Andrej Trost

Načrtovanje digitalnih el. sistemov

Načrtovanje vezij v jeziku VHDL

Komponente, podprogrami in paketi

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

Hierarhično načrtovanje

- Veliko vezje razdelimo na posamezne gradnike
  - komponente
- Večkratna uporaba delov vezja
  - podprogrami
- Vnaprej pripravljeni gradniki
  - knjižnice in paketi

Komponente

- Model vezja je sestavljen iz para entity-architecture
- Modele vezja lahko povezujemo med seboj, tako da v opis vezja vključimo vezja kot komponente
  - takšen princip uporabljamo pri risanju sheme vezja
- Hierarhično načrtovanje
  - celotno vezje je sestavljeno iz komponent, ki so spet lahko zgrajene iz komponent...

Deklaracija komponent

- Komponente deklariramo na začetku arhitekturnega stavka (pred begin)
- Navedemo ime komponente in zunanje priključke (stavek port)

```
architecture struktura of test is
  component reg is
    port ( d : in std_logic_vector(7 downto 0);
           clk, en : in std_logic;
           q : out std_logic_vector(7 downto 0) );
  end component;
begin
```

Vključevanje komponente

- Komponento vključimo v vezje s stavkom port map
  - vsaki komponenti damo enolično oznako
  - v oklepaju navedemo povezave

```
begin
  r1: reg port map (d=>data, clk=>clk, en=>en1, q=>data_reg);
  r2: reg port map (d=>data, clk=>clk, en=>en2, q=>op_reg);
  ...
end struktura;
```

Vključevanje komponent (2)

```
oznaka: ime_komp port map (sk1=>sv1, sk2=>sv2, sk3=>open);
```

- Oznaka predstavlja ime dela vezja
  - oznake uporabljamo za poimenovanje procesov, komponent in ostalih gradnikov vezja
- Ime komponente je ime iz entity stavka
- Povezovanje: najprej navedemo ime signala na komponenti, ki ga priredimo signalu v vezju
  - če kakšen signal ni povezan, uporabimo open

Uporaba parametrov

- S parametri modeliramo družino vezij s podobno zgradbo
- Modeliramo spremenljive zakasnitve v vezju
  - zakasnitve so odvisne od tehnologije
- Modeliramo vezja s ponavljajočo strukturo
  - določimo velikost strukture v obliki parametra
- Modeliramo delovanje vezja
  - naredimo splošen model, ki mu natančno delovanje določimo s parametri

Deklaracija parametrov

- Parametre deklariramo s stavkom generic
  - parametre lahko uporabimo v port stavku in v arhitekturnem delu opisa vezja
  - pri deklaraciji navedemo privzeto vrednost

```
entity reg is
   generic (n: integer := 8);
   port ( d : in std_logic_vector(n-1 downto 0);
          clk, en : in std_logic;
          q : out std_logic_vector(n-1 downto 0) );
end reg;

architecture opis of reg is
...
```

Vključevanje vezij s parametri

- Uporabimo stavka port map in generic map
- Vrednost parametra določimo za vsako inštanco posebej
  - parametri se prenašajo po hierarhiji komponent
- Če ne uporabimo stavka generic map, se vzame privzeta vrednost parametra

```
r1: reg  generic map (n => 2)
         port map (d=>data, clk=>clk, en=>en2, q=>op_reg);
r2: reg port map (d=>data, clk=>clk, en=>en1, q=>data_reg);
```

Vezja z regularno strukturo

- Določene vrste vezij imajo regularno strukturo
  - register je sestavljen iz vrste flip-flopov
  - seštevalnik je iz vrste polnih seštevalnikov
  - paralelni množilnik in delilnik sta iz seštevalnikov
  - sistolična polja so iz matrike procesnih elementov
- Pri sestavljanju vezij z regularno strukturo uporabimo zanko za vključevanje komponent
  - z uporabo zanke je opis ponavljajoče se strukture zelo kompakten
  - lažje parametriziramo strukturo

Stavek generate

- S stavkom generate naredimo zanko v kateri določimo inštance komponent
- Znotraj zanke uporabljamo indeks (npr. i), ki ga ni potrebno posebej deklarirati
- Primer: iz 4 flip-flopov naredimo register

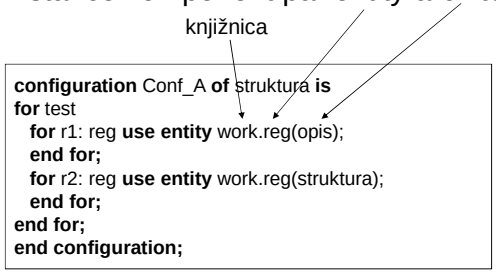
```
dreg: for i in (0 to 3) generate
  gen_reg: dff port map (d=>din(i), clk=>clk, q=>qout(i));
end generate;
```

Model vezja z več arhitekturami

- Model vezja ima lahko v jeziku VHDL več arhitekturnih stavkov
- Različne arhitekture se uporabljajo za različne načine opisa vezja, ki ima enake zunanje priključke
  - npr. visokonivojski model, RTL model, model vezja z ocenjenimi zakasnitvami
- Naenkrat lahko uporabljamo le en par entity - architecture

Deklaracija konfiguracije

- Z deklaracijo konfiguracije določimo za inštance komponent par entity-architecture




```
configuration Conf_A of struktura is
for test
  for r1: reg use entity work.reg(opis);
  end for;
  for r2: reg use entity work.reg(struktura);
  end for;
end for;
end configuration;
```

Podprogrami

- Podprograme uporabljamo za opis delov vezja, ki jih večkrat vključimo v celotno vezje
  - uporabjamo jih za opis podvezij, podobno kot komponente
- Podprogram je sekvenčno okolje
  - znotraj podprograma uporabljamo enake stavke kot znotraj procesnega okolja
- Pri deklaraciji navedemo zunanje signale in spremenljivke
  - navedemo vrsto, ime, smer in tip signala/sprem.

Opis podprograma

- Podprogram definiramo znotraj arhitekturnega stavka (pred begin)
  - Primer: opis flip-flopa v obliki podprograma

```
architecture struktura of test is
  procedure dff  (signal d, clk : in std_logic;
                  signal q, qn : out std_logic) is
  begin
    if rising_edge(clk) then
      q <= d;
      qn <= not d;
    end if;
  end procedure;
begin
```

Klic podprograma

- Podprogram kličemo v arhitekturnem stavku

```
architecture struktura of test is
  procedure dff  (signal d, clk : in std_logic;
  ...
begin
  dff(clk => clk, d => s1, q => r1, qn => open);
  ...
```

- Definiramo lahko podprograme z različnimi argumenti (število in tip) in enakim imenom
  - prevajalnik iz klica ugotovi, katero verzijo potrebuje

Knjižnice in paketi

- V knjižnicah in paketih hranimo modele, ki jih pogosto uporabljamo pri načrtovanju vezij
  - podprogrami, podatkovni tipi, konstante...
- Uporabljamo lahko vgrajene knjižnice (IEEE, STD) ali pa knjižnico naredimo sami
- Knjižnice vključujemo na vrhu opisa vezja
  - work je privzeto ime knjižnice v katero prevajamo vezja

```
library work;
use work.ime_paketa.all;
 ...
```

Primer lastnega paketa

- v delu package so definicije lastnih podatkov. tipov in konstant, deklaracije procedur…

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

package bcd is
   subtype bcd_vector is std_logic_vector(3 downto 0);
   type bcd3 is array (2 downto 0) of bcd_vector;
end bcd;

package body bcd is    -- telo paketa
end bcd;
```

Strukture za testiranje vezij

- Testiranje s simulacijo, kjer simulator ročno nastavljamo
  - omejene možnosti, predvsem za preprosta vezja
  - simulator praktično nikoli ne poženemo le enkrat !
- Testiranje s testno strukturo
  - v testni strukturi določimo časovni potek vhodov v vezje
  - ko je testna struktura narejena, jo večkrat uporabimo za izvedbo simulacije vezja
  - v simulatorju se ni potrebno ukvarjati z nastavljanjem vrednosti

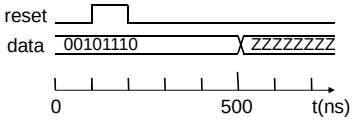
Testna struktura v jeziku VHDL

- Testna struktura - Test Bench
  - model vezja (entity – architecture), ki običajno nima zunanjih priključkov
- Testna struktura vključuje testirano vezje kot komponento
- Za generiranje stimulatorjev (vhodov v vezje) uporabljamo VHDL konstrukte brez omejitev
  - definiramo časovno spreminjanje signalov
  - uporabljamo stavke za izpis poročil med simulacijo
  - uporabljamo IO funkcije za branje in zapis datotek

Časovno spreminjanje signalov

- Signalu lahko priredimo več dogodkov, ki so vezani na čas:

```
reset  <= '0', '1' after 100 ns, '0' after 200 ns;
data <= "00101100", "ZZZZZZZZ" after 500 ns;
```

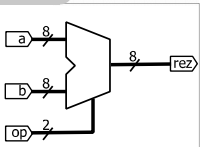


- Kadar je v testu veliko dogodkov uporabimo proces
  - Prirejanje dogodkov le v najbolj preprostih primerih, npr. za enkratne dogodke

Primer: test ALE

```
architecture behavior of test is
  component ale is
port ( a, b : in std_logic_vector(7 downto 0);
       op : in std_logic_vector(1 downto 0);
      rez : out std_logic_vector(7 downto 0) );
  end component;

signal a,b: std_logic_vector(7 downto 0);
signal op: std_logic_vector(1 downto 0);
signal rez: std_logic_vector(7 downto 0);
begin
```




```
UUT: ALE port map (a=>a, b=>b, op=>op, rez=>rez);
```


Test ALE (2): nastavljanje vhodov

- Napišemo proces v katerem spreminjamo vhode
  - časovni potek signalov določajo stavki wait for
  - uporabimo proces brez seznama signalov

```
stim_proc : process
  begin
   a <= "00000011";
   b <= "00000010";
   op <= "00";       -- operacija AND
   wait for 100 ns;
   op <= "01";       -- vsota
   …
   wait;     -- zamrzni proces, da se ne ponavlja
end process;
```


Generiranje periodičnih signalov

- Za generiranje ure uporabimo procesno okolje in stavke wait for
  - proces se izvaja, dokler ga ne ustavimo z wait

```
GEN_URE: process
  begin
    clk <= '0';
    wait for 50 ns;
    clk <= '1';
    wait for 50 ns;
end process;
```

```
GEN_URE: process
  begin
   if endsim=false then
      clk <= '0';
      wait for 50 ns;
      clk <= '1';
      wait for 50 ns;
   else wait;
end if;
end process;
```

Uporaba datotek

- Vrednosti signalov lahko preberemo iz datoteke ali zapišemo v datoteko
  - povezava simulacije z zunanjimi programi
- V testno strukturo vključimo paket TEXTIO, kjer so definirane funkcije
  - funkcija za odpiranje datoteke: file_open()
  - funkciji za branje: read(), readline()
  - funkciji za pisanje: write(), writeline()
  - zapiranje datoteke: file_close()

Poročila

- V testnem okolju preverjamo pogoje (assert) in izpišemo opozorilo (report)
- Med simulacijo se izpišejo opozorila
  - določimo lahko stopnjo opozorila (note, warning, error, failure)

```
assert qn = not q                          -- ali pogoj drži?
 report "Flip-flop ne deluje pravilno"     -- če ne, javi napako
 severity error;
```

Uporaba datotek v testni strukturi

```
stim_proc: process
  file vectors: text is in "podatki.txt";
  file outputs: text is out "rez.txt";
  variable ln: line;
  variable val: std_logic_vector(7 downto 0);
begin
  while not endfile(vectors) loop
     readline(vectors, ln);        -- read test vector
     read(ln, val);
     pin <= val;                   -- send value to input of UUT
     wait for clk_period;

     val := pout;                  -- get the outputs of UUT
     write(ln, val);               -- write value to file
     writeline(outputs, ln);
end loop;
```