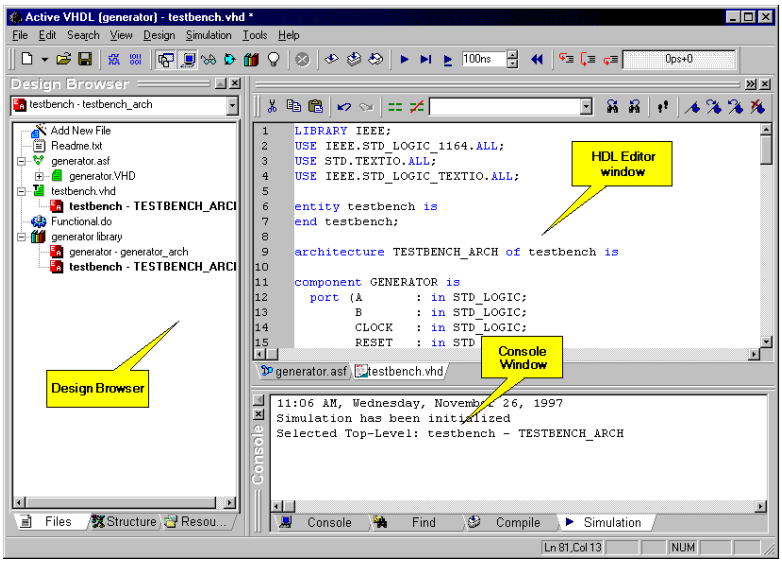
Način opisa tako razdeljenega vezja v jeziku VHDL se imenuje RTL opis (Register Transfer Level). Pri obravnavi sekvenčnih vezij bomo spoznali strukture za opis registrov, ki so osnova za RTL opis vezja in uspešno izvedeno sintezo sekvenčnih vezij. Sekvenčno vezje lahko opisujemo z visokonivojskimi stavki in simuliramo tudi brez posebne delitve na pomnilni in kombinacijski del, vendar sinteza vezja v tem primeru ni vedno izvedljiva. Kakšen opis lahko sintetiziramo je v tem primeru predvsem odvisno od zmogljivosti računalniškega programa za sintezo.

Pri opisu in simulaciji digitalnih vezij se srečujemo s signali: vhodnimi signali v vezje, izhodi vezja in notranjimi signali. V jeziku VHDL ima vsak signal definiran svoj podatkovni tip, podobno kot spremenljivke v programskih jezikih. Signali v digitalnih vezjih zavzamejo običajno eno izmed dveh možnih logičnih vrednosti: logično ničlo ali enico. Takšni signali so definirani kot podatkovni tip **bit**. Več signalov združujemo v vodila, za katere uporabljamo v jeziku VHDL vektorske podatkovne tipe: npr. **bit_vector**. Za simulacijo je včasih bolj primeren tip **std_logic** (in **std_logic_vector**), ker vsebuje tudi nedefinirano stanje, kratek stik, stanje visoke impedance itd. VHDL pozna še veliko drugih podatkovnih tipov, ki pa v tem delu jih ne bomo obravnavali.

1.1 Uvod v program Active-HDL

Program Active-HDL uporabljamo za opis is simulacijo digitalnih vezij v jeziku VHDL. Ko v Windows okolju zaženemo program imamo na izbiro odpiranje že obstoječega načrta vezja ali kreiranje novega. Postopek kreiranja novega načrta je podrobneje opisan v nadaljevanju. Osnovno okno programa Active-HDL je sestavljeno iz treh okvirjev:

- *Design Browser* za pregled nad vsemi datotekami
- Urejevalnik VHDL programa ali simulacijsko okno
- *Console Window* v katerem se izpisujejo sporočila pri prevajanju, simulaciji...



Slika 1.3: Program Active-HDL

Načrtovanje novega vezja

- Izberemo *Create New Design* v prvem oknu, ki se odpre po zagonu programa ali pa iz menuja **File ➔ New Design**.

Sedaj bomo s pomočjo čarovnika (*New Design Wizard*) naredili nov načrt in ogrodje VHDL programa. Postopek sestavljajo naslednji koraki:

- Vpišemo ime novega načrta in lokacijo na disku
- V naslednjem oknu izberemo *Create new source files now*
- Pritisnemo gumb **New** in vpišemo ime vezja (*Entity Name*). V istem oknu pritisnemo še na gumb **Ports...**, s katerim se odpre okno za vnašanje vhodov in izhodov vezja
- Ko smo vpisali vse podatke, pritisnemo **Finish** in pridemo v osnovno okno.

Čarovnik nam je generiral datoteko z ogrodjem VHDL programa, ki jo vidimo v okvirju *Design Browser* v obliki modre ikone z vprašajem (vprašaj označuje, da program še ni bil preveden).

Program odpremo v tekstovnem urejevalniku tako, da dvakrat kliknemo na ikono.

Prevajanje

Ko je program napisan, ga prevedemo s pritiskom na ikono za prevajanje v zgornji opravilni vrstici. Med prevajanjem se v spodnjem okvirju izpisujejo sporočila. Če pride do napak med prevajanjem se vrstice VHDL programa v katerih so napake podčrtajo z rdečo. Na napako v VHDL datoteki opozarja tudi rdeč x zraven ikone v okvirju *Design Browser*, ki se spremeni v zeleno kljukico, kadar opravimo prevajanje brez napak.

Simulacija

Pred začetkom simulacije vezja moramo opraviti inicializacijo simulatorja - izberemo iz menuja: **Simulation ➔ Initialize Simulation** in nato ime vezja, ki ga želimo simulirati.

Simulacijsko okno odpremo s klikom na ikono *New Waveform*. Odpre se nam prazno okno, v katerega bomo dodali opazovane signale.

Signale dodajamo s pritiskom na ikono *Add Signals* in izbiro signalov v oknu, ki se nam odpre. Izberemo lahko posamezni signal ali več signalov (pritisnemo z levim gumbom miške na prvi signal, nato držimo tipko Shift in pritisnemo na zadnji signal, nato pa s pritiskom gumba **Add** dodamo signale v simulacijsko okno).

Sedaj moramo vhodnim signalom dodati še stimulatorje. V simulacijskem oknu izberemo signal (pritisnemo z levim gumbom miške na ime signala), nato pa pritisnemo ikono *Stimulators*. Definirati moramo vrsto stimulatorja in v ustrezna polja zapisati parametre oz. vrednosti:

- za konstantne signale izberemo *Value* in ustrezno vrednost *Force Value*
- za urin signal izberemo *Clock* in vpišemo parametre
- za ostale signale je najbolje, da izberemo *Hotkey* ter v okvirčku *Press new hotkey* pritisnemo neko črko na tipkovnici - med izvajanjem simulacije bomo lahko s pritiskom na to črko menjali vrednost signala (med 0 in 1)

Simulacijo poženemo s pritiskom na eno izmed ikon *Run*. Npr. z ikono *Run For* poženemo simulacijo v dolžini časovnega intervala, ki je vpisan v okvirčku. Simulacijo lahko poganjamo postopoma in vmes spreminjamo vrednosti stimulatorjev (vpišemo novo konstanto ali pa pritisneno na črko, ki je definirana kot *hotkey*).

Če želimo izbrisati vsebino simulacijskega okna in pognati simulacijo od začetka, pritisnemo na ikono *Restart Simulation* in nato izberemo menu **Waveform ➔ Clear All Waveforms**.

2.

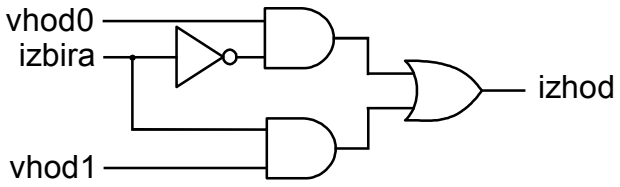
Načrtovanje vezij v jeziku VHDL

Obravnavali bomo načrtovanje nekaterih osnovnih kombinacijskih in sekvenčnih gradnikov digitalnih vezij v jeziku VHDL. Na primerih tipičnih vezij bo prikazano načrtovanje kombinacijskih vezij na nivoju logičnih funkcij in na višjem nivoju, RTL opis sekvenčnih vezij in strukturno načrtovanje kompleksnejših vezij. Na koncu vsakega poglavja so naloge, ki so namenjene za reševanje na laboratorijskih vajah.

2.1 Načrtovanje kombinacijskih gradnikov

Multiplekser

Primer kombinacijskega vezja je dvovhodni multiplekser, ki je shematsko prikazan na sliki 2.1. Multiplekser ima dva signalna vhoda, izbirni vhod ter en izhod. Na izhod se prenese vrednost enega izmed signalnih vhodov, glede na vrednost izbirnega vhoda. Če je na izbirnem vhodu logična ničla, se prenese na izhod vrednost iz prvega vhoda, sicer pa vrednost iz drugega vhoda.



izbira = 0	izhod = vhod0
izbira = 1	izhod = vhod1

Slika 2.1: Vezje dvovhodnega multiplekserja

Poljubno kombinacijsko vezje lahko zapišemo v jeziku VHDL v obliki logične funkcije. V primeru dvovhodnega multiplekserja preberemo funkcijo kar iz sheme vezja:

```
izhod <= (vhod0 and (not izbira)) or (vhod1 and izbira);
```

Za prirejanje vrednosti uporabljamo prireditveni operator <=. Na koncu vsakega stavka mora biti podpičje, podobno kot pri programskih jezikih C ali Pascal. Logično funkcijo zapišemo z logičnimi operatorji: and, or, not, nand, nor, xor, xnor. Rezervirane besede v jeziku VHDL bomo označevali s poudarjenim besedilom.

Opis kombinacijskega vezja v obliki logične funkcije je lahko zelo nepregleden. Že pri tako preprostem vezju, kot je dvovhodni multiplekser, težko vidimo delovanje vezja iz zapisa logične funkcije. Pri bolj kompleksnih vezjih pa lahko postane takšen zapis popolnoma nepregleden. Jezik VHDL vsebuje veliko konstruktov s katerimi opišemo delovanje vezja na bolj pregleden način. Vsako digitalno vezje je v končni fazi narejeno

iz osnovnih logičnih vrat, vendar lahko pri načrtovanju vezja opisujemo vezje na višjem nivoju in prepustimo sintezo logičnih vrat programski opremi.

Delovanje dvovhodnega multiplekserja lahko opišemo s pogojnim prireditvenim stavkom:

```
izhod <= vhod0 when izbira='0' else
         vhod1;
```

POGOJNI PRIREDITVENI STAVEK

```
signal <= izraz1 when pogoj1 else izraz2;

signal <= izraz1 when pogoj1 else
          izraz2 when pogoj2 else
          izraz3;
```

Signal izhod dobi vrednost signala vhod0 kadar je signal izbira na logični ničli, sicer dobi izhod vrednost signala vhod1. V tem primeru nam za opis vezja ni bilo potrebno poznati logične funkcije.

Demultiplekser

Zapišimo še vezje demultiplekserja z dvema izhodoma, ki opravlja nasprotno nalogo od multiplekserja - vhodni signal pripelje na en ali drugi izhod, glede na stanje izbirnega signala:

```
izhod0 <= vhod when izbira='0' else '1';
izhod1 <= vhod when izbira='1' else '1';
```

Z zgornjima stavkoma smo opisali delovanje vezja, sedaj pa moramo še dodati ogrodje VHDL opisa vezja. Opis kateregakoli vezja v jeziku VHDL je sestavljen iz dveh glavnih sestavnih delov: v prvem delu (**entity**) so opisani priključki vezja, v drugem delu (**architecture**) pa delovanje vezja. Na vrhu so navedene knjižnjice, v katerih so definirani podatkovni tipi, pretvarjanje med različnimi podatkovnimi tipi, dodatne funkcije ipd. Celoten opis demultiplekserja izgleda takole:

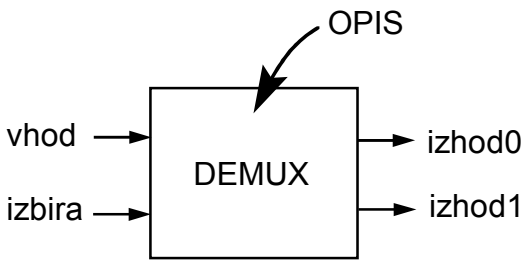
```
library IEEE;
use IEEE.std_logic_1164.all;

entity demux is
   port ( vhod, izbira: in std_logic;
          izhod0, izhod1: out std_logic );
end demux;

architecture opis of demux is
begin
 izhod0 <= vhod when izbira='0' else '1';
 izhod1 <= vhod when izbira='1' else '1';
end opis;
```

ZGRADBA VHDL PROGRAMA

```
entity ime_vezja is
   opis parametrov in signalov vezja
end ime_vezja;

architecture opis_vezja of ime_vezja is
begin
  stavki ...
end opis_vezja;
```

Program Active-HDL nam sam generira ogrodje VHDL opisa vezja, če za vsako novo vezje uporabimo čarovnik (*New Design Wizard*).

Vezje je s stavkom **entity** opisano kot komponenta z imenom demux in z definiranimi vhodnimi in izhodnimi signali. V jeziku VHDL mora imeti vsaka komponenta definirano eno ali več arhitektur, v katerih je opisano obnašanje vezja.



Slika 2.2: Demultiplekser kot komponenta

Priključki vezja so opisani s stavkom **port**. Vsak priključek ima definirano smer in tip signala.

- Vhodni signali se pojavljajo na desni strani prireditvenih stavkov v opisu vezja (lahko jih le beremo)

- Izhodni signali pojavljajo le na levi strani prireditvenih stavkov (lahko jih vpisujemo, ne moremo pa jih brati).

- Vhodno-izhodni signali se lahko pojavljajo na obeh straneh, uporabljamo pa jih za opis dvosmernih vodil.

- Izhodne signale z notranjo povezavo uporabljamo pri izhodih, ki jih nekje v vezju tudi beremo – pojavljajo se torej lahko na obeh straneh.

OPIS VHODNIH IN IZHODNIH SIGNALOV

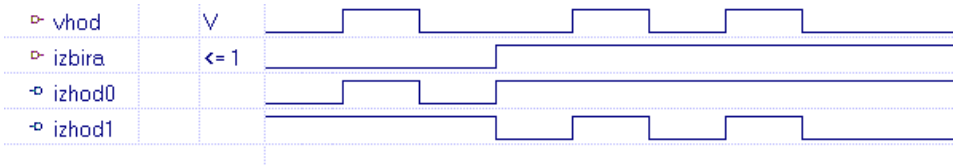
port (ime1: *smer tip_signala*;
ime2, ime3: *smer tip_signala*;
...
);

smer: vhodni signali **in**
izhodni signali **out**
vhodno-izhodni **inout**
izhodni z notranjo povezavo **buffer**

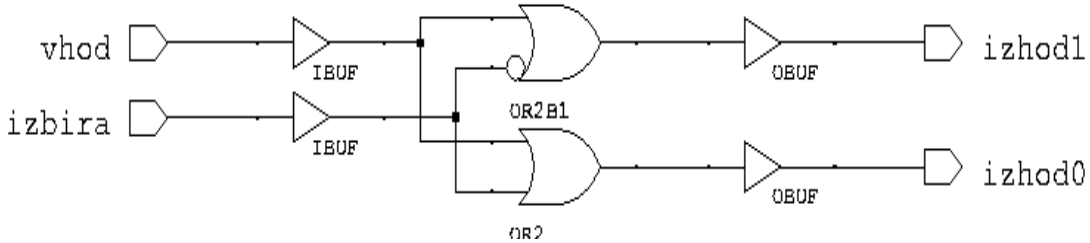
tip_signala: bit, bit_vector, std_logic ...

Osnovna podatkovna tipa, ki predstavljata signale (povezave) v vezju sta **bit** in **std_logic**. Signali tipa **bit** lahko zavzamejo le vrednost logične ničle ali enice. V primerih vezij bomo večinoma uporabljali podatkovni tip **std_logic**, ki je bolj primeren za simulacijo, ker nam med simulacijo lahko prikazuje tudi nedefinirano vrednost, kratek stik ipd. Kadar uporabljamo podatkovni tip **std_logic** moramo na začetku opisa vezja vključiti knjižnjico IEEE, kjer je ta podatkovni tip definiran. V nadaljevanju bomo obravnavali še vektorska podatkovna tipa: **bit_vector** in **std_logic_vector**.

Slika 2.3 prikazuje simulacijo demultiplekserja, na sliki 2.4 pa je vezje, ki smo ga dobili po sintezi iz VHDL opisa. Program za sintezo je iz pogojnih prireditvenih stavkov, ki opisujejo obnašanje demultiplekserja, generiral logična vrata.



Slika 2.3: Simulacija demultiplekserja



Slika 2.4: Sintetizirano vezje demultiplekserja

Naloga

1. Načrtajte v jeziku VHDL vezje dvobitnega multiplekserja, ki ima 4 vhodne signale, dva izbirna signala in en izhodni signal. Kombinacija na obeh izbirnih signalih naj določa, kateri vhodni signal gre na izhod.

3.

Operacije z vektorji

Primerjalnik

Primerjalnik je vezje, ki primerja dve binarni vrednosti na vhodu in ugotovi ali je prva vrednost večja, manjša ali enaka drugi vrednosti.

Za primerjavo vrednosti dveh signalov istega tipa uporabljamo v jeziku VHDL relacijske operatorje.

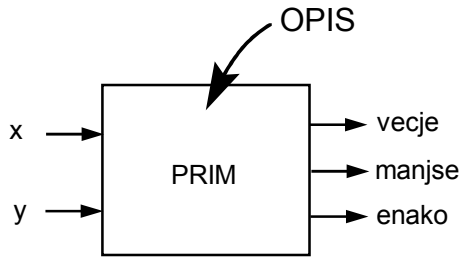
RELACIJSKI OPERATORJI

=	/=	>	>=	<	<=
enako	ni enako	večje	večje ali enako	manjše	manjše ali enako

Operatorja enako in ni enako sta definirana za vse tipe podatkov, ostali pa za skalarne in vektorske tipe.

Opis enobitnega primerjalnika v jeziku VHDL:

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

entity prim is
   port ( x, y: in std_logic;
          vecje, manjse, enako: out std_logic );
end prim;

architecture opis of prim is
begin
  vecje <= '1' when x > y else '0';
  manjse <= '1' when x < y else '0';
  enako <= '1' when x = y else '0';
end opis;
```



Slika 3.1: Enobitni primerjalnik

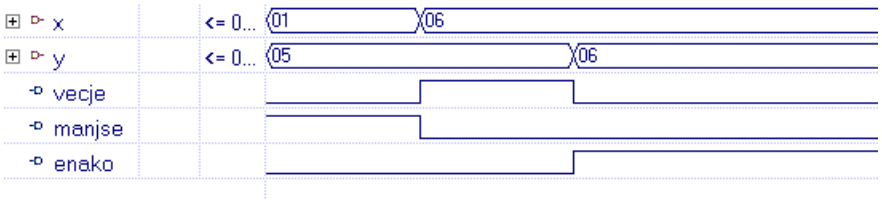
V praksi bolj pogosto srečujemo večbitne primerjalnike, npr. primerjalnik, ki primerja dve 8 bitni števili med seboj. Večbitni signali so v shematskih vezjih predstavljeni z vodili, kar je bolj praktično kot predstavitev s posameznimi povezavami. V jeziku VHDL so večbitni signali predstavljeni kot vektorji. Osnovna vektorska podatkovna tipa sta **bit_vector** in **std_logic_vector**. Pri deklaraciji vektorskega signala v stavku **port** navedemo tudi velikost vektorja (npr. 7 **downto** 0 za osem bitni vektor). Opis 8 bitnega primerjalnika.

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

entity prim8 is
   port ( x, y: in std_logic_vector (7 downto 0);
          vecje, manjse, enako: out std_logic);
end prim8;

architecture opis of prim8 is
begin
  vecje <= '1' when x > y else '0';
  manjse <= '1' when x < y else '0';
  enako <= '1' when x = y else '0';
end opis;
```

VEKTORJI

Deklaracija:
a: std_logic_vector (7 **downto** 0)
Prirejanje vrednosti:
a <= "01011001"
Posamezni element vektorja:
a(7) = '0'
Podvektor:
a(3 **downto** 0) = "1001"

Na sliki 3.2 je prikazana simulacija 8 bitnega primerjalnika. Simulator prikazuje vrednosti na celotnem vhodnem vodilu x oz. y, lahko pa vodilo razširimo in opazujemo posamezne signale.

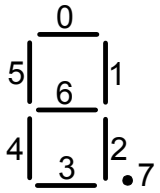


Slika 3.2: Simulacija 8 bitnega primerjalnika

Dekodirnik

Dekodirnik iz BCD v sedem segmentno kodo uporabljamo za prikaz 4 bitnega dvojiškega števila na sedem segmentnem prikazovalniku. Prikazovalnik krmilimo z 8 bitnim vodilom. Slika 3.3 prikazuje priključitev posameznih bitov vodila na segmente prikazovalnika. Naloga dekodirnika je pretvorba 4 bitne BCD kode, ki predstavlja vrednosti od 0 do 9, v signale za prižiganje ustreznih segmentov. Primer: število 1 (ali "0001" v 4 bitni BCD kodi) prikažemo tako, da postavimo segmenta 1 in 2 na logično enico, vse ostale pa na logično ničlo. Dekodirnik naj ima tudi vhod z imenom pika, ki je neposredno povezan na decimalno piko prikazovalnika (bit 7 na vodilu).

Za opis dekodirnika v jeziku VHDL je najbolj primeren **with...select** stavek s katerim zapišemo preslikavo iz ene kode v drugo. V našem primeru 10 različnih vrednosti iz BCD vhoda preslikamo v 10 sedem bitnih izhodnih vrednosti. Zadnji bit osem bitnega vodila (bit 7) pa je priključen na signal pika.



Slika 3.3: Sedem segmentni prikazovalnik

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

entity bin2led is
   port (
      vhod: in std_logic_vector (3 downto 0);
      pika: in std_logic;
      prikaz: out std_logic_vector (7 downto 0)
   );
end bin2led;

architecture opis of bin2led is
begin
 with vhod select
  prikaz(6 downto 0) <= "0000110" when "0001",
                        "1011011" when "0010",
                        "1001111" when "0011",
                        "1100110" when "0100",
                        "1101101" when "0101",
                        "1111101" when "0110",
                        "0000111" when "0111",
                        "1111111" when "1000",
                        "1101111" when "1001",
                        "0111111" when others;
 prikaz(7) <= pika;
end opis;
```

WITH...SELECT STAVEK

```
with signal1 select
  signal2 <= izraz2a when vrednost-signala1,
             izraz2b when vrednost-signala1,
             ...
             izraz2z when others;
```

Pomikanje vektorjev

Pomikanje vektorja v levo predstavlja množenje z 2, pomikanje v desno pa deljenje z 2. Napišimo program, ki pomakne 8 bitni vhodni vektor za eno mesto v levo, kadar je vhodni signal smer enak 0, oz. za eno mesto v desno, kadar je signal smer enak 1. Za sestavljanje vektorja iz ustreznega podvektorja in konstantne vrednosti ‘0’ smo uporabili združevalni operator.

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

entity pomik is
   port (
      vhod: in std_logic_vector (7 downto 0);
      smer: in std_logic;
      izhod: out std_logic_vector (7 downto 0)
   );
end pomik;

architecture opis of pomik is
begin
  izhod <= vhod(6 downto 0) & ‘0’ when
           smer='0'  else
           ‘0’ & vhod(7 downto 1);
end opis;
```

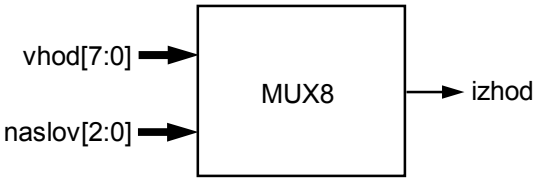
ZDRUŽEVALNI OPERATOR

Operator **&** se uporablja za združevanje vektorjev in signalov v večji vektor. Npr.:

```
a: std_logic_vector(3 downto 0)
b: std_logic_vector(1 downto 0)

a <= b &  “01”;
b <= ‘0’ & ‘1’;
a <= ‘0’ & b & ‘0’;
```

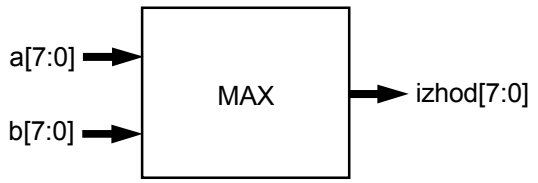
Nalogi

1. Zapišite v jeziku VHDL vezje multiplekserja z 8 signalnimi vhodi in enim izhodom. Multiplekser ima 3 naslovne vhode, ki določajo, kateri izmed osmih vhodov bo priključen na izhod.



Navodilo: indeks posameznega signala iz vhodnega vektorja je celo število (podatkovni tip **integer**). Vhodni signal naslov, ki je tipa std_logic_vector, lahko s funkcijo conv_integer( ) pretvorimo v celo število in ga uporabimo kot indeks za izbiro vhodnega signala. Funkcija za pretvorbo vektorja v nepredznačeno celo število se nahaja v knjižnjici IEEE.std_logic_unsigned, ki jo moramo vključiti v vezje poleg knjižnjice IEEE.std_logic_1164.

2. Načrtajte vezje za funkcijo maksimum. Vezja naj ima dva 8 bitna vhoda in en 8 bitni izhod. Na izhodu naj bo vedno maksimalna izmed obeh vhodnih vrednosti.

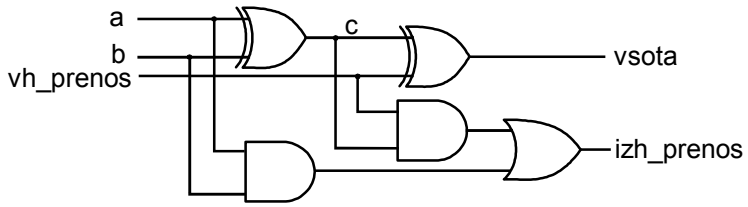


4.

Aritmetične in logične operacije

Polni seštevalnik

Zapišimo v jeziku VHDL vezje polnega seštevalnika, ki je shematsko prikazano na sliki 4.1. Vezje lahko preprosto opišemo z logičnimi funkcijami. Pri opisu vezja smo uporabili poleg vhodnih in izhodnih signalov tudi notranji signal (c). Dodatne signale deklariramo pri opisu arhitekture vezja.



Slika 4.1: Shema polnega seštevalnika

```
library IEEE;
use IEEE.std_logic_1164.all;
entity sest is
   port (
      a: in std_logic;
      b: in std_logic;
      vh_prenos: in std_logic;
      vsota: out std_logic;
      izh_prenos: out std_logic
   );
end sest;

architecture opis of sest is
  signal c: std_logic;
begin
  c <= a xor b;
  vsota <= c xor vh_prenos;
  izh_prenos <= (a and b) or (c and vh_prenos);
end opis;
```

NOTRANJI SIGNALI

Deklaracija notranjih signalov:

```
architecture opis_vezja of ime_vezja is
 signal ime: tip_signala;
 signal ime1, ime2: tip_signala;
begin
  stavki ...
end opis_vezja;
```

tip_signala: katerikoli podatkovni tip (std_logic, std_logic_vector, bit, integer...)

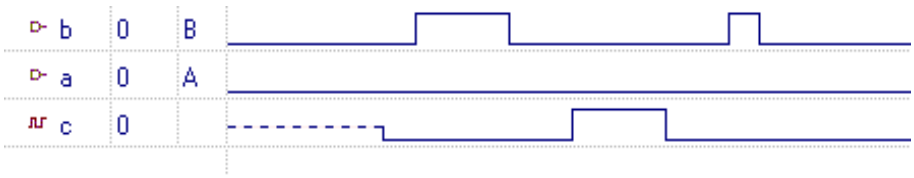
Modeliranje zakasnitev

Pri natančnem opisu digitalnih vezij moramo upoštevati tudi zakasnitve posameznih elementov in povezav v vezju. V jeziku VHDL lahko modeliramo zakasnitve v vezju na nivoju posameznih logičnih vrat (logičnih operacij v prireditvenih stavkih) ali pa na nivoju celotnih sklopov. Delovanje zakasnitev opazujemo pri simulaciji vezja, sinteza vezja iz VHDL opisa, ki vsebuje zakasnitve, pa ni mogoča.

Inercialni model zakasnitev uporabljamo za modeliranje zakasnitev aktivnih elementov, kot so logična vrata. V realnem vezju morajo biti vhodni signali prisotni nek minimalen čas, da bo izhod vezja reagiral na spremembo vhoda. Če je na vhodu prekratek impulz, vezje sploh ne bo reagiralo in bo izhod ostal v prejšnjem stanju – pravimo, da ima vezje neko inercijo. Poleg tega imamo še zakasnitev med vhodom in izhodom vezja, ki je posledica končnih preklopnih časov tranzistorjev. Npr. XOR vrata, pri katerih mora biti vhodni signal prisoten vsaj 2 ns in je izhod glede na vhod zakasnjen za 5 ns opišemo s stavkom:

```
c <= reject 2 ns inertial (a xor b) after 5 ns;
```

Na sliki 4.2 je simulacija XOR vrat, kjer smo vhod b spremenili iz 0 na 1 najprej za 3 ns potem pa za 1 ns. V prvem primeru je izhod dobil vrednost logične enice po 5 ns, v drugem primeru pa sploh ni reagiral na spremembo na vhodu.



Slika 4.2: Simulacija XOR vrat z inercialno zakasnitvijo

Pogosto uporabljamo poenostavljen inercialni model zakasnitev, kjer navedemo le zakasnitev med vhodom in izhodom vezja:

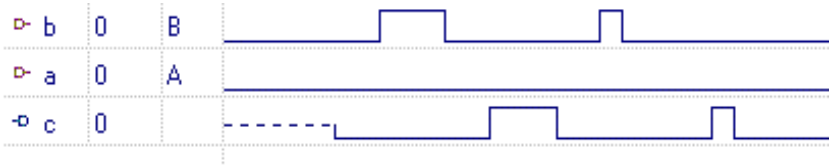
```
c <= (a xor b) after 5 ns;
```

Spremembe na vhodu, ki so krajše od 5 ns se pri izračunu XOR operacije ne bodo upoštevale, daljše spremembe na vhodu pa bodo povzročile spremembe izhoda z zakasnitvijo 5 ns.

Zakasnitve na povezavah modeliramo s transportnim modelom zakasnitev. Pri transportnem modelu nimamo inercije – še tako kratka sprememba na vhodu se pozna na izhodu z določeno zakasnitvijo.

```
c <= transport (a xor b) after 5 ns;
```

Na sliki 4.3 vidimo, da se na primeru XOR vrat s transportno zakasnitvijo vsaka sprememba na vhodu prenese na izhod z zakasnitvijo 5 ns.



Slika 4.3: Simulacija XOR vrat s transportno zakasnitvijo

Aritmetično logična enota

Aritmetično logična enota izvaja aritmetične in logične operacije nad vhodnima vektorjema. Vhodna vektorja in izhodni vektor naj bodo 8 bitni vektorji. Krmilni vektor ukaz pa naj določa vrsto operacije: logične in, ali in negacijo prvega vektorja, pomik prvega vektorja v levo ali v desno ter seštevanje in odštevanje vektorjev. Poleg izhodnega vektorja imamo tudi izhod ničla, ki je v stanju logične enice, kadar je izhod enak nič.

Operacije nad vektorji različnih tipov so definirane s funkcijami. Seštevanje večbitnih vektorjev opišemo podobno kot smo seštevanje enobitnih vrednosti s polnim seštevalnikom. Ker se osnovne aritmetične in logične operacije pogosto pojavljajo, so definirane v knjižnjicah s funkcijami, ki sprejmejo vektorje različnih dolžin in tipov. Za uporabo teh operacij moramo le vključiti ustrezne knjižnjice.

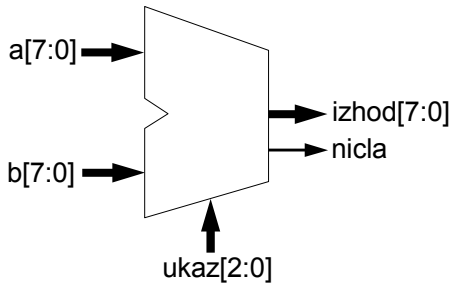
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity ale is
   port ( a,b: in std_logic_vector (7 downto 0);
          ukaz: in std_logic_vector (2 downto 0);
          izhod: out std_logic_vector (7 downto 0);
          nicla: out std_logic );
end ale;

architecture opis of ale is
 signal rez: std_logic_vector (7 downto 0);
begin
  with ukaz select
   rez <= a and b when "001",
          a or b when "010",
          not a when "011",
          a(6 downto 0) & '0' when "100",
          '0' & a(7 downto 1) when "101",
          a + b when "110",
          a - b when "111",
          "00000000" when others;
   nicla <= '1' when rez = "00000000" else '0';
   izhod <= rez;
end opis;
```

ARITMETIČNE OPERACIJE

Knjižnjica za aritmetične operacije:

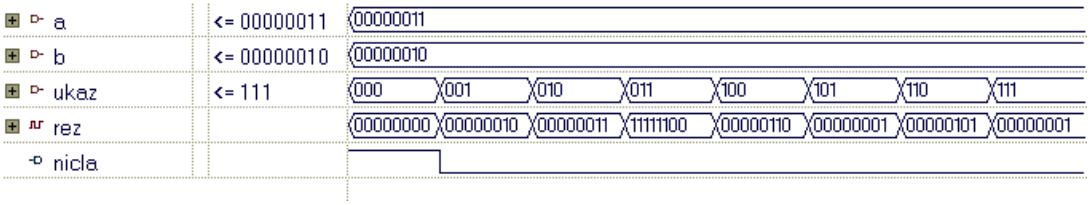
```
library IEEE;
use IEEE.std_logic_signed.all;
```

Vsota in razlika vektorjev:
vektor1 + vektor2
vektor1 – vektor2



Slika 4.4: ALE kot komponenta

Slika 4.5 prikazuje simulacijo aritmetično logične enote. Nastavili smo vhodna vektorja na vrednost 3 oz. 2 in pregledali izvajanje vseh ukazov.



Slika 4.5: Simulacija aritmetično logične enote

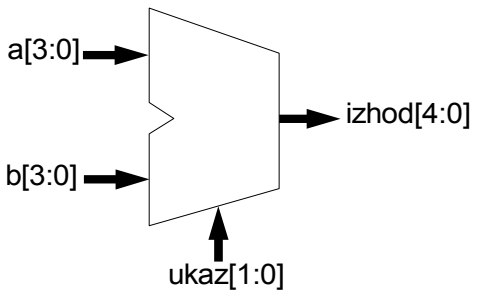
Nalogi

1. Spremenite VHDL opis polnega seštevalnika na nivoju logičnih vrat, tako da bodo upoštevane zakasnitve posameznih vrat v poenostavljenem inercialnem modelu zakasnitev. Vrata XOR naj imajo 4 ns, vrata AND in OR pa 2 ns zakasnitve med vhodom in izhodom. Opazujte vpliv zakasnitev pri simulaciji vezja.

Navodilo: za natančno modeliranje zakasnitev na vezju, moramo zapisati model za vsaka logična vrata posebej. To pomeni, da moramo poleg notranjega signala c dodati še dva notranja signala, ki bosta predstavljala ustrezno zakasnjena izhoda obeh AND vrat.

2. Načrtajte v jeziku VHDL aritmetično in logično enoto, ki ima na vhodu dve štiri bitni števili. Izhod naj bo 5 bitno število, pri katerem katerem naj najvišji bit pomeni prenos. Enota naj izračuna izhodne signale glede na vrednost 2 bitnega ukaza po tabeli:

ukaz	izhod
00	a
01	a xor b
10	a + b
11	a - b



Navodilo: pri seštevanju ali odštevanju vektorjev je velikost izhodnega vektorja vedno enaka velikosti večjega izmed obeh vektorjev. Prenosni bit na najvišjem mestu se pri takem izračunu izgubi. Če želimo ohraniti prenos, moramo vhodna vektorja pred operacijo podaljšati za eno mesto (na levi strani) – na dodatnem mestu se bo sedaj pojavil prenos pri seštevanju. Pri takšnem podaljševanju vektorjev moramo upoštevati, ali vektor predstavlja predznačeno ali nepredznačeno število.

5.

Sekvenčni gradniki

Flip-flopi

Sekvenčna vezja vsebujejo pomnilne elemente, ki jih opisujemo s procesnim stavkom. Znotraj procesnega stavka opisujemo delovanje vezja s sekvenčnimi stavki, ki se izvajajo v takšnem vrstnem redu, kot so zapisani (podobno kot pri običajnih programskih jezikih). Opis vezja lahko vsebuje več procesnih stavkov, zato dodamo na začetku procesnega stavka neko oznako, s katero ločujemo posamezne procese. Za rezervirano besedo **process** sledi (kot opcija) seznam signalov, na katere je procesni stavek občutljiv - to pomeni, da se bo proces izvajal le ob spremembi katerega izmed teh signalov.

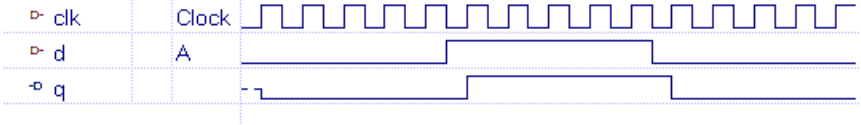
Poglejmo najprej opis D flip-flopa, ki je prožen s prvo fronto urinega signala. Ob prehodu ure iz 0 v 1 se mora podatek na vhodu flip-flopa prepisati na izhod, kar dosežemo s pogojnim (**if**) stavkom. Za detekcijo prehoda ure uporabljamo atribut *event*, ki ga pripišemo v k urinemu signalu. Pogoj clk'event je izpolnjen, kadar pride do spremembe urinega signala (prva in zadnja fronta). Če je po spremembi vrednost signala enaka 1, pomeni, da je prišlo do prehoda signala iz 0 v 1, kar pa je ravno prva fronta.

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

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

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

PROCESNI STAVEK

OZNAKA: **process** (*seznam*)
deklaracije notranjih signalov;
begin
sekvenčni stavki;
end process;

seznam: signali na katere je procesni stavek občutljiv



Slika 5.1: Simulacija robno proženega flip-flopa

Flip-flopi imajo običajno tudi reset vhod, s katerim postavimo izhod na vrednost 0. Vhod za resetiranje je lahko asinhroni (izhod se postavi na 0 takoj ob prisotnosti reset signala) ali pa sinhroni (izhod se postavi na 0 ob prisotnosti reset signala in prvi fronti urinega signala). Poglejmo najprej flip-flop z asinhronim reset signalom:

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

entity dff1 is
   port (d, clk, reset: in std_logic;
         q: out std_logic);
end dff1;

architecture opis of dff1 is
begin
 FF: process (clk, reset)
 begin
  if reset='1' then
    q <= '0';
  elsif clk'event and clk='1' then
    q <= d;
  end if;
 end process;
end opis;
```

POGOJNI STAVEK

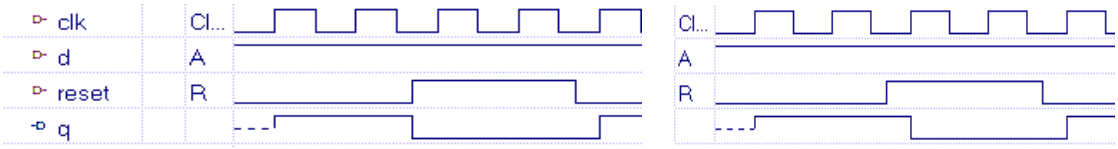
```
if pogoj1 then
 sekvenčni stavki;
end if;

if pogoj1 then
 sekvenčni stavki;
elsif pogoj2 then
 sekvenčni stavki;
elsif pogoj3 then
 ...
end if;
```

Flip-flop s sinhronim reset signalom:

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

entity dff2 is
   port (d, clk, reset: in std_logic;
         q: out std_logic);
end dff2;

architecture opis of dff2 is
begin
 FF: process (clk)
 begin
  if clk'event and clk='1' then
    if reset='1' then
      q <= '0';
    else
      q <= d;
    end if;
  end if;
 end process;
end opis;
```

POGOJNI STAVEK II

```
if pogoj1 then
 sekvenčni stavki;
else
 sekvenčni stavki;
end if;

if pogoj1 then
 sekvenčni stavki;
elsif pogoj2 then
 ...
else
 sekvenčni stavki;
end if;
```



Slika 5.2: Simulacija flip-flopa z asinhronim in sinhronim reset signalom

Register

Registre opišemo podobno kot flip-flope, le da uporabimo vektorske podatkovne tipe. Opis 8 bitnega registra z asinhronim reset signalom:

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

entity reg8 is
   port ( d: in std_logic_vector (7 downto 0);
          clk, reset: in std_logic;
          q: out std_logic_vector (7 downto 0));
end reg8;

architecture opis of reg8 is
begin
 REG: process (clk, reset)
 begin
  if reset='1' then
    q <= "00000000";
  elsif clk'event and clk='1' then
    q <= d;
  end if;
 end process;
end opis;
```

Struktura flip-flopa ali registra je osnova za zapis sekvenčnih vezij v jeziku VHDL, ki jih je mogoče sintetizirati. Dodati moramo le še kombinacijske funkcije za določanje vsakokratnih izhodov na registru glede na trenutne vrednosti na izhodih registra ter vrednosti vhodnih signalov.

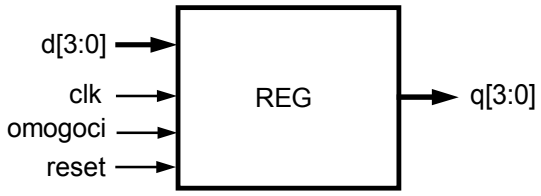
Dvojiški števec

Dvojiški števec dobimo tako, da izhod iz registra ob vsakem urinem impulzu povečamo za 1. Prištevanje enice je aritmetična operacija nad vektorji, zato moramo na začetku VHDL opisa števca vključiti knjižnjico IEEE.std_logic_signed. Pozorni moramo biti tudi na dejstvo, da se signal q (izhod števca) pojavlja na levi in na desni strani prireditvenega stavka, zato mora biti na začetku deklariran kot izhod z notranjo povezavo (**buffer**).

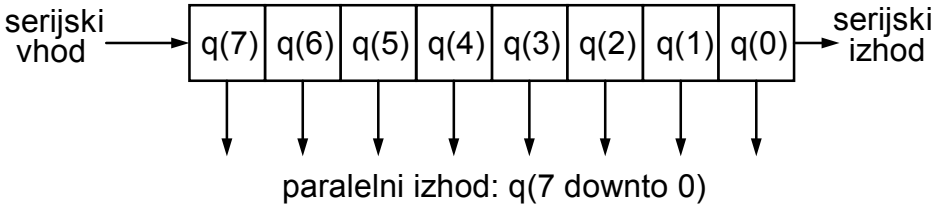
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_signed.all;

entity stevec is
   port ( clk, reset: in std_logic;
          q: buffer std_logic_vector (7 downto 0));
end stevec;

architecture opis of stevec is
begin
 ST: process (clk, reset)
 begin
  if reset='1' then
    q <= "00000000";
  elsif clk'event and clk='1' then
    q <= q + “00000001”;
  end if;
 end process;
end opis;
```

Nalogi

1. Načrtajte 4 bitni register, ki ima poleg vhoda in urinega signala še sinhroni reset signal in signal za omogočanje vpisa podatkov v register. Vhodni podatek naj se prepiše na izhod ob urinem impulzu in pogoju, da je signal za omogočanje vpisa enak 1.



2. Zapišite v jeziku VHDL 8 bitni serijsko paralelni pomikalni register. Pomikalni register naj ima serijski vhod in serijski ter paralelni izhod. Podatki se pomikajo ob urinem impulzu, ob resetu pa naj se asinhrono postavijo na "00000000".



6.

RTL opis sekvenčnih vezij

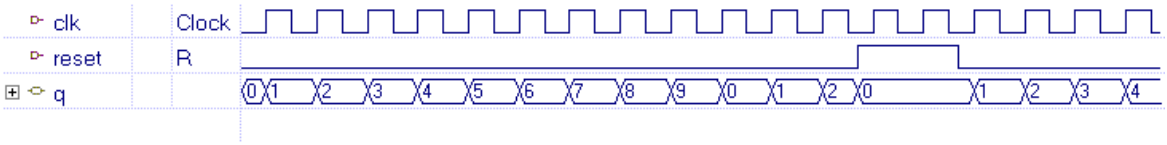
Programi za sintezo logičnih vezij iz VHDL opisa v splošnem ne morejo uspešno opraviti sinteze vezja, če se ne držimo nekaterih pravil pri pisanju vezja. Za sekvenčna vezja je najbolje, če uporabljamo RTL opis, pri katerem razdelimo vezje na kombinacijski del in pomnilni del.

BCD Števec

Števec, ki daje na izhodu BCD števila (dvojiška števila od 0 do 9), naredimo z uvedbo dodatnega pogojnega stavka, s katerim resetiramo števec, ko prišteje do 9. Na podoben način naredimo števec za poljubni modul štetja. Števec naj ima poleg ure in reset signala tudi dodaten vhod za omogočanje štetja.

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

entity stevec is
   port ( clk, reset, omogoci: in std_logic;
          q: buffer std_logic_vector (3 downto 0));
end stevec;

architecture opis of stevec is
begin
 S1: process(clk, reset)
 begin
   if reset='1' then
     q <= "0000";
   elsif clk'event and clk='1' then
     if omogoci = '1' then
       if q = "1001" then
         q <= "0000";
       else
         q <= q + "0001";
       end if;
     end if;
   end if;
 end process;
end opis;
```

Signal za omogočanje štetja bi lahko na prvi pogled zapisali kot dodaten pogoj v stavku, ki definira prvo fronto ure in s tem še bolj poenostavili opis BCD števca. Žal pa takšna struktura ni primerna za sintezo z vsemi programskimi orodji, ki zahtevajo v naprej predpisano obliko zapisa sekvenčnih gradnikov.



Slika 6.1: Simulacija števca po modulu 10

Če želimo BCD števec opisati na RTL nivoju, moramo razdeliti števec na pomnilni del, ki predstavlja 4 bitni register z asinhronim reset signalom ter na kombinacijski del, ki predstavlja logiko za izračun vhoda v register. Prvi procesni stavek opisuje 4 bitni register za katerega moramo posebej deklarirati vhodni signal d (notranji signal, ki prej ni bil potreben), drugi procesni stavek pa kombinacijsko logiko.

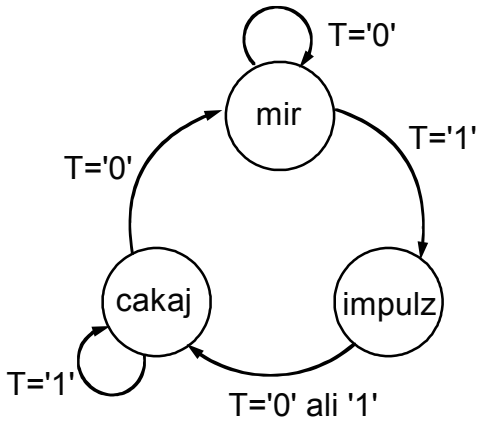
```
architecture RTLopis of stevec is
 signal d: std_logic_vector(3 downto 0);
begin

 R1: process(clk, reset)
 begin
   if reset='1' then
     q <= "0000";
   elsif clk'event and clk='1' then
     q <= d;
   end if;
 end process;

 S1: process(omogoci, q)
 begin
    if omogoci = '1' then
      if q = "1001" then
        d <= "0000";
      else
        d <= q + "0001";
      end if;
   else
      d <= q;
   end if;
end process;

end RTLopis;
```

V drugem procesnem stavku, ki predstavlja kombinacijsko logiko, moramo v čisto vseh primerih definirati signal d. To pomeni, da ima vsak if stavek tudi svoj else stavek, v katerem je definirana vrednost tega signala. V nasprotnem primeru bi se ob neizpolnjenem pogoju vrednost morala ohranjati in program za sintezo vezja bi nam generiral nivojsko proženi zapah (angl. latch), ki običajno ni zaželen element v vezju.

Pri opisovanju kombinacijskih gradnikov s procesnim stavkom moramo tudi paziti na seznam signalov na katere je proces občutljiv. Seznam mora vključevati vse signale, ki se pojavljajo v pogojih if stavkov in na desnih straneh prireditvenih stavkov. V našem primeru sta to signala omogoči in q.

Sekvenčni avtomat

Avtomati so tipična sekvenčna vezja, za katere uporabljamo RTL opis. Prikazali bomo načrtovanje avtomata za detekcijo pritiska tipke, ki ga prikazuje diagram prehajanja stanj na sliki 6.2. Avtomat ima en vhodni signal (T – tipka), prehodi med stanji pa naj se vršijo ob prvi fronti zunanje ure.



Slika 6.2: Diagram prehajanja stanj

Ob pritisku na tipko, gre avtomat iz mirovnega stanja v stanje impulz, ob naslednjem urinem ciklu pa v stanje čakaj, kjer ostane tako dolgo, dokler je tipka pritisnjena. Avtomat v stanju impulz generira izhodni signal, ki je dolg natanko eno urino periodo.

Za zapis stanj avtomata v jeziku VHDL definiramo nov podatkovni tip, v katerem naštejemo vsa možna stanja. Sekvenčni avtomat zapišemo običajno na RTL nivoju z dvema procesnima stavkoma. Pomnilni del avtomata predstavlja prvi proces, kjer se ob urinem impulzu stanje spremeni v naslednje stanje. Obe spremenljivki, ki hranita stanja sta deklarirani kot notranja signala. V drugem procesu določimo naslednje stanje v odvisnosti od trenutnega stanja in vhodnega signala ter izhodni signal, ki je odvisen le od trenutnega stanja.

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

entity avtomat is
   port (clk: in std_logic;
         reset: in std_logic;
         T: in std_logic;
         izhod: out std_logic );
end avtomat;

architecture RTLopis of avtomat is

  type vsa_stanja is (mir, impulz, cakaj);
  signal stanje, naslednje: vsa_stanja;

begin
```

NAŠTEVNI TIP PODATKOV

Deklaracija:

```
type ime_tipa is (ime1, ime2, ime3...);
signal  ime_signala: ime_tipa;
```

Prireditveni stavek:

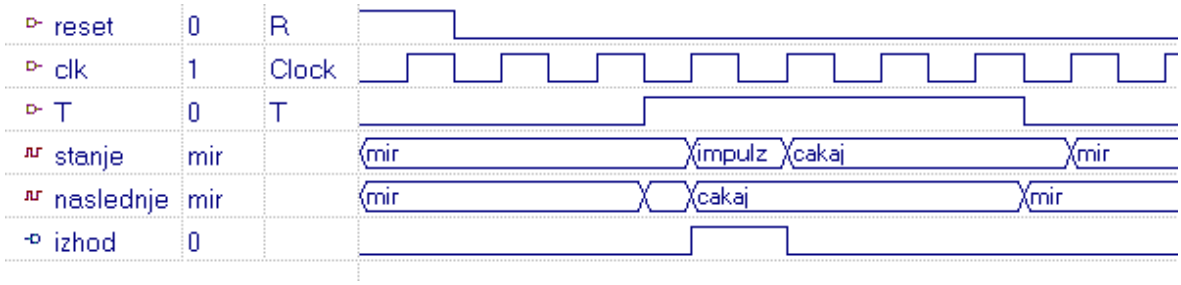
```
ime_signala <= ime1;
```

```
REG: process (clk, reset)
begin
    if reset='1' then
       stanje <= mir;
    elsif clk'event and clk='1' then
       stanje <= naslednje;
    end if;
end process;
```

```
PREHODI: process (stanje, T)
begin
    case stanje is
        when mir =>
           izhod <= '0';
           if T = '1' then
              naslednje <= impulz;
           else
              naslednje <= mir;
           end if;

        when impulz =>
          izhod <= '1';
          naslednje <= cakaj;

        when others =>
           izhod <= '0';
           if T = '0' then
              naslednje <= mir;
           else
              naslednje <= cakaj;
           end if;
     end case;
end process;

end RTLopis;
```

CASE STAVEK

```
case signal is
   when vrednost  =>
      stavki;
   when vrednost  =>
      stavki;
   when vrednost  =>
      stavki;
   when others  =>
      stavki;
end case;
```



Slika 6.3: Simulacija sekvenčnega avtomata

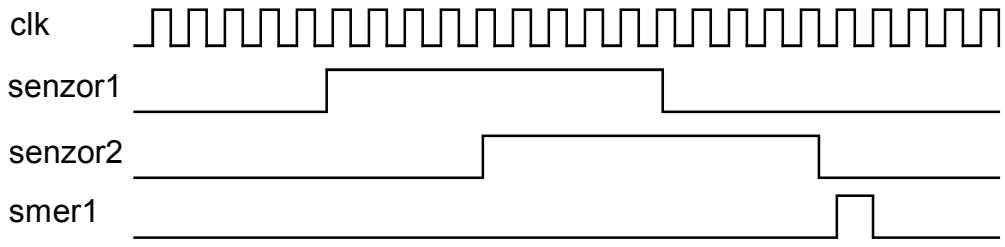
Kot smo videli, se nam pri opisovanju avtomata v jeziku VHDL s kodiranjem stanj sploh ni potrebno ukvarjati. Stanja moramo le našteti, potem pa delamo vse s simboličnimi imeni. Kodiranje stanj opravi namesto nas program za sintezo vezja. Običajno se uporablja binarno kodiranje (stanja predstavljajo zaporedne dvojiške

vrednosti) ali pa kodiranje z eno enico v zapisu vsakega stanja (angl. "one hot"). V prvem primeru potrebujemo najmanj bitov in kompleksnejšo logiko za izbiro naslednjega stanja, v drugem pa je register daljši, kombinacijski del pa enostavnejši.
Opisani avtomat ima dve potencialni pomanjkljivosti, ki se lahko pojavita v vezju z realnimi zakasnitvami. Prva pomanjkljivost je povezava asinhronega vhodnega signala (tipka) neposredno na vhod vezja, ki določa prehajanje stanj avtomata. Ob vsakem urinem impulzu se v pomnilni del avtomata (register) naloži naslednje stanje. Signal za naslednje stanje se lahko zaradi asinhronega vhoda spreminja ravno v kritičnem času nalaganja novega stanja v register, kar ima za posledico, da se v register naloži nedefinirana vrednost. Vrednost, ki se naloži v register, lahko pomeni čisto drugo stanje ali pa celo neuporabljeno kombinacijo – v obeh primerih bo delovanje vezja napačno in posledice nepredvidljive. Vsi vhodi v kombinacijski del avtomata morajo biti sinhronizirani z uro, s katero prehajajo stanja avtomata!

Druga potencialna nevarnost pa se skriva v izhodnem signalu, ki je odvisen od trenutnega stanja oz. kombinacije v registru avtomata. Zaradi zakasnitev v vezju se lahko pojavi motnja (angl. glitch) na signalu pri prehodu avtomata med dvema stanjema, v katerih je signal sicer neaktiven. Takšen signal ne smemo uporabiti kot uro ali asinhroni set / reset signal za druga sekvenčna vezja.

Naloga

1. Načrtajte avtomat za analizo signalov iz dveh senzorjev. Iz vsakega senzorja dobimo na vhodu vezja asinhroni signal, ki je aktiven v stanju logične enice. Avtomat naj ima dva izhoda: smer1 in smer2, na katerih se mora pojaviti impulz ob ustreznem zaporedju aktiviranja obeh senzorjev. Kadar se najprej aktivira prvi senzor in nato drugi, se mora pojaviti impulz na izhodu smer1, če pa se aktivira najprej drugi senzor in nato prvi, pa impulz na izhodu smer2.



Ob začetku in koncu sekvence morata biti oba senzorja neaktivna, vmes pa mora biti vsaj ena urina perioda ko sta oba senzorja aktivirana, sicer se sekvenca obravnava kot neveljavna in na izhodih ne smemo dobiti impulza.

7.

Strukturno načrtovanje

Kompleksnejša vezja načrtujemo s povezovanjem posameznih gradnikov, ki predstavljajo ločene VHDL opise vezij. Posamezni gradniki se v jeziku VHDL imenujejo komponente. Pri strukturnem načrtovanju vezij se osredotočimo na povezovanje komponent, ne pa na njihove funkcije, ki so opisane posebej.

Števec z dekodirnikom

Prikazali bomo strukturno načrtovanje BCD števca, ki ima na izhodu dekodirnik iz BCD v sedem segmentno kodo. Vzeli bomo kar VHDL opis BCD števca in dekodirnika iz prejšnjih poglavij, ki jih kot ločeni VHDL datoteki vključimo v projekt. Napisati moramo še eno VHDL datoteko, v katero vključimo števec in dekodirnik kot komponenti (ime komponente mora biti enako imenu VHDL datoteke) in ju povežemo med seboj s stavkom **port map**.

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

entity strukt is
        port (
                clk: in std_logic;
                reset: in std_logic;
                led: out std_logic_vector (7 downto 0)
        );
end strukt;

architecture opis of strukt is

 component bin2led
   port (
      vhod: in std_logic_vector (3 downto 0);
      pika: in std_logic;
      prikaz: out std_logic_vector (7 downto 0)
   );
 end component;

 component stevec
    port (clk, reset: in std_logic;
          q: buffer std_logic_vector (3 downto 0));
 end component;

 signal stev: std_logic_vector (3 downto 0);
 signal nula: std_logic;

begin
```

KOMPONENTE

Deklaracija:

```
component ime_komponente
  port ( ime: smer  tip_signala ...);
end component;
```

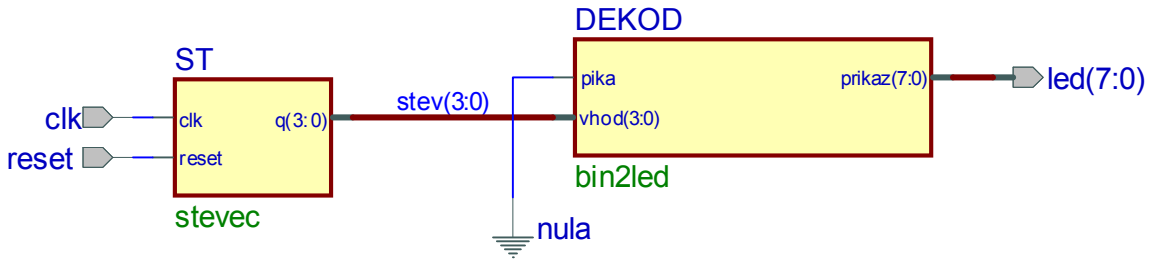
Povezovanje komponent:

```
oznaka: ime_komponente port map
  ( ime1 => IME1, ime2 => IME2 ... );
```

ime1, ime2: signali v komponenti
IME1, IME2: signali v vezju

```
        ST: stevec port map (clk => clk, reset => reset, q => stev);
        DEKOD: bin2led port map (vhod => stev, pika => nula, prikaz => led);

        nula <= '0';
end opis;
```

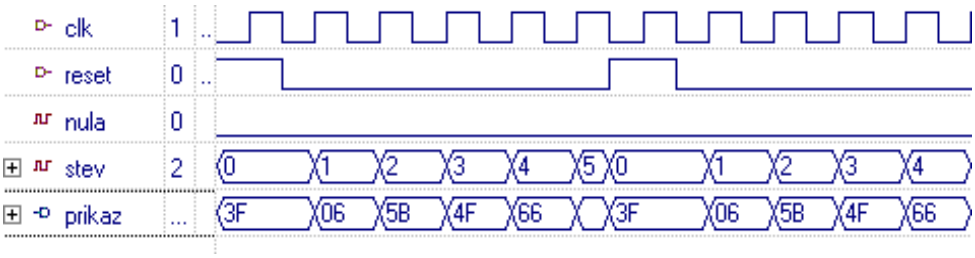
Deklaracijo komponent naredimo takoj za stavkom **architecture**. Pri deklaraciji moramo v stavku **port** navesti vse vhodne in izhodne signale komponente, kot so opisani v VHDL datoteki s komponento (najbolje je, da stavek **port** kar prekopiramo). Za povezovanje komponent smo uporabili vhodne in izhodne signale celotnega vezja ter notranje signale, ki smo jih posebej deklarirali.



Slika 7.1: Povezava BCD števca in dekodirnika

Boljši programi za načrtovanje digitalnih vezij omogočajo izbiro različnih načinov vnosa vezja (shema, VHDL program, diagram prehajanja stanj) in prehajanje med različnimi oblikami opisa vezja. V programu Active HDL lahko strukturno načrtovanje vezja in simulacijo izvedemo na nivoju sheme v urejevalniku blokovnih diagramov. Vezja, ki so vključena v projekt in prevedena, se pojavijo kot komponente, ki jih moramo le še povezati med seboj in dodati zunanje priključke. Shema se nato prevede v VHDL zapis, ki ga uporabimo pri sintezi vezja.

Predno začnemo izvajati simulacijo ali sintezo strukturnega VHDL opisa vezja, moramo v programskem orodju označiti glavno datoteko (tisto, ki vsebuje ostale datoteke kot komponente). Slika 7.2 prikazuje simulacijo števca z dekodirnikom.



Slika 7.2: Simulacija števca z dekodirnikom

Testne strukture

Testne strukture (angl. testbench) uporabljamo kot pripomoček za simulacijo vezja. Vezje, ki ga testiramo, je povezano kot komponenta znotraj testne strukture. Sama testna struktura običajno nima zunanjih signalov (stavek **entity** je prazen), pač pa le notranje signale, ki so povezani s testiranim vezjem.

Prikazali bomo izdelavo testne strukture, s katero bomo preverili naš avtomat za analizo signalov iz dveh senzorjev. V arhitekturnem stavku najprej deklariramo testirano vezje kot komponento, ki jo povežemo s testno strukturo. Deklariramo tudi vse notranje signale, ki jih potrebujemo za povezavo komponente in generiranje stimulatorjev.

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

entity avtomat_tb is
end avtomat_tb;

architecture TB_ARCHITECTURE of avtomat_tb is

component avtomat                          -- testirano vezje
    port(
        clk : in std_logic;
        reset : in std_logic;
        senzor1 : in std_logic;
        senzor2 : in std_logic;
        smer1 : out std_logic;
        smer2 : out std_logic );
    end component;

    signal clk : std_logic;                -- zunanji signali testiranega vezja
    signal reset : std_logic;
    signal senzor1 : std_logic;
    signal senzor2 : std_logic;
    signal smer1 : std_logic;
    signal smer2 : std_logic;

    signal ENDSIM: boolean;                -- signal za zaustavitev simulacije
    constant CLK_PERIOD: time := 33 ns;    -- frekvenca ure je 30 MHz

begin

    UUT : avtomat                          -- povezava testiranega vezja
    port map
        (clk => clk,
        reset => reset,
        senzor1 => senzor1,
        senzor2 => senzor2,
        smer1 => smer1,
        smer2 => smer2 );
```

```
    ENDSIM <= false, true after 2 us;              -- simulacija se izvaja 2 μs

    CLK_GEN: process                               -- proces za generiranje ure
        begin
        if ENDSIM=false
                then
                CLK <= '0';
                wait for CLK_PERIOD/2;
                CLK <= '1';
                wait for CLK_PERIOD/2;
        else
                wait;
        end if;
    end process;

    reset <= '1', '0' after CLK_PERIOD;

    senzor1 <= '0', '1' after 320 ns, '0' after 540 ns, '1' after 1010 ns, '0' after 1230 ns;

    senzor2 <= '0', '1' after 410 ns, '0' after 700 ns, '1' after 930 ns, '0' after 1100 ns;

end TB_ARCHITECTURE;
```

Za generiranje urinega signala napišemo poseben procesni stavek. V procesnem stavku CLK_GEN nismo definirali seznama signalov, na katere je proces občutljiv. Namesto seznama smo uporabili stavek **wait for**, s katerim suspendiramo izvajanje procesa za določen čas. Ko se izteče predvideni čas simulacije (ENDSIM = true) pa dokončno ustavimo proces s stavkom **wait**.

Definirati moramo še ostale vhodne signale v testirano vezje, za katere smo uporabili posebno obliko prireditvenih stavkov (z uporabo konstrukta **after** čas_konstanta), s katero lahko določimo poljubno časovno spreminjanje signala. V testnih strukturah lahko sicer uporabljamo vse VHDL konstrukte in stavke, ker jih uporabljamo le za simulacijo in nismo omejeni z naborom ukazov primernih za sintezo vezja.

Naloga

1. Skonstruirajte števec, ki šteje od 0 do 59 in izpisuje vrednosti na dveh LED prikazovalnikih. Števec je sestavljen iz števca enic, ki povečuje vrednost ob vsakem urinem impulzu in šteje od 0 do 9 ter števca desetic, ki poveča svojo vrednost ob prehodu števca enic iz 9 na 0. Izhoda obeh števcev sta vezana na dva pretvornika iz BCD v 7 segmentno kodo, ki ju vključite v vezje kot komponenti.

A.

Načrtovalska pravila pri sintezi v FPGA vezja

Za uspešno sintezo vezij v jeziku VHDL ne zadostuje le poznavanje sintakse jezika, ampak se moramo držati še nekaterih načrtovalskih pravil. Nekatera pravila so le splošni napotki za pisanje VHDL kode primerne za sintezo, druga pa so specifična za načrtovanje s FPGA vezji (upoštevajo lastnosti FPGA vezij, kot so: relativno velike zakasnitve na povezavah, signali za distribucijo ure, možnosti resetiranja sekvenčnih gradnikov ipd.).

1. Vsakemu notranjemu ali izhodnemu signalu moramo nedvoumno prirediti vrednost.

- Izhod kombinacijskega dela vezja mora biti nedvoumno definiran. Hitro se nam lahko zgodi, da pozabimo končni *else* v pogojnem prireditvenem stavku ali v *if* stavku.

Primer 1

```
dovoljen_prehod <= '1' when luc=zelena;
```

Vrednosti signala *dovoljen_prehod* nismo definirali za primer, ko pogoj ni izpolnjen. Pravilen zapis je:

```
dovoljen_prehod <= '1' when luc=zelena else '0';
```

Kombinacijska vezja lahko opisujemo tudi s procesnim stavkom, znotraj katerega uporabljamo *if* stavek. Tudi v tem primeru ne smemo pozabiti na končni else:

```
d: process(luc)
begin
 if luc=zelena then
   dovoljen_prehod <= '1';
 elsif luc=rdeca then
  dovoljen_prehod <= '0';
 end if;
end process;
```

Kaj pa če signal *luc* zavzame vrednost rumena? V tem primeru signal *dovoljen_prehod* spet ni popolnoma definiran.

- Signalu lahko v splošnem priredimo vrednost samo enkrat in samo na enem mestu (v enem sočasnem stavku). Signale v jeziku VHDL ne smemo mešati s spremenljivkami v drugih programskih jezikih, kjer lahko globalne spremenljivke nastavljamo kjerkoli v programu. Signali se obnašajo podobno kot spremenljivke le znotraj procesnega stavka, v katerem so sekvenčni (zaporedni) stavki. Le znotraj istega procesa lahko signalu večkrat priredimo vrednost.

Primer 2

```
p1: process(clk)
begin
 if clk'event and clk='1' then
  if tipka='1' then
    delam_reg <= '1';
  end if;
 end if;
end process;
```

```
p2: process(clk)
begin
 if clk'event and clk='1' then
    q <= q + 1;
    if q=5 then
      delam_reg <= '0';
    end if;
 end if;
end process;
```

Prvi proces nastavi signal *delam_reg* na '1' , drugi proces pa ga postavi nazaj na '0' ko je vrednost števca *q* enaka 5. Kaj pa če sta oba pogoja hkrati izpolnjena? Program za sintezo se v takšnih primerih ne more odločiti in javi napako: »signal *delam_reg* je dvakrat definiran«.

VHDL kodo moramo popraviti tako, da signalu *delam_reg* prirejamo vrednost samo v enem stavku (v našem primeru v enem procesnem stavku):

```
p1: process(clk)
begin
 if clk'event and clk='1' then
  if tipka='1' then
    delam_reg <= '1';
  elsif q=5 then
    delam_reg <= '0';
  end if;
 end if;
```

```
p2: process(clk)
begin
 if clk'event and clk='1' then
    q <= q + 1;
  end if;
end process;
```

- Edina izjema k zgornjemu pravilu so signali, ki predstavljajo priključke na tristanjsko vodilo. V tem primeru imamo skupaj vezanih več izhodov, kar pomeni, da je v VHDL opisu več prireditvenih stavkom z istim signalom na levi strani.

2. Za opis sekvenčnih gradnikov uporabljajmo le predpisano strukturo.

- Signali, ki naj predstavljajo registre v sekvenčnem vezju, se naj spreminjajo le ob prvi ali zadnji fronti ure in nikakor ne ob obeh frontah ali celo z več različnimi urami.

Primer 3

```
p: process(start, stop)
begin
   if start'event and start='1' then
      en_reg <= '1';
   elsif stop'event and stop='1' then
      en_reg <= '0';
   end if;
end process;
```

Takšen zapis zahteva dve različni uri (*start* in *stop*) za nastavljanje istega signala – sinteza takšnega zapisa ni mogoča!

V enem procesnem stavku imamo lahko le enkrat uporabljen atribut *'event* !

Rešitev 1: Popravimo specifikacijo vezja, tako da en signal deluje na fronto, drugi pa na nivo. Npr. signal *stop* uporabimo kot reset signal:

```
r1: process(start, stop)
begin
   if stop='1' then
      en_reg <= '0';
   elsif start'event and start='1' then
      en_reg <= '1';
   end if;
end process;
```

Rešitev 2: Z dovolj hitro uro vzorčimo signala *start* in *stop* ter naredimo avtomat, ki nam detektira prvo fronto enega ali drugega signala na podlagi časovnega zaporedja vzorčenih vrednosti. (Prva fronta nastopi tedaj, ko je bil zadnji vzorec še enak 0, novi vzorec pa je že 1)

- Pogojnemu stavku "*if clk'event and clk='1' then ...*" ne smemo dodati nobenih dodatnih pogojev – v ta namen moramo uporabiti nov pogojni stavek znotraj tega stavka.

Primer 4

```
p: process(clk)
begin
   if clk'event and clk='1' and q1_reg=9 then
     q2_reg <= d2;
   end if;
end process;
```

Program za sintezo takšno strukturo ne zna pravilno obravnavati. Dodatni pogoj (*q1_reg=9*) moramo zapisati znotraj stavka s *clk'event* atributom:

```
p: process(clk)
begin
   if clk'event and clk='1' then
 if q1_reg=9 then
   q2_reg <= d2;
      end if;
   end if;
```

3. Sinhrona vezja naj imajo minimalno število različnih urinih signalov.

- Pogoj za pravilno delovanje sinhronih sekvenčnih vezij je, da pride ura do vseh flip-flopov z enako zakasnitvijo. Zakasnitev na povezavah v FPGA vezju je lahko zelo velika, zato imajo programirljiva vezja posebne povezovalne vire za urine signale. Posebnih povezovalnih virov, ki imajo dostop do vseh flip-flopov v vezju z enako zakasnitvijo je omejeno število (običajno 4 do 8), zato je priporočljivo uporabljati v vezju čim manj različnih ur.

4. Za uro in asinhroni reset uporabljajmo le zunanje signale ali signale na izhodih flip-flopov

- Na signalih, ki prihajajo iz kombinacijske logike, se pogosto pojavljajo motnje zaradi različnih zakasnitev signalov na vhodih, zaradi uporabe večnivojske logike in zakasnitev na povezavah. Kombinacijski signali niso primerni za uro ali asinhroni reset, kjer lahko vsaka motnja povzroči nepravilno delovanje vezja.

5. Upoštevajmo omejitve pri resetiranju gradnikov v FPGA vezjih

- V FPGA vezjih Xilinx lahko uporabljamo le asinhroni reset ali set in ne oba hkrati.

Primer 5

```
p: process(clk, reset)
begin
   if reset='1' then
q1_ff <= '0';
q2_ff <= '1';
   elsif clk'event and clk='1' then
q1_ff <= ...
```

Prvi flip-flop (*q1_ff*) ima asinhroni reset, drugi pa asinhroni set. Ker gre za dva različna flip-flopa je vse v redu.

```
p: process(clk, set, reset)
begin
   if reset='1' then
q_ff <= '0';
   elsif set='1' then
q_ff <= '1';
   elsif clk'event and clk='1' then
q_ff <= ...
```

V tem primeru program za sintezo v FPGA vezja Xilinx javi napako, ker v teh vezjih ni na voljo flip-flopov z asinhronima reset in set signaloma. Enega izmed obeh asinhronih signalov moramo pretvoriti v sinhronega in ga postaviti znotraj *if clk'event*... stavka.

B.

Literatura

[1] S. Yalamanchili, "VHDL Starter's Guide", Prentice Hall, New Jersey, 1998, ISBN 0-13-519802-X

[2] K. Skahill, "VHDL for Programmable Logic", Addison-Wesley, 1996

[3] P. J. Ashenden, "The Designer's Guide to VHDL", MKP Inc., 1996

[4] D. L. Perry, "VHDL", New York, McGraw-Hill, 1993

[5] J. F. Wakerly, "Digital Design Principles and Practices", Prentice Hall, New Jersey, 2000, ISBN 0-13-769191-2