Andrej Trost

Načrtovanje digitalnih el. sistemov

Načrtovanje vezij v jeziku VHDL

Model pretoka podatkov in opis obnašanja vezja

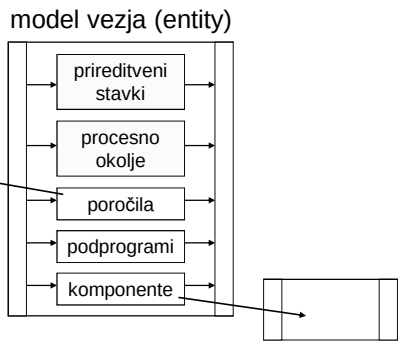
http://lniv.fe.uni-lj.si/ndes.html

Visokonivojski jeziki za opis vezij

- Standardizirani jeziki (IEEE standard)
  - VHDL (Very high-speed IC Hardware Description Language)
  - Verilog
- Modeliranje vezja na različnih nivojih
  - logična vrata, RTL, nivo obnašanja
- Funkcionalna in časovna simulacija vezja
  - s simulacijo preverimo delovanje vezja
- Sinteza vezja
  - pretvorba vezja na nivo logičnih vrat in flip-flopov

Stavki jezika VHDL

- sočasni stavki
- opis obnašanja
- simulacijski pripomočki
- hierarhično načrtovanje

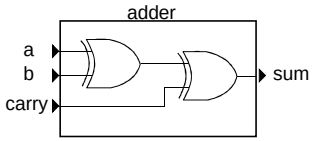


Osnovni koncepti jezika VHDL

- Model vezja je sestavljen iz:
  - opisa vmesnika (entity)
  - opisa delovanja (architecture)
- Osnovni elementi opisa vezja so signali, ki predstavljajo povezave
- Pri opisu vmesnika določimo zunanje signale in parametre vezja
  - vezje obravnavamo kot črno škatlo

Opis vmesnika (entity)

```
entity adder is
  port ( a, b : in bit;
         carry : in bit;
         sum : out bit );
end adder;
```

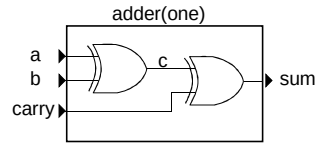


- Stavek port je deklaracija zunanjih signalov
- Za vsak signal določimo:
  - ime signala
  - smer (in, out, inout, buffer)
  - podatkovni tip (npr. bit)

Opis delovanja (architecture)

```
entity adder is
  port ( a, b : in bit;
         carry : in bit;
         sum : out bit );
end adder;

architecture one of adder is
  signal c : bit;
begin
   sum <= c xor carry;
   c <= a xor b;
end one;
```

deklaracija notranjega signala



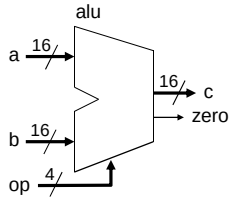
- V arhitekturnem stavku določimo notranje in izhodne signale

Zunanji signali

- Zunanji signali so enobitni (bit) ali vektorski signali (bit_vector)

```
entity alu is
 port ( a, b : in   bit_vector(15 downto 0);
        op   : in   bit_vector(3 downto 0);
        c    : out  bit_vector(15 downto 0);
        zero :out  bit );
end alu;
```

MSB LSB



- Podatkovni tip bit pozna dve vrednosti: 0 ali 1

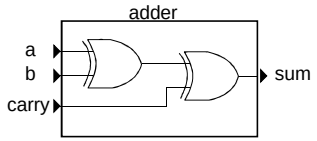
Večvrednostna logika

- Za bolj realistično simulacijo potrebujemo večvrednostno logiko
  - ‘U’ nedefinirano stanje
  - ‘X’ kratek stik
  - ‘Z’ visoka impedanca (odprte sponke)
  - ‘H’ pasivno visoko stanje (pullup)
  - ...
- Potrebujemo tudi funkcije za razrešitev
  - npr. ko signalu vsilimo vrednost ‘0’ in ‘Z’, prevlada vrednost ‘0’

Večvrednostna logika: std_logic

- Vse to omogoča podatkovni tip: std_logic in std_logic_vector
- Definiran je v knjižnici IEEE

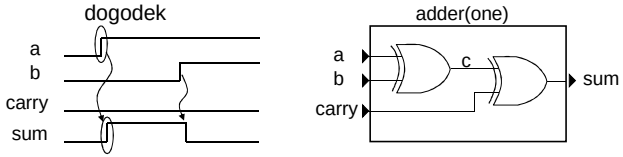
```
library IEEE;
use IEEE.std_logic_1164.all;

entity adder is
  port ( a, b : in std_logic;
         carry : in std_logic;
         sum : out std_logic );
end adder;
```



Obravnava VHDL modelov

- VHDL pozna dva načina obravnave modelov digitalnih vezij
- Simulacija modela
  - na simulatorju določimo spreminjaje vhodnih signalov in opazujemo izhode
- Sinteza modela
  - program za sintezo določi zgradbo vezja, ki izhaja iz VHDL modela

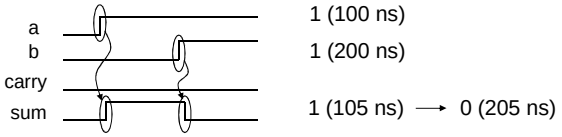
Simulacija modela vezja



- Pri simulaciji vezij se ukvarjamo s signali in njihovimi vrednostmi
- Dogodek je sprememba vrednosti signala
- Graf simulacije (`waveform`) je časovno urejeno zaporedje dogodkov

Model dogodkov

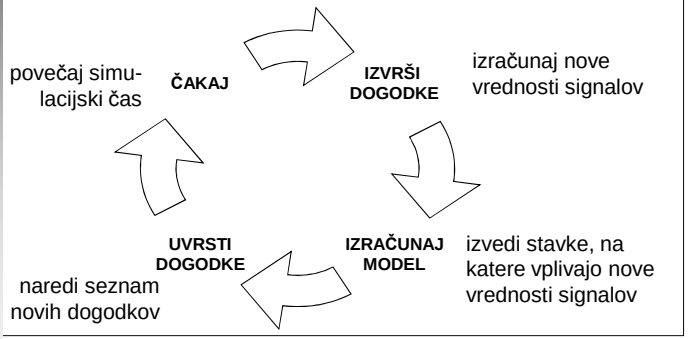
- Dogodki so opisani z vrednostjo signala in časom ob katerem se zgodijo



- Simulator izračunava dogodke na signalih in jih uvršča na seznam
- Dogodek se izvrši ob ustreznem sim. času

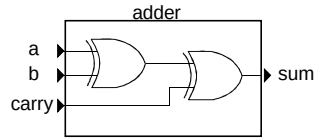
Delovanje simulatorja

- Simulator diskretnih dogodkov



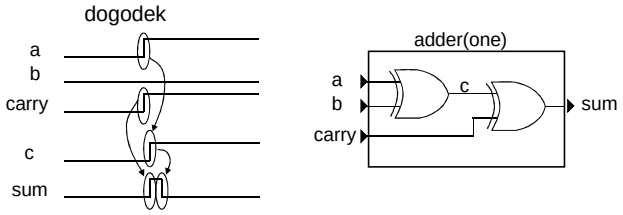
Potek simulacije

Vhodni dogodki:
a: 0, 1 (T)
carry: 0, 1 (T)



korak	čas	a	b	carry	sum	c
čakaj	0	0, 1(T)	0	0, 1(T)	0	0
izvrši	T	1	0	1	0	0
izračunaj	T	1	0	1	1(T+Δ)	1(T+Δ)
izvrši	T+Δ	1	0	1	1	1
izračunaj	T+Δ	1	0	1	0(T+2Δ)	1
izvrši	T+2Δ	1	0	1	0	1

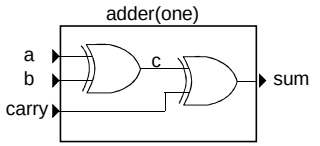
Časovni potek simulacije



- Vsak dogodek se izvrši z neko zakasnitvijo glede na vzrok, ki ga je povzročil
  - če v modelu ni realnih zakasnitev, se vzame korak simulacije Δ

Model vezja z zakasnitvami

```
entity adder is
  port ( a, b : in bit;
         carry : in bit;
         sum : out bit );
end adder;

architecture one of adder is
  signal c : bit;
begin
   sum <= c xor carry after 5 ns;
   c <= a xor b after 5 ns;
end one;
```



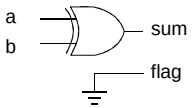
model zakasnitve

- V model vezja vnesemo (realne, ocenjene ?) zakasnitve

Opis pretoka podatkov

- Prireditveni stavek:

```
flag <= '0';
sum <= a xor b;
```



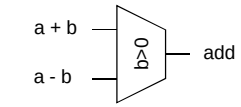
- Prireditveni stavek predstavlja logiko, katere izhod je signal, ki mu prirejamo vrednost

Za isti signal ne moremo imeti več sočasnih prireditvenih stavkov, ker bi opisali kratek stik!

Pogojni prireditveni stavek

- Signalu priredimo vrednost ali izraz ob določenem pogoju

```
mux <= a when select='0' else b;
flag <= '1' when a>b else '0';
add <= a+b when b>0 else a-b;
```



- Pogojni prireditveni stavek v osnovni obliki predstavlja izbiralnik (multiplekser)
  - izrazi predstavljajo logiko na vhodu izbiralnika, pogoj pa logiko za izbiro vhoda

Signali in konstante

- Zunanji signali so deklarirani v stavku port, notranji pa v arhitekturnem stavku
  - signalu lahko ob deklaraciji priredimo začetno vrednost
- Konstante deklariramo v arhitekturnem stavku

```
architecture one of test is

  signal flag: std_logic;
  signal reg: std_logic_vector(3 downto 0) := "0000";
  constant zero: std_logic := '0';

begin
```

Operacije z vektorji

- Logične operacije: **and, or, not, xor, xnor...**
- Sestavljanje vektorjev: **&**

```
a <= high & low;
b <= sign & "000" & low;
```

- Pomikanje vektorjev
  - pomik naredimo s sestavljanjem in podvektorjem

```
b  <=  a(6 downto 0) & '0';     -- pomik v levo
c  <=  '0' & a(7 downto 1);     -- pomik v desno
d  <=  a(6 downto 0) & a(7);  -- rotacija v levo
```

Aritmetične operacije z vektorji

- Aritmetične operacije: **+, -, ***
  - vključimo knjižnico: std_logic_unsigned (nepredznačena) ali std_logic_signed (predznačena števila)

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

architecture one of test is
  signal a, b, sum, inc, m: std_logic_vector(7 downto 0);
  signal d, e: std_logic_vector(3 downto 0);
begin
  sum <= a + b;     -- 8 bitni seštevalnik
  inc <= a + '1';   -- '1' se razširi na "00000001" in prišteje
  m <= d * e;       -- 4 x 4 biti = 8 bitni rezultat
```

Opis obnašanja vezja

- Osnovni element opisa je proces
  - vrstni red stavkov je pomemben (sekvenčni stavki)
  - pogojni stavek (if ... then ... else)
  - signalu lahko priredimo vrednost na več mestih, dejansko se izvrši le zadnja prireditev

```
primerjava: process (a, b)
begin
  enako <= '0';
  if a=b then
    enako <= '1';
  end if;
end process;
```

Zgradba procesnega okolja

```
oznaka: process (seznam signalov)
begin
.....

end process;
```

- Z oznako poimenujemo del vezja, ki ga predstavlja proces
- Simulacija procesa se izvrši ob spremembi enega izmed signalov iz seznama
  - pri opisu kombinacijskih vezij moramo navesti vse signale, ki predstavljajo vhode v proces

Pogojni stavek (if)

- Pogojni stavek spada med sekvenčne stavke
  - uporabljamo ga lahko le znotraj procesa

```
bin2bcd: process(bin)
begin
  if bin>9 then
    enice <= bin – 10;
    desetice <= '1';
  else
    enice <= bin;
    desetice <= '0';
  end if;
end process;
```

=

```
enice <= bin - 10  when bin>9
         else bin;

desetice <= '1' when bin>9
            else '0';
```

  - s pogojnim prireditvenim stavkom nastavljamo le en signal naenkrat!

Zaporedje pogojev: prioriteta

```
p: process(int0, int1, int2)
begin
  if int0='1' then
     v <= "01"
  elsif int1='1' then
     v <= "10";
  elsif int2='1' then
     v <= "11";
  else
     v <= "00";
  end if;
end process;
```

=

```
v <="01"  when int0='1' else
      "10"  when int1='1' else
      "11"  when int2='1' else
      "00";
```

int0 ima prednost pred int1, ki ima prednost pred int2 ...

- Zaporedni pogoji se ovrednotijo s prioriteto

Sekvenčni izbirni stavek (case)

- Odločamo se glede na vrednost enega signala
- Izbirni stavek nima prioritete

```
case ime_signala is
   when vrednost1 =>
      stavki(1);
   when vrednost2 =>
      stavki(2);
   ...
   when others  =>
      stavki(n);
 end case;
```

```
decod: process(digit)
begin
  case digit is
    when "00" =>
      display <= "00111111";
    when "01" =>
      display <= "00000110";
    when others  =>
      display <= "11111001";
  end case;
end process;
```

Sekvenčni stavki in sinteza vezja

- Prirejanje vrednosti signalom na več mestih
- Če signalu pod kakšnim pogojem ne določimo vrednosti, se bo ohranjala zadnja vrednost
  - dobimo sekvenčno vezje !

```
p: process(set_flag, clear_flag)
begin
  if set_flag='1' then
     flag <= '1';
  elsif clear_flag='1' then
     flag <= '0';
  end if;
end process;
```

Naredili smo asinhroni zapah za signal flag !

Opis kombinacijskih vezij

- Definiramo vrednosti pri vseh pogojih:
  - vrednost priredimo pri vsakem if in else ali
  - vrednost priredimo na začetku procesa in jo spremenimo v if stavkih

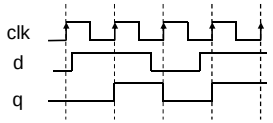
```
primerjava: process (a, b)
begin
   if a=b then
      enako <= '1';
   else
      enako <= '0';
   end if;
end process;
```

```
primerjava: process (a, b)
begin
   enako <= '0';
   if a=b then
      enako <= '1';
   end if;
end process;
```

Opis sekvenčnih vezij

- Sinhrona sekvenčna vezja spreminjajo izhode ob fronti ure
  - prva fronta: clk'event and clk='1' ali rising_edge(clk)
  - zadnja: clk'event and clk='0' ali falling_edge(clk)

```
reg: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
```



Sekvenčni elementi in sinteza vezja

- Program za sintezo zna narediti register ali flip-flop, če uporabimo ustrezno obliko opisa vezja

I. oblika: popolnoma sinhrono vezje

```
reg: process (clk)
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
```

II. oblika: asinhroni reset

```
reg: process (clk, reset)
begin
  if reset='1' then
    q <= "00000000";
  elsif rising_edge(clk) then
    q <= d;
  end if;
end process;
```

Povratne zanke

- Znotraj sinhronega procesa lahko uporabljamo povratno zanko
  - na podlagi stanja registra izračunamo novo stanje
  - Npr. števec, pomikalni register, sinhroni avtomat

I. oblika

```
stev: process (clk)
begin
  if rising_edge(clk) then
    q <= q + '1';
  end if;
end process;
```

- signal q uporabljamo v povratni zanki
- če je q zunanji signal, mora biti buffer

Primer: BCD števec

- BCD števec šteje od 0 do 9

I. oblika

```
stev: process (clk)
begin
  if rising_edge(clk) then
    if q < 9 then
      q <= q + '1';
    else
      q <= "0000";
    end if;
  end if;
end process;
```

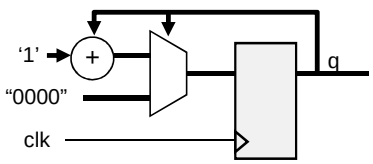
II. oblika

```
stev: process (clk)
begin
 if reset='1' then
     q <= "0000"
 elsif rising_edge(clk) then
   if q < 9 then
     q <= q + '1';
   else
     q <= "0000";
   end if;
 end if;
end process;
```

Zgradba vezja BCD števca

- Signal q je register, ki ima na vhodu logiko
  - povratna zanka je med izhodom registra in logiko

```
stev: process (clk)
begin
  if rising_edge(clk) then
    if q < 9 then
      q <= q + '1';
    else
      q <= "0000";
    end if;
  end if;
end process;
```



Registri in flip-flopi

- Prireditev vrednosti znotraj pogoja za fronto ure pomeni, da je signal register ali flip-flop

I. oblika

4 bitni register

flip-flop

```
accum: process (clk)
begin
  if rising_edge(clk) then
    acc <= acc + data;
    if acc="1111" then
      full <= '1';
    else
      full <= '0';
    end if;
  end if;
end process;
```

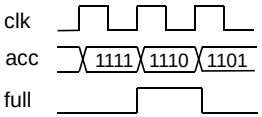
Potek simulacije

- Procesni stavek se izvede ob spremembi clk
  - pogoj rising_edge() je izpolnjen, ko gre clk iz 0 na 1

```
accum: process (clk)
begin
  if rising_edge(clk) then
    acc <= acc + data;
    if acc="1111" then
      full <= '1';
    else
      full <= '0';
    end if;
  end if;
end process;
```

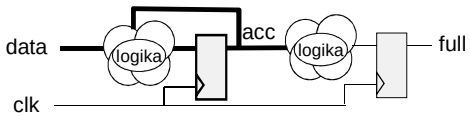
korak	čas	clk	data	acc	full
čakaj	0	0, 1	1111	0000	0
račun	T	1	1111	1111	0
izvrši	T+Δ	1	1111	1111	0
čakaj	T2	0, 1	1111	1111	0
račun	T2+Δ	1	1111	1110	1
izvrši	T2+Δ	1	1111	1110	1

Časovni diagram in zgradba vezja



- Signal full se postavi na '1' šele ob naslednji fronti ure
  - full je izhod iz flip-flopa !

- Vezje je zgrajeno v obliki cevovoda:



VHDL kviz: števec

- Kakšno je stanje števca, ko je reset='1' ?

```
s1: process (clk)
begin
 if rising_edge(clk) then
    stev <= stev + 1;
    if reset='1' then
      stev <= "0000";
    end if;
 end if;
end process;
```

```
s2: process (clk)
begin
 if rising_edge(clk) then
    if reset='1' then
      stev <= "0000";
    end if;
    stev <= stev + 1;
 end if;
end process;
```

a) "0000"
b) "0001"
c) stev+1

Celoštevilski podatkovni tip (integer)

- Tip integer: 32-bitna cela števila
  - pri deklaraciji lahko omejimo obseg (range)
- Uporaba: števci, konstante, indeksi vektorjev

```
architecture one of test is
  signal n: integer := 0;
  signal s: integer range 0 to 255;
begin

s1: process (clk)
begin
 if rising_edge(clk) then
    n <= n + 1;   -- 32 bitni števec
    s <= s + 1;   -- 8 bitni števec
```

Izbira podatkovnih tipov

- Zunanji signali realnega modela so vektorji
- Notranji sig. so lahko vektorji ali pa cela števila
- Za pretvorbo uporabimo ustrezne funkcije:

```
use IEEE.std_logic_arith.all;  -- conv_std_logic_vector()
use IEEE.std_logic_unsigned.all;  -- conv_integer()

architecture one of test is
  signal i, n: integer;
  signal a, b: std_logic_vector(7 downto 0);
begin
  i <= conv_integer(a);                  -- vektor v integer
  b <= conv_std_logic_vector(n, 8); -- integer v vektor
```

Pravila glede podatkovnih tipov

- V prireditvenih stavkih velja:
  tip in velikost signala = tip in velikost izraza

```
signal x, y: std_logic;
signal a: std_logic_vector(32 downto 0);
signal b,c: std_logic_vector(7 downto 0);
```

Pravilno	Napačno

x <= '0';
y <= x;
a<="00001111000011110000111100001111";
a <= (others =>'0'); -- vse bite na '0'
b <= X"A5"; -- šestnajstiška vrednost
c <= b;
	x <= "0"; -- x ni vektorski signal
	y <= a; -- y ni 32 bitni vektor
	a <= '0'; -- a ni enobitni signal
	a <= 0; -- a ni celo število
	b <= "0"; -- napačno število bitov
	c <= a; -- napačno število bitov

Obnašanje operatorjev

- Operatorji so dodatno definirani v knjižnicah!
  - angl. operator overloading
- Knjižnica IEEE in paketi:
  - std_logic_arith oz. numeric_std
  - std_logic_signed oz. std_logic_unsigned
- Primer:
  - kombiniranje std_logic_vector in integer

```
if stev < 9 then          -- rezultat odvisen od knjižnice!
    stev <= stev + 1;  -- lahko seštevamo vektor in celo število
```

Podatkovna struktura zbirka

- Z zbirko (array) modeliramo LUT, ROM, RAM
- Definiramo nov podatkovni tip (type), ki ga uporabimo pri deklaraciji signalov

```
type beseda is array (15 downto 0) of std_logic;
type tabela1 is array (0 to 1023) of std_logic_vector(15 downto 0);
type tabela2 is array (0 to 1023) of beseda;

signal b: beseda;
signal t1: tabela1;
signal t2: tabela2;
```

Tip std_logic_vector je narejen kot zbirka std_logic!

Opis ROM pomnilnika

- Definiramo zbirko in deklariramo signal
  - ob deklaraciji nastavimo začetne vrednosti signala

```
type tabela is array (0 to 3) of std_logic_vector(7 downto 0);
constant rom : tabela := (x"08", x"09", x"0A", x"00");
```

- Uporabimo indeks za posamezni element
  - če je indeks tipa std_logic_vector, ga pretvorimo v integer

```
podatek <= rom(conv_integer(naslov));
```

Opis RAM pomnilnika

- RAM deklariramo in beremo tako kot ROM
- Za pisanje v RAM zapišemo proces:

```
P: process(adr, data, wr)
begin
  if wr='0' then
    ram(conv_integer(adr)) <= data;
  end if;
end process;
```

model asinhronega SRAM

- Sinhroni RAM opišemo s sinhronim procesom