Uvod u mikroračunarsku elektroniku

Predavanje IV

Sadržaj predavanja

- Naredbe instancioniranja komponenti
- GENERATE naredbe
- Obrada dizajna

Naredbe instancioniranja komponenti

Naredbe instancioniranja komponenti I

- Da bi smo opisali strukturnu implementaciju, moramo koristiti konkurentnu naredbu *instancioniranja komponente*, definisanu sintaksnim pravilom

naredba_instancioniranja_komponente ⇐
labela:
entity ime_entiteta [(identifikator_arhitekture)]
[**port map** (lista_asocijacije_portova)];

- Ova forma naredbe instancioniranja komponente izvodi direktno instancioniranje entiteta

- Možemo zamisliti da instancioniranje komponente vrši kreiranje kopije imenovanog entiteta, pri čemu odgovarajuće ahritekturno telo zamenjuje instancu entiteta

Naredbe instancioniranja komponenti II

- *Port mapa* definiše koji portovi entiteta su priključeni na odgovarajuće signale u navedenom arhitekturnom telu

- Sintaksno pravilo za listu asocijacije portova glasi

  lista_asocijacije_portova $\Leftarrow$
  ([ime_porta $\Rightarrow$] (ime_signala | izraz | **open**))$\{, \ldots\}$

- Svaki element u listi povezuje (asocira) po jedan port entiteta ili sa jednim signalom iz navedenog arhitekturnog tela ili sa vrednošću izraza ili ostavlja taj port otvorenim (neasociranim) što se specificira pomoću ključne reči *open*

Naredbe instancioniranja komponenti III

- Pretpostavimo da imamo entitet deklarisan kao

```
entity DRAM_controller is
    port (rd, wr, mem: in bit; ras, cas, we, ready: out bit);
end entity DRAM_controller;
```

- i odgovarajuću arhitekturu, *fpld*. Mogli bi smo kreirati instancu ovog entiteta na sledeći način:

```
main_mem_controller: entity work.DRAM_controller (fpld)
    port map (cpu_rd, cpu_wr, cpu_mem, mem_ras, mem_cas, mem_we, cpu_rdy);
```

- U ovom primeru, ime *work* odnosi se na tekuću radnu biblioteku u kojoj su smešteni entiteti i arhitekturna tela.
- Port mapa navodi signale iz odabranog arhitekturnog tela na koje su povezani portovi kopije entiteta
- Korišćena je *poziciona asocijacija*: svaki signal naveden u port mapi povezan je sa portom koji se nalazi na istoj poziciji u deklaraciji entiteta. Tako je signal *cpu_rd* povezan sa portom *rd*, signal *cpu_wr* sa portom *wr*, itd.

Naredbe instancioniranja komponenti IV

- Jedan od problema kada se koristi poziciona asocijacija je da nije odmah jasno koji signali se povezuju sa kojim portovima
- Da bi smo to utvrdili moramo pregledati deklaraciju entiteta
- Bolji način jeste da se koristi *imenovana asocijacija*, kao što je pokazano u sledećem primeru:

```
main_mem_controller: entity work DRAM_controller (fpld)
        port map (rd => cpu_rd, wr => cpu_wr, mem => cpu_mem, ready => cpu_rdy,
                  ras => mem_ras, cas => mem_cas, we => mem_we);
```

- Ovde je svaki port eksplicitno imenovan zajedno sa signalom sa kojim je spojen. Redosled navođenja je nebitan.
- Prednost ovog načina je u tome da je odmah jasno kako je entitet spojen sa strukturom odabranog arhitektunog tela.

Naredbe instancioniranja komponenti V

- U jednom od prethodnih predavanja bio je prikazan bihevijalni model za ivicom okidan flipflop
- Možemo ga iskoristiti kao osnovu za četvorobitni ivicom okidan registar
- Desno su prikazani deklaracija entiteta i struktuno arhitekturno telo četvorobitnog ivicom okidanog registra.

```
entity reg4 is
    port (clk, clr, d0, d1, d2, d3: in bit;
              q0, q1, q2, q3: out bit);
end entity reg4;

architecture struct of reg4 is
begin
    bit0: entity work.edge_triggered_Dff(behavioral)
                port map (d0, clk, clr, q0);
    bit1: entity work.edge_triggered_Dff(behavioral)
                port map (d1, clk, clr, q1);
    bit2: entity work.edge_triggered_Dff(behavioral)
                port map (d2, clk, clr, q2);
    bit3: entity work.edge_triggered_Dff(behavioral)
                port map (d3, clk, clr, q3);
end architecture struct;
```

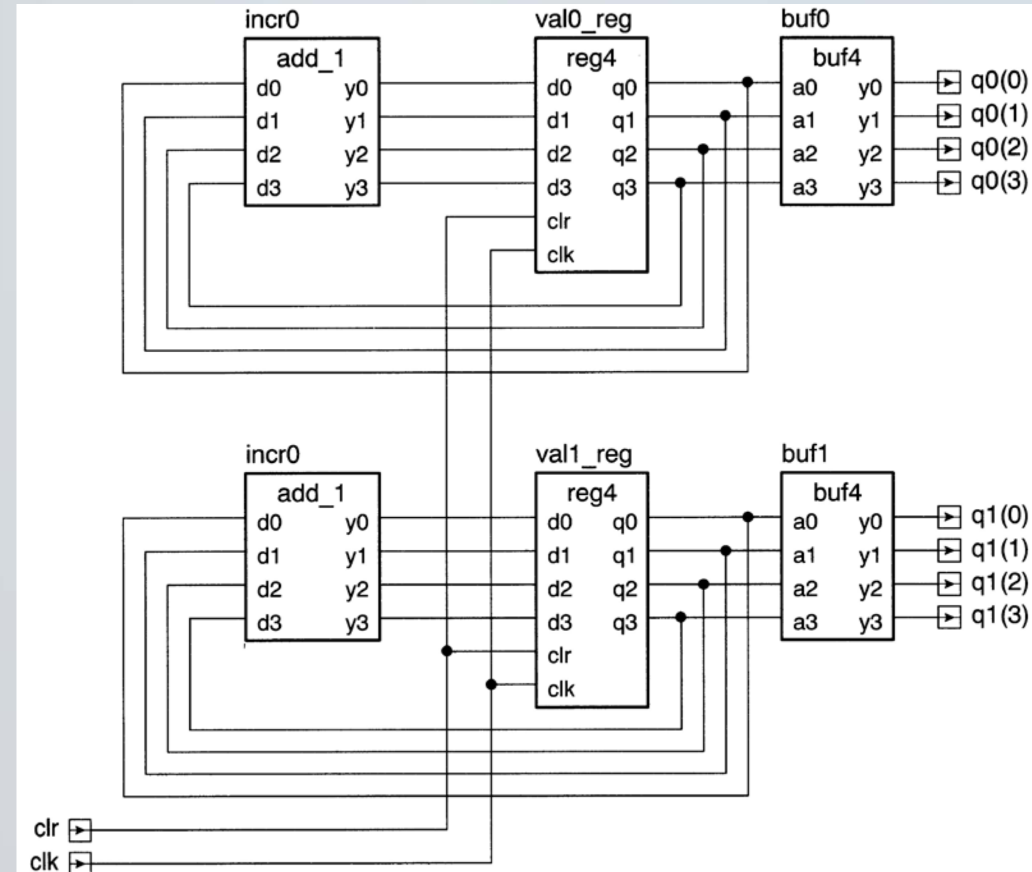
Naredbe instancioniranja komponenti VI

- Možemo koristiti ovaj entitet za registar, zajedno sa drugim entitetima, kao deo strukturne arhitekture za dvocifreni decimalni brojač predstavljen šemom desno

- Pretpostavimo da je cifra predstavljena kao bit vektor dužine četiri, opisan pomoću deklaracije podtipa

  subtype digit **is** bit_vector(3 **downto** 0);

- Model na sledećem prikazuje deklaraciju entiteta za brojač, zajedno sa okvirnim izgledom strukturnog arhitekturnog tela.



Naredbe instancioniranja komponenti VII

```
entity counter is
   port (clk, clr: in bit; q0, q1: out digit);
end entity counter;

architecture registered of counter is
   signal current_val0, current_val1, next_val0, next_val1: digit;
begin
   val0_reg: entity work.reg4(struct)
               port map (d0 => next_val0(0), d1 => next_val0(1), d2 => next_val0(2), d3 => next_val0(3),
                         q0 => current_val0(0), q1 => current_val0(1), q2 => current_val0(2), q3 => current_val0(3),
                         clk => clk, clr => clr);

   val1_reg: entity work.reg4(struct)
              port map (d0 => next_val1(0), d1 => next_val1(1), d2 => next_val1(2), d3 => next_val1(3),
                        q0 => current_val1(0), q1 => current_val1(1), q2 => current_val1(2), q3 => current_val1(3),
                        clk => clk, clr => clr);

   incr0: entity work.add_1(boolean_eqn)...;
   incr1: entity work.add_1(boolean_eqn)...;

   buf0: entity work.buf4(basic)...;
   buf1: entity work.buf4(basic)...;
end architecture registered;
```

Naredbe instancioniranja komponenti VIII

- Iz prethodnog primera videli smo da možemo asocirati posebne portove neke instance sa pojedinačnim elementima nekog signala koji je kompozitnog tipa, na primer niz ili struktura

- Ako instanca ima kompozitni port možemo asocirati pojedinačne signale sa pojedinim elementima porta. Ovo se ponekad naziva *subelementna asocijacija*.

- Na primer, ako instanca *DMA_buffer* ima port *status* tipa *FIFO_status* deklarisan kao

```
type FIFO_status is record
        nearly_full, nearly_empty, full, empty: bit;
end record FIFO_status;
```

- možemo asocirati signal svakom elementu tog porta na sledeći način:

```
DMA_buffer: entity work.FIFO
   port map (..., status.nearly_full ⇒ start_flush, status.nearly_empty ⇒ end_flush,
            status.full ⇒ DMA_buffer_full, status.empty ⇒ DMA_buffer_empty, ...);
```

Naredbe instancioniranja komponenti IX

- Možemo koristiti subelementnu asocijaciju za portove nizovnog tipa pišući indeksirano ime elementa na levoj strani asocijacije
- Štaviše, možemo asocirati deo porta sa signalom koji je jednodimenzionalni niz, kao što je prikazano u sledećem primeru
- Pretpostavimo da imamo entitet registra, deklarisan kao što je prikazano gore desno
- Portovi *d* i *q* su nizovi bitova. Arhitekturno telo za mikroprocesor, okvirno prikazano dole desno, instancionira gore pomenuti entitet kao program status registar (PSR).
- Individualni bitovi unutar registra predstavljaju uslovne i interapt flegove, a polje od bita 6 do bita 4 predstavlja tekući nivo prioriteta interapta.

```
entity reg is
   port (d: in bit_vector(7 downto 0);
          q: out bit_vector(7 downto 0); clk: in bit);
end entity reg;

architecture RTL of microprocessor is
   signal interrupt_req: bit;
   signal interrupt_level: bit_vector(2 downto 0);
   signal carry_flad, negative_flag, overflow_flag,
           zero_flag: bit;
   signal clk_PSR: bit;
   ...
begin
   PSR: entity work.reg
      port map (d(7) => interrupt_req,
                 d(6 downto 4) => interrupt_level,
                 d(3) => carry_flag, d(2) => negative_flag,
                 d(1) => overflow_flag, d(0) => zero_flag,
                 q => program_status,
                 clk => clk_PSR);
   ...
end architecture RTL;
```

Naredbe instancioniranja komponenti X

- Takođe možemo koristiti subelementnu asocijaciju za port koji je tipa neograničenog niza
- Granice indeksa za port su određene najmanjim i najvećim indeksom iz liste asocijacija, a smer indeksa određen je tipom porta
- Na primer, pretpostavimo da smo deklarisali entitet *and_gate* kao:

```
entity and_gate is
    port (i: in bit_vector; y: out bit);
end entity and_gate;
```

- i signale:

```
signal serial_select, write_en, bus_clk, serial_wr: bit;
```

- Možemo instancionirati entitet kao troulazno I kolo

```
serial_write_gate: entity work.and_gate
    port map (i(1) ⇒ serial_select, i(2) ⇒ write_en, i(3) ⇒ bus_clk, y ⇒ serial_wr);
```

Naredbe instancioniranja komponenti XI

- Ako imamo četvoroulazni multiplekser opisan sa deklaracijom entiteta

```
entity mux4 is
    port (i0, i1, i2, i3, sel0, sel1: in bit; z: out bit);
end entity mux4;
```

- možemo ga iskoristiti kao dvoulazni multiplekser instancionirajući ga na sledeći način:

```
a_mux: entity work.mux4
        port map (sel0 => select_line, i0 => line0, i1 => line1,
                  z => result_line, sel1 => ‘0’, i2 => ‘1’, i3 => ‘1’);
```

- Za ovu instancu komponente, selektorski ulaz *sel1* fiksiran je na vrednost '1', obezbeđujući da se ka izlazu prosleđuju samo linije *line0* ili *line1*

- Prateći preporuku, koja se daje za mnoge logičke familije, nekorišćene izlaze vezali smo na neki fiksni nivo, u ovom slučaju '1'.

Naredbe instancioniranja komponenti XII

- Neki entiteti mogu biti projektovani tako da ulazi ostanu otvoreni specificirajući podrazumevanu vrednost za port. Kada se entitet instancionira, možemo specificirati da je port otvoren koristeći ključnu reč *open* u listi asocijacija, kao što je pokazano u sintaksnom pravilu.

- Na primer, deklaracija *and_or_inv* entiteta uključuje podrazumevanu vrednost ‘1’ za svaki od svojih ulaznih portova, kao što je ponovo prikazano ovde

```
entity and_or_inv is
    port (a1, a2, b1, b2: in bit := '1'; y: out bit);
end entity and_or_inv;
```

- Možemo instancionirati komponentu koja će realizovati funkciju *not((A andB) or C)* koristeći gornji entitet na sledeći način:

```
f_cell: entity work.and_or_inv
            port map (a1 => A, a2 => B, b1 => C, b2 => open, y => F);
```

- Port *b2* ostavljen je otvorenim, pa zbog toga uzima podrazumevanu vednost '1' određenu u deklaraciji entiteta

Naredbe instancioniranja komponenti XIII

- Izlazni i bidirekcioni portovi mogu se ostaviti neasocirani korišćenjem ključne reči *open*, pod uslovom da nisu neograničenog tipa

- Ako se port moda *out* ostavi otvorenim, svaka vrednost koju bi entitet generisao na tom portu se ignoriše

- Ako se port moda *inout* ostavi otvorenim, vrednost koja se koristi unutar entiteta (efektivna vrednost) je vrednost koju obezbeđuje port

Naredbe instancioniranja komponenti XIV

- Poslednja stvar u vezi sa neasociranim portovima jeste da jednostavno možemo izostaviti neki port iz liste asocijacije da bi smo specificirali da je on otvoren. Na primer, ako imamo entitet deklarisan kao:

```
entity and3 is
    port (a, b, c: in bit := '1'; z, not_z: out bit);
end entity and3;
```

- instancioniranje komponente

```
g1: entity work.and3 port map (a ⇒ s1, b ⇒ s2, not_z ⇒ ctrl1);
```

- ima isto značenje kao i

```
g1: entity work.and3 port map (a ⇒s1, b ⇒ s2, not_z ⇒ ctrl1, c ⇒ open, z ⇒ open);
```

Alternativni način instancioniranja komponenti I

- Alternativno, komponente se unutar VHDL modela mogu instancionirati i korišćenjem deklaracije komponente
- Deklaracija komponente prosto specificira spoljašnji interfejs komponente u terminima generičkih konstanti u portova
- Za potrebe instancioniranja komponenti, ovo je zapravo sva potrebna informacija
- Ne interesuje nas konkretna implementacija funkcionalnosti koju modeluje komponenta već samo način kako se komponenta povezuje sa svojim okruženjem

Alternativni način instancioniranja komponenti II

- Sintaksno pravilo za deklaraciju komponente ima sledeći izgled:

  deklaracija_komponente ⇐

  component identifikator [**is**]

  [**generic** (generic_interfejs_lista);]

  [**port** (port_interfejs_list);]

  end component;

- Primer deklaracije komponente koji poštuje gorenavedeno pravilo je

```
component flipflop is
    generic (Tprop, Tsetup, Thold: delay_length);
    port (clk: in bit; clr: in bit; d: in bit; q: out bit);
end component flipflop;
```

- Ova deklaracija definiše komponentu koja predstavlja flipflop sa klok, clear i data ulazima, *clk*, *clr* i *d*, i data izlazom *q*

Alternativni način instancioniranja komponenti III

- Nakon što je komponenta deklarisana, možemo je instancionirati unutar odgovarajućeg VHDL modula

- Već smo pokazali kako je moguće direktno instancionirati odgovarajući entitet, a sada ćemo pokazati alternativni način instancioniranja prethodno deklarisane komponente

  naredba_instancioniranja_komponente ⇐
  labela:
  [**component**] ime_komponente
  [**generic map**] (lista_asocijacije_generika)]
  [**port map**] (lista_asocijacije_portova)];

- Sintaksno pravilo nam pokazuje da prilikom instancioniranja komponente jednostavno možemo iskoristiti ime komponente koju smo prethodno deklarisali unutar arhitekturnog tela, i eventualno, navesti asocijacije portova i generika sa konkretnim signalima i generičkim vrednostima. Labela je obavezna da bi se obezbedila jednoznačna identifikacija instance.

Alternativni način instancioniranja komponenti IV

- Strukturni model četvorobitnog registra, korišćenog ranije, može se napisati i korišćenjem deklaracije komponente flipflop

- Entity deklaracija i arhitekturno telo koje opisuje strukturu četvorobitnog registra korišćenjem komponente *flipflop* prikazano je desno

```
entity reg4 is
   port ( clk, clr : in bit; d : in bit_vector(0 to 3);
   q : out bit_vector(0 to 3) );
end entity reg4;

architecture struct of reg4 is
   component flipflop is
      generic ( Tprop, Tsetup, Thold : delay_length );
      port ( clk : in bit; clr : in bit; d : in bit; q : out bit );
   end component flipflop;
begin
   bit0 : component flipflop
      generic map ( Tprop => 2 ns, Tsetup => 2 ns, Thold => 1 ns)
      port map ( clk => clk, clr => clr, d => d(0), q => q(0) );
   bit1 : component flipflop
      generic map ( Tprop => 2 ns, Tsetup => 2 ns, Thold => 1 ns)
      port map ( clk => clk, clr => clr, d => d(1), q => q(1) );
   bit2 : flipflop
      generic map ( Tprop => 2 ns, Tsetup => 2 ns, Thold => 1 ns)
      port map ( clk => clk, clr => clr, d => d(2), q => q(2) );
   bit3 : flipflop
      generic map ( Tprop => 2 ns, Tsetup => 2 ns, Thold => 1 ns)
      port map ( clk => clk, clr => clr, d => d(3), q => q(3) );
end architecture struct;
```

GENERATE naredbe

GENERATE naredbe I

- Većina digitalnih sistema može se modelovati kao regularna iterativna struktura sastvaljena od odgovarajućih podsistema
- Memorije su dobar primer jer su organizovane kao 2D regularna matrica elementarnih ćelija
- Ovakve implementacije su preferirane, jer omogućavaju jednostavno kreiranje kompektnog, prostorno efikasnog fizičkog rasporeda na taj način smanjujući cenu proizvodnje
- Ukoliko se dizajn može iskazati kao repeticija odgovarajućih podsistema, bilo bi dobro ukoliko bi ga na taj način i mogli opisivati u VHDL-u; podsistem bi bio opisan samo jednom, a zatim bi se opisala iterativna struktura umesto pojedinačnog instancioniranja svake indicidualne komponente

GENERATE naredbe II

- VHDL obezbeđuje mehanizam za ovakvo iterativno modelovanje, korišćenjem *generate* naredbi

- *Generate* naredbe su konkurentne naredbe koje omogućavaju repliciranje drugih konkurentnih naredbi tokom elaboracije dizajna

- Postoje dve vrste generate naredbi:
  - **for generate** – za generisanje iterativnih struktura
  - **if generate** – za uslovno generisanje struktura

FOR GENERATE naredbe I

- Sintaksno pravilo za *for generate* naredbu glasi

  for_generate_naredba ⇐
  labela:
  for identifikator **in** diskretni_opseg **generate**
  telo_generate_naredbe
  end generate [labela]

  telo_generate_naredbe ⇐
  [{deklarativni_blok}
  begin]
  {konkurentna_naredba}
  [**end**;]

- Labela je obavezna ta bi se jednoznačno mogla identifikovati generate struktura
- Zaglavlje for-generate naredbe je slično sa zaglavljem for petlje i zapravo vrši istu funkciju

FOR GENERATE naredbe II

- Diskretni opseg specificira skup vrednosti, a za svaku vrednost, telo generate naredbe se replicira jednom
- U svakoj replikaciji koristi se tekuća vrednost diskretnog opsega, koja se naziva generate parametar
- U svakoj iteraciji generate petlje generate parametar je konstantan i pripada tipu koji je jednak baznom tipu diskretnog opsega
- Deklaracije koje možemo uključiti unutar generate naredbe su identične deklaracijama iz deklarativnog dela arhitekture i uključuju konkstante, tipove, podtipove, potprograme i signale
- Ovi objekti su zatim replicirani po jednom za svaku kopiju tela generate petlje

FOR GENERATE naredbe III

- VHDL model prikazan desno predstavlja model registra promenljive širine sa tristate izlazima
- Širina (broj bita) registra kontrolisana je preko generičke konstante *width*
- For-generate naredba replicira instancioniranje komponenti D_flipflop i tristate_buffer, pri čemu je njihov broj određen vrednošću generičke konstante *width*
- Za svaku kopiju, generate parametar *bit_index* uzima sukcesivne vrednosti iz opsega 0 do width-1
- Ova vrednost se koristi unutar svake kopije za određivanje koji će od elemenata data_in i data_out portova biti povezani sa flipflopom i tristate baferom

```
library ieee; use ieee.std_logic_1164.all;
entity register_tristate is
   generic (width: positive);
   port (clock: in std_logic; out_enable : in std_logic;
            data_in: in std_logic_vector(0 to width - 1);
            data_out: out std_logic_vector(0 to width - 1));
end entity register_tristate;

architecture cell_level of register_tristate is
   component D_flipflop is
      port (clk : in std_logic; d : in std_logic; q : out std_logic );
   end component D_flipflop;
   component tristate_buffer is
      port (a : in std_logic; en : in std_logic; y : out std_logic );
   end component tristate_buffer;
begin
   cell_array : for bit_index in 0 to width - 1 generate
      signal data_unbuffered : std_logic;
   begin
      cell_storage : component D_flipflop
         port map ( clk => clock, d => data_in(bit_index), q => data_unbuffered );
      cell_buffer : component tristate_buffer
         port map ( a => data_unbuffered, en => out_enable, y => data_out(bit_index) );
   end generate cell_array;
end architecture cell_level;
```

IF GENERATE naredbe I

- U većini digitlanih sistema sa iterativnom strukturom često je potrebno neke od ćelija tretirati na neki poseban način
- Ovo je čest slučaj sa ćelijama koje se nalaze na ivicama iterativne strukture
- Ove ćelije nemaju susede sa svih strana, kao što je slučaj sa ćelijama koje se nalaze unutar iterativne strukture
- Umesto toga, ivične ćelje povezane su sa signalima ili portovima iz okružujućeg arhitekturnog tela
- Ove specifičnosti ivičnih ćelija mogu se efikasno modelovati korišćenjem drugog tipa generate naredbe, **if generate** naredbe

IF GENERATE naredbe II

- Sintaksno pravilo *if generate* naredbe glasi:

```
if_generate_naredba ⇐
        labela:
        if uslov generate
            telo_generate_naredbe
        { elsif uslov generate
            telo_generate_naredbe}
        [ else generate
            telo_generate_naredbe]
        end generate [labela];
```

- If generate naredba podseća na standardnu if naredbu, sa tom razlikom da ovde uslovi kontrolišu koje su deklarativne i konkurentne naredbe kopirane u dizajn

IF GENERATE naredbe III

- Kada se model elaborira, uslovi u if-generate naredbama se proveravaju, počevši od prvog do poslednjeg, dok se ne pronađe onaj koji je zadovoljen
- Deklaracije, ukoliko postoje, i sve konkurentne naredbe u odgovarajućem telu generate naredbe se zatim uključuju u elaborirani model
- Ukoliko nijedan od uslova nije zadovoljen, a postoji else generate grana, deklaracije i naredbe iz ove grane uključuje se u model
- Else grana je opciona, što dozvoljava mogućnost da se u elaborirani model ne uključe nikakve deklaracije i naredbe, ukoliko ni jedan od if-generate uslova nije zadovoljen
- Labela u okviru if-generate naredbe je obavezna, da bi se omogućila jednoznačna identifikacija strukture koja se generiše

IF GENERATE naredbe IV

- VHDL model prikazan desno ilustruje kako bi mogao izgledati model ripple-carry sabirača registra baziran na korišćenju if-generate naredbe

- Širina generisanog ripple carry sabirača određena je vrednošću generičkog parametra *width*

- Ripple carry sabirač koristi komponentu polusabirač na poziciji najmanje značajnosti i različita povezivanja sa carry-in i carry-out (c_in i c_out) portovima za pune sabirače koji se nalaze unutar ripple-carry sabirača i za pun sabirač koji se nalazi na poziciji najveće značajnosti

```
entity ripple_carry_adder is
   generic (width: positive );
   port (a, b: in std_logic_vector(width – 1 downto 0);
          s: out std_logic_vector(width - 1 downto 0);
          c_out: out std_logic);
end entity ripple_carry_adder;

architecture struct of ripple_carry_adder  is
begin
   adder: for i in width-1 downto 0 generate
      signal carry_chain : unsigned(width-1 downto 1);
   begin
      adder_cell: if i = width-1 generate -- most-significant cell
         add_bit: component full_adder
            port map (a => a(i), b => b(i), s => s(i),
                        c_in => carry_chain(i), c_out => c_out);
      elsif i = 0 generate -- least-significant cell
         add_bit: component half_adder
            port map (a => a(i), b => b(i), s => s(i),
                        c_out => carry_chain(i+1));
      else generate -- middle cell
         add_bit: component full_adder
            port map (a => a(i), b => b(i), s => s(i),
                        c_in => carry_chain(i), c_out => carry_chain(i+1));
      end generate adder_cell;
   end generate adder;
end architecture struct;
```

Obrada dizajna

Obrada dizajna

- VHDL opis dizajna obično se koristi za simulaciju dizajna ili za sintezu odgovarajućeg hardvera
- Oba ova postupka zahtevaju obradu opisa pomoću programskih alata da bi se kreirao imulacioni program ili pak netlista za hardver koji zatim treba napraviti
- I simulacija i sinteza zahtevaju dva pripremna koraka:
  - *analizu* i
  - *elaboraciju*
- Simulacija zatim uključuje izvršavanje elaboriranog modela, dok sinteza zatim kreira net_listu primitivnih kola koja obavljaju istu funkciju kao i elaborirani model

Analiza dizajna I

- Prvi korak u obradi dizajna jeste da se analiziraju VHDL modeli
- Korektan kod mora zadovoljiti pravilo sintakse i semantike VHDL jezika. Analizator je programski alat koji vrši ovu analizu.
- Ako opis ne ispunjava neko pravilo, analizator obezbeđuje poruku o grešci u kojoj je navedena lokacija greške kao i koje pravilo je prekršeno
- Drugi zadatak koji obavlja analizator jeste prevođenje opisa u internu formu koja je lakša za obradu narednim alatima. Bez obzira da li se ovo prevođenje obavi ili ne, analizator svaki uspešno analiziran opis smešta u *dizajn biblioteku*.
- Celokupan VHDL opis obično se sastoji iz većeg broja deklaracija entiteta i njihovih arhitekturnih tela. Njih zovemo *dizajn jedinice*.
- Organizovanje dizajna kao hijerarhije modula, umesto kao jedan veliki dizajn predstavlja dobru inžinjersku praksu. Na taj način opis se daleko lakše može razumeti i obrađivati.

Analiza dizajna II

- Analizator analizira svaku dizajn jedinicu posebno i smešta internu formu u biblioteku kao bibliotečku jedinicu (library unit)

- Ako jedinica koja se analizira koristi drugu jedinicu, analizator izvlači informaciju o drugoj jedinici iz biblioteke, da bi proverio da li se jedinica ispravno koristi

- Na primer, ako arhitekturno telo instancionira entitet, analizator mora da proveri broj, tip i mod portova entiteta da bi se uverio da se entitet instancionira ispravno

- Da bi ovo mogao da uradi, entitet mora već biti analiziran i smešten u biblioteku

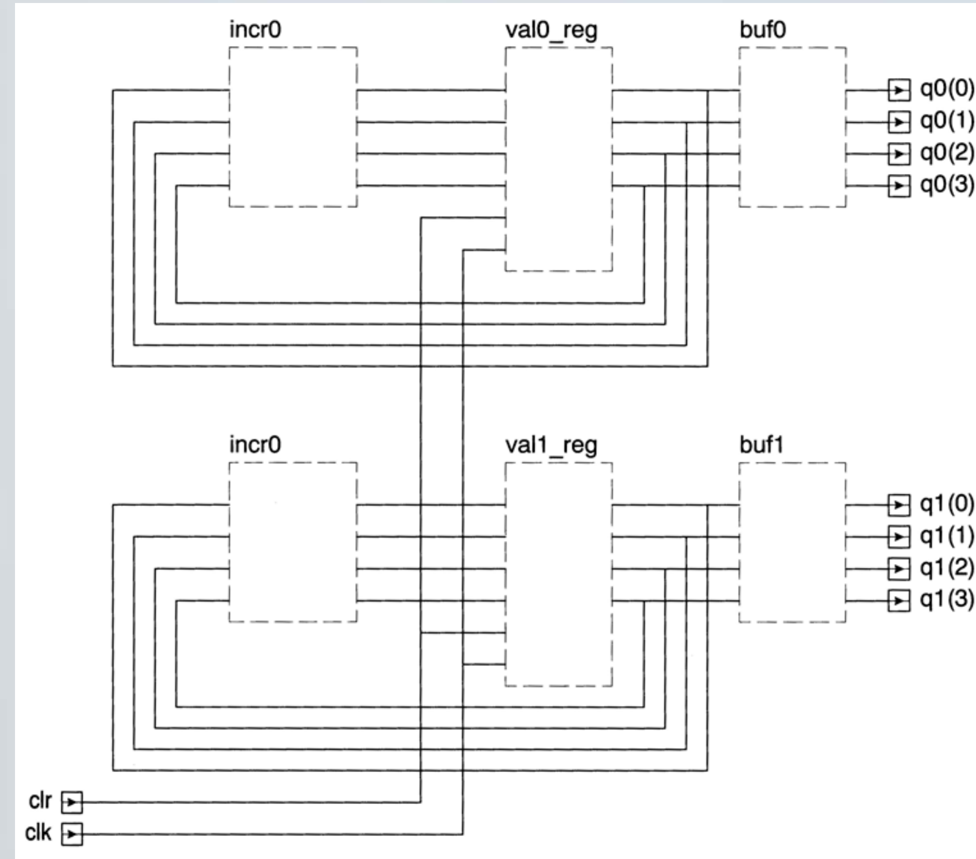
- Tako da možemo videti da postoje zavisnosti između bibliotečkih jedinica u kompletnom opisu koje nameću redosled analiziranja dizajn jedinica

Elaboracija dizajna I

- Kada su sve jedinice iz hijerarhije dizajna analizirane, hijerarhija dizajna može da se razvije
- Efekat razvijanja je da se ukloni hijerarhija, na taj način što se formiraju procesi povezani sa mrežama signala
- Ovo se radi tako što se sadržaj arhitekturnih tela zamenjuje sa svakim instancioniranjem njemu pripadajućeg entiteta
- Svaka mreža u razvijenom dizajnu sastoji se od signala i portova zamenjenih tela arhitektura na koje je signal spojen

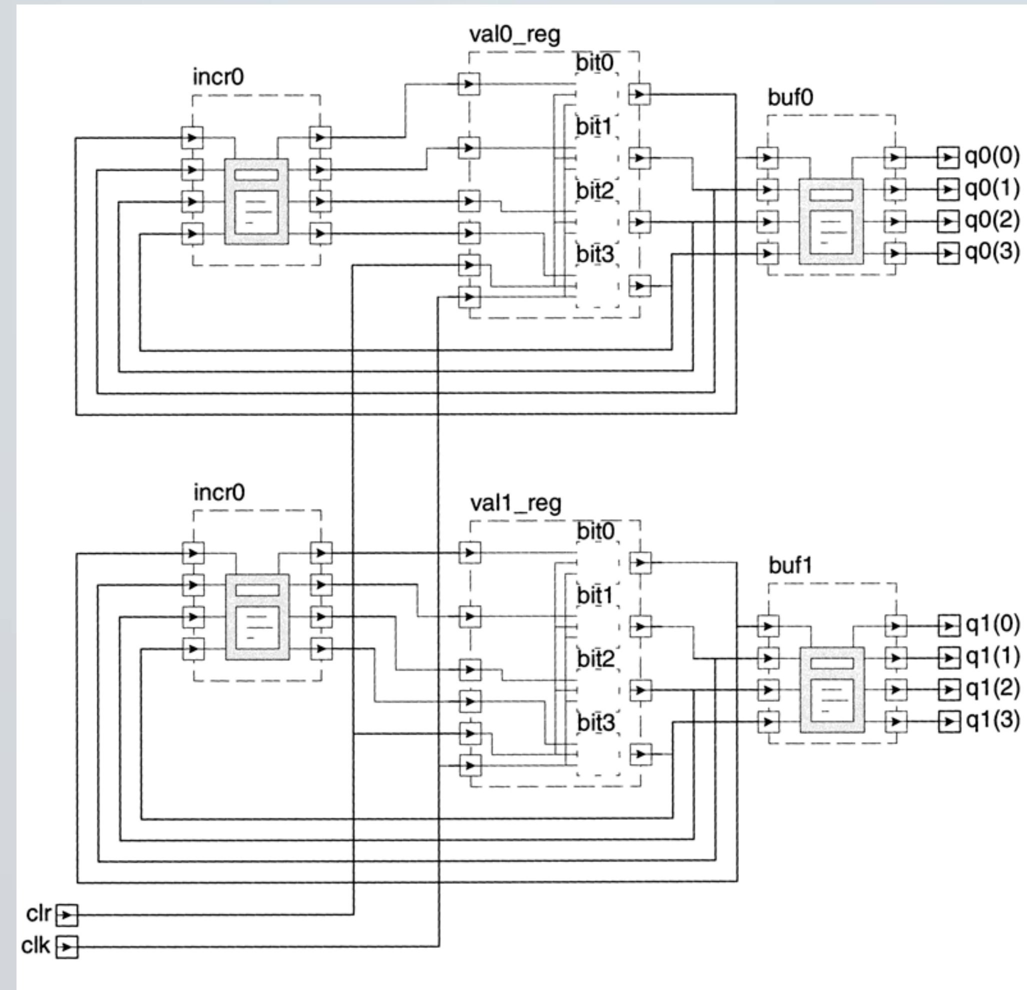
Elaboracija dizajna II

- Elaboracija je rekuzivna operacija, koja počinje sa najvišim entitetom u hijerarhiji dizajna
- Koristićemo *counter* primer sa slajda 10 kao najviši entitet
- Prvi korak je da se kreiraju portovi entiteta.
- Zatim se bira arhitekturno telo koje odgovara entitetu. Ako ne specificiramo eksplicitno koje arhitekturno telo treba koristiti, poslednje arhitekturno telo koje je analizirano biće upotrebljeno.
- Za ovaj primer, koristićemo arhitekturu *registered*.
- Ovo arhitekturno telo se zatim razvija, prvo tako što se kreiraju signali koje ono deklariše, zatim se elaborira svaka konkurentna naredba iz njegovog tela



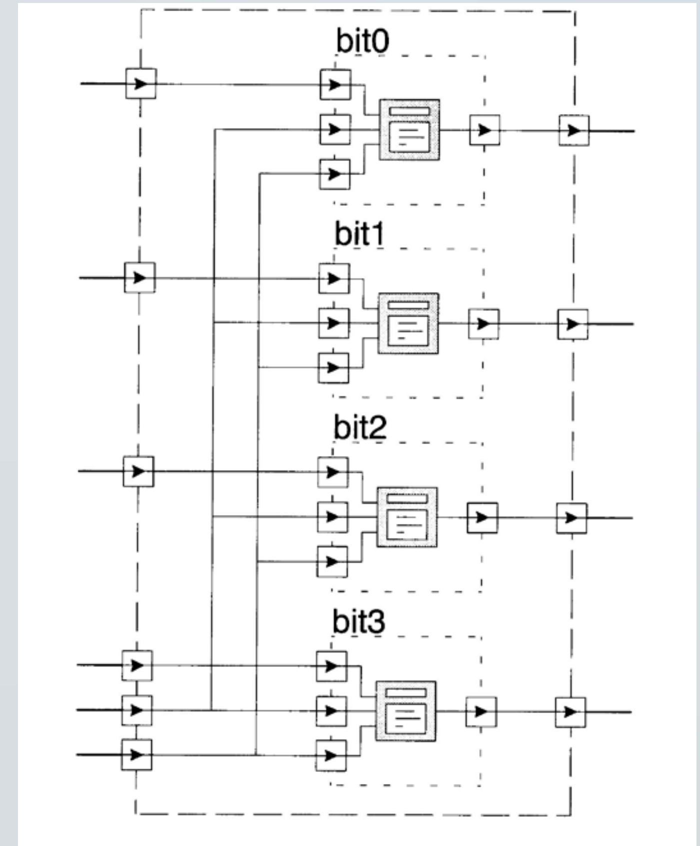
Elaboracija dizajna III

- Konkurentne naredbe u ovom arhitekturnom telu su sve naredbe instancioniranja komponente
- Svaka od njih se zatim razvija na taj način što se kreiraju nove instance portova specificirane instancioniranim entitetom, a zatim se oni spajaju u mreže predstavljene signalima sa kojima su portovi asocirani
- Zatim se unutrašnja struktura arhitekturnog tela instancioniranog entiteta kopira na mesto instance komponente kao što je prikazano na slici desno
- Arhitekture koje zamenjuju instance entiteta *add_1* i *reg4* obe predstavljaju opise načina rada, i sastoje se od procesa koji čitaju sa ulaznih portova i vrše dodele izlaznim portovima. Stoga je elaboracija za ove arhitekture završena.



Elaboracija dizajna IV

- Međutim, arhitektura *struct*, koja menja svaku instancu entiteta *reg4*, sadrži dodatne signale i instance komponenti
- Zbog toga se one elaboriraju i dobija se struktura sa slike desno za svaku od instanci. Sada smo dostigli fazu u kojoj imamo kolekciju mreža koje se sastoje od signala i portova, i procese koji su osetljivi i stimulišu te mreže.
- Svaka *process* naredba u dizajnu elaborira se na taj način što se kreiraju nove instance promenljivih koje proces deklariše i što se kreiraju drajveri za svaki od signala za koje postoje naredbe dodele vrednosti
- Drajveri su spojeni u mreže koje sadrže signale koje oni pobuđuju. Na primer, *storage* proces unutar *bit0* iz *val0_reg* ima drajver za port *q*, koji je deo mreže bazirane na signalu *current_val(0)*.
- Jednom akda su sve instance komponenti i svi rezultujući procesi elaborirani, razvijanje hijerarhije dizajna je završeno
- Sada imamo potpuno razvijenu verziju dizajna, koja se sastoji od instanci procesa i mreža koje ih međusobno povezuju



Izvršavanje I

- Nakon što smo formirali razvijenu verziju hijerarhije dizajna, možemo je izvršiti da bi smo simulirali rad sistema koji smo modelovali
- Simulacioni algoritam sastoji se iz faze inicijalizacije nakon koje sledi faza simulacionog ciklusa koja se ponavlja
- Simulator vodi sat da bi merio proticanje simulacionog vremena
- U fazi inicijalizacije, simulaciono vreme postavlja se na nulu
- Svaki drajver se inicijalizuje tako da signal kojem je on pridružen dobije početnu vrednost koja je deklarisana za taj signal, ili podrazumevanu vrednost ako za signal nije deklarisana početna vrednost.

Izvršavanje II

- Sledeće, svaka od instanci procesa u dizajnu se startuje i izvršavaju se sekvencijalne naredbe iz njenog tela
- Model se obično piše tako da makar neka od ovih inicijalnih naredbi planira transakcije da bi se simulacija pokrenula
- Zatim se procesi zaustavljaju na *wait* naredbama
- Kada su se sve instance procesa zaustavile, inicijalizacija je završena i simulator može početi prvi simulacioni ciklus

Izvršavanje III

- Na početku simulacionog ciklusa, može biti više drajvera sa planiranim transakcijama na njima i veći broj instanci procesa koje imaju planirane timeout-e
- Prvi korak u simulacionom ciklusu je da se simulaciono vreme pomeri na najraniji trenutak u kome postoji planirana transakcija ili time-out procesa
- Drugo, sve transakcije planirane za taj trenutak se izvršavaju, ažuriraju odgovarajuće signale što može dovesti do pojave događaja na nekim od tih signala
- Treće, sve instance procesa koje su osetljive na neki od ovih događaja postaju aktivne

Izvršavanje IV

- Takođe, instance procesa čiji je timeout istekao se nastavljaju
- Svi ti aktivni procesi izvršavaju svoje sekvencijalne naredbe, eventualno planiraju transakcije ili timeout, i na kraju se ponovo zaustavljaju izvršavajući *wait* naredbe
- Kada se svi aktivni procesi zaustave, simulacioni ciklus je završen i sledeći ciklus može da počne
- Ako nema više transakcija ili timeout-ova, ili ako je simulaciono vreme dostiglo *time'high* vrednost (najveće vreme koje tip *time* može da reprezentuje), simulacija je završena