

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>

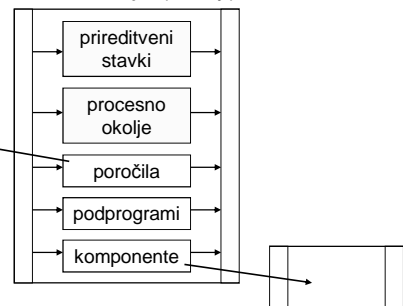
Visokonivojski jeziki za opis vezij

- 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

model vezja (entity)

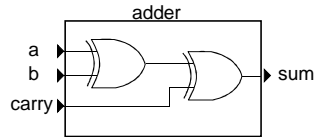


Osnovni koncepti jezika VHDL

- 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;
```

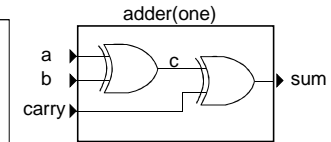


- Stavček port je deklaracija zunanjih signalov
- Za vsak signal določimo:
 - ime signala
 - smer (in, out, inout, buffer)
 - podatkovni tip (npr. bit)

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;
```



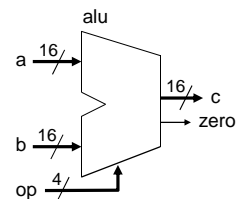
deklaracija notranjega signala

- 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;
```



- Podatkovni tip bit pozna dve vrednosti: 0 ali 1

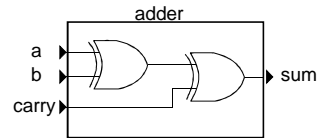
Večvrednostna logika

- 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

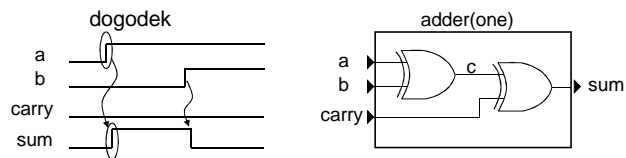
```
library IEEE;  
use IEEE.std_logic_1164.all;  
entity adder is  
  port ( a, b : in std_logic;  
        carry : in std_logic;  
        sum : out std_logic );  
end adder;
```



Obravnavanje 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

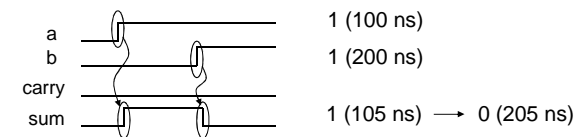
Simulacija modela vezja



- 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

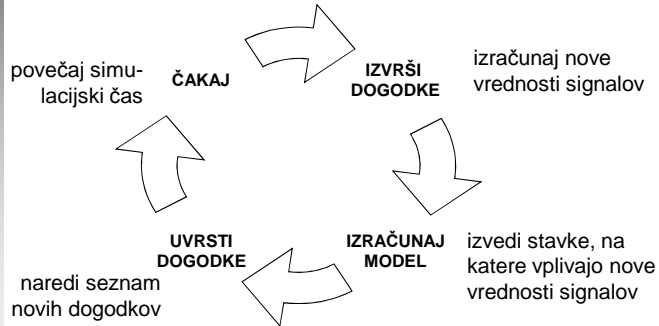
- Dogodki so opisani z vrednostjo signala in časom ob katerem se zgodijo



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

Delovanje simulatorja

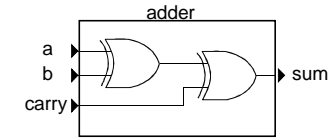
- Simulator diskretnih dogodkov



Potek simulacije

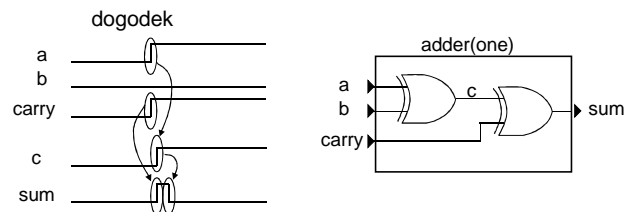
Vhodni dogodki:

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

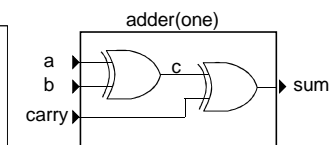


- Vsak dogodek se izvrši z neko zakasnitvijo glede na vzrok, ki ga je povzročil
 - če v modelu ni realnih zakasnitev, se vzame korak simulacije Δ

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;
```

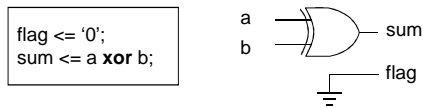


model zakasnitve

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

Opis pretoka podatkov

- Prireditveni stavek:

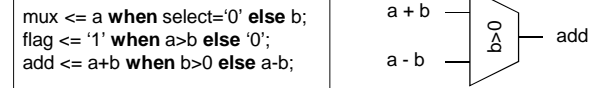


- Prireditveni stavek predstavlja logiko, katere izhod je signal, ki mu prirejamo vrednost

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



- 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 prireditvev

```
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 nastavljammo 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

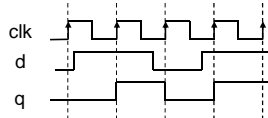
```
primerjava: process (a, b)
begin
  if a=b then
    enako <= '1';
  else
    enako <= '0';
  end if;
end process;
```

```
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

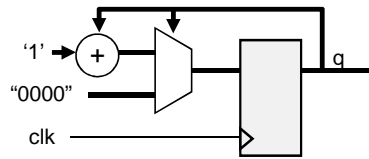
```
stev: process (clk)
begin
  if reset='1' then
    q <= "0000";
  elsif rising_edge(clk) then
    if q < 9 then
      q <= q + '1';
    else
      q <= "0000";
    end if;
  end if;
end process;
```


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

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

I. oblika

```

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;
    
```

4 bitni register → acc
flip-flop → full

Potek simulacije

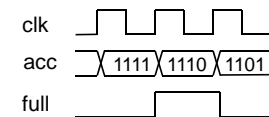
- Procesni stavek se izvede ob spremembi clk
 - pogoj rising_edge() je izpolnjen, ko gre clk iz 0 na 1

```

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:

