Uvod u mikroračunarsku elektroniku

Predavanje VIII

Sadržaj predavanja

- Deklaracija paketa
- Tela paketa
- USE naredbe
- Generičke konstante
- Parametrizovani modeli

Paketi

- Paketi (packages) u VHDL obezbeđuju važan način organizacije podataka i podprograma deklarisanih u modelu

- U ovom predavanju opisaćemo osnove paketa i pokazati kako se oni mogu koristiti prilikom pisanja VHDL modela

Deklaracija paketa

Deklaracija paketa I

- VHDL paket predstavlja način grupisanja kolekcije povezanih deklaracija koje imaju istu namenu
- To bi mogao biti skup potprograma koji obezbeđuju operacije na određenom tipu podataka, ili prosto može biti skup deklaracija potrebnih za modelovanje odgovarajućeg dizajna
- Važna stvar je da oni mogu biti grupisani zajedno u zasebnu jedinicu dizajna na kojoj se može raditi nezavisno i koja se može iskoristiti na raznim mestima u modelu
- Sledeća bitna stvar u vezi sa paketima je da oni razdvajaju spoljni pogled na objekte koje deklarišu od njihove implementacije
- Spoljni pogled je specificiran u *deklaraciji paketa*, dok je implementacija definisana u posebnom *telu paketa*

Deklaracija paketa II

- Sintaksno pravilo za deklaraciju paketa je

deklaracija_paketa ⇐
package identifikator **is**
{deklarativni_deo}
end [package][identifikator];

- Identifikator obezbeđuje ime za paket, koje koristimo u modelu kada se referišemo na paket
- Unutar deklarativnog dela paketa pišemo kolekciju deklaracija, uključijući deklaracije tipova, podtipova, konstanti, signala i potprograma, kao i ostale vrste deklaracija koje ćemo videti u narednim poglavljima. Ovo su deklaracije koje su obezbeđene za korisnike paketa.
- Prednost njihovog stavljanja u paket je da se na taj način ne opterećuju ostali delovi modela, i da se mogu deliti unutar i između modela bez potrebe da se prepisuju

Deklaracija paketa III

- U nastavku prikazana je jednostavna deklaracija paketa

```
package cpu_types is
    constant word_size: positive := 16;
    constant address_size: positive := 24;
    subtype word is bit_vector(word_size-1 downto 0);
    subtype address is bit_vecotr(address_size-1 downto 0);
    type status_value is (halted, idle, fetch, mem_read,
                                    mem_write, io_read, io_write, int_ack);
end package cpu_types;
```

Deklaracija paketa IV

- Paket predstavlja još jedan oblik *jedinice dizajna*, zajedno sa deklaracijom entiteta i arhitekturnim telima
- On se posebno analizira i stavlja u radnu biblioteku
- Odatle, druge bibliotečke jedinice mogu izvršiti referisanje na objekat deklarisan u paketu koristeći *selektovano ime* objekta
- Selektovano ime formira se pišući ime biblioteke, zatim ime paketa i na kraju ime objekta, međusobno razdvojenih tačkama

Deklaracija paketa V

- Pretpostavimo da je *cpu_package*, prikazan desno, analiziran i smešten u biblioteku *work*
- Možemo iskoristiti neke od deklarisanih objekata prilikom modelovanja adresnog dekodera koji ide uz procesor
- Deklaracija entiteta i ahritekturno telo za dekoder takođe su prikazani desno

```
entity address_decoder is
   port (addr: in work.cpu_types.address;
         status: in work.cpu_types.status_value;
         mem_sel, int_sel, io_sel: out bit);
end entity address_decoder;

architecture functional of address_decoder is
   constant mem_low: work.cpu_types.address := X"000000";
   constant mem_high: work.cpu_types.address := X"EFFFFF";
   constant io_low: work.cpu_types.address := X"F00000";
   constant io_high: work.cpu_types.address := X"FFFFFF";
begin
   mem_decoder:
      mem_sel <=
         '1' when (work.cpu_types."="(status, work.cpu_types.fetch)
                  or work.cpu_types."="(status, work.cpu_types.mem_read)
                  or work.cpu_types."="(status, work.cpu_types.mem_write))
                  and addr >= mem_low and addr <= mem_high
         else '0';
   int_decoder:
      int_sel <= '1' when work.cpu_types."="(status, work.cpu_types.int_ack)
                 else '0';
   io_decoder:
      io_sel <=
         '1' when (work.cpu_types."="(status, work.cpu_types.io_read)
                  or work.cpu_types."="(status, work.cpu_types.io_write))
                  and addr >= io_low and addr <= io_high
         else '0';
end architecture functional;
```

Deklaracija paketa VI

- Videli smo da se paket, nakon što se analizira, smešta u radnu biblioteku
- Objektima iz paketa mogu pristupiti druge bibliotečke jedinice koristeći selektovana imena koja počinju se rečju *work*
- Međutim, ako pišemo paket sa opšte korisnim deklaracijama, možda bi smo želeli da ih smestimo u drugu biblioteku, na primer projektnu biblioteku, gde bi im mogli pristupiti drugi dizajneri
- Različita VHDL radna okruženja obezbeđuju različite načine za specificiranje biblioteke u koju se smešta bibliotečka jedinica. Za detalje je potrebno konsultovati dokumentaciju pojedinog proizvođača.
- Međutim, jednom kada je paket smešten u biblioteku resursa, možemo se referisati na objekte deklarisane u njemu koristeći selektovana imena koja počinju sa imenom biblioteke resursa
- Na primer, možemo razmotriti *IEEE standard_logic* paket, koji mora biti smešten u biblioteku sa imenom *ieee*. Možemo se referisati na tipove deklarisane u tom paketu, na primer:

variable strored_state: ieee.std_logic_1164.std_logic;

Deklaracija paketa VII

- Jedna vrsta deklaracije koju možemo uključiti u paket je deklaracija signala. Ovo nam daje mogućnost definisanja signala, kao na primer signala glavnog takta i reset signala, koji su globalni za čitav dizajn, umesto da budu ograničeni na jedno arhitekturno telo.

- Svaki modul koji se mora referisati na globalni signal jednostavno ga imenuje koristeći selektovano ime kao što je malopre opisano. Ovim se izbegava potreba specificiranja signala kao porta u svakom entitetu koji ga koristi, čineći na taj način model malo prostijim.

- Međutim, ovo znači i da model može uticati na sveukupno ponašanje sistema na druge načine osim onih preko svojih portova, vršeći dodelu globalnim signalima. Ovo znači da je deo interfejsa modula implicitan, za razliku od onog dela koji je specificiran port mapom entiteta.

- Globalni signali deklarisani u paketima trebalo bi da se koriste retko, a način njihovog korišćenja mora biti jasno dokumentovan sa komentarima unutar modela.

Deklaracija paketa VIII

- Paket prikazan desno deklariše dva takt signala koji će biti iskorišćeni prilikom projektovanja integrisanog kola za ulazno/izlazni interfejs kontroler

- Arhitektura kontrolera na najvišem nivou (top-level) okvirno je prikazana na sledećem slajdu

```
library ieee; use ieee.std_logic_1164.all;

package clock_pkg is
    constant Tpw: delay_length := 4 ns;
    signal clock_phase1, clock_phase2 : std_ulogic;
end package clock_pkg;
```

Deklaracija paketa IX

- Instanca entiteta *phase_locked_clock_gen* koristi *ref_clock* port kola da generiše dvofazne talasne oblike globalnih takt signala

- Arhitektura takođe sadrži i instancu entiteta koji upravlja operacijama na magistrali koristeći kontrolne signale magistrale i takođe generiše unutrašnje kontrolne signale za registar

```
library ieee; use ieee.std_logic_1164.all;
entity io_controller is
   port (ref_clock: in std_logic; …);
end entity io_controler;

architecture top_level of io_controller is
   …
begin
   internal_clock_gen: entity
      work.phase_locked_clock_gen(std_cell)
         port map (refernce => ref_clock,
                   phi1 => work.clock_pkg.clock_phase1,
                   phi2 => work.clock_pkg.clock_phase2);

   the_bus_sequencer: entity work.bus_sequencer(fsm)
         port map (rd, wr, sel, width, burst,
                   addr(1 downto 0), ready,
                   control_reg_wr, status_reg_rd,
                   data_fifo_wr, data_fifo_rd, …);
   …
end architecture top_level;
```

Deklaracija paketa X

- Arhitekturno telo kontrolera magistrale okvirno je prikazano desno
- Ono kreira instancu registar entiteta i spaja globalne takt signale na njegove takt portove

```
architecture fsm of bus_sequencer is
-- Ovo arhitektura implementira kontroler kao konačni
-- automat.
-- NAPOMENA: Arhitektura koristi takt signale iz
--                 clock_pkg za sinhronizaciju konačnog
--                 automata.
   signal next_state_vector: …;
begin
   bus_sequencer_state_register: entity
      work.state_register(std_cell)
         port map (phi1 => work.clock_pkg.clock_phase1,
                   phi2 => work.clock_pkg.clock_phase2,
                   next_state => next_state_vector,
                   …);
   …
end architecture fsm;
```

Potprogrami u deklaracijama paketa I

- Druga vrsta deklaracije koja može biti uključena u deklaraciju paketa je deklaracija potprograma, procedure ili funkcije
- Ovo nam omogućava da napišemo podprograme koji implementiraju korisne operacije i pozivamo ih iz većeg broja različitih modula
- Važna upotreba ove mogućnosti je da se deklarišu podprogrami koji operišu sa vrednostima deklarisanog u paketu
- Ovo nam daje način da proširimo VHDL sa novim tipovima i operacijama, takozvanim *apstraktnim tipovima podataka*, tema na koju ćemo se kasnije vratiti

Potprogrami u deklaracijama paketa II

- Važan aspekt deklaracije potprograma u deklaraciji paketa je da jedino pišemo zaglavlje potprograma, odnosno deo koji uključuje ime i interfejs listu koja definiše parametre (i tip rezultata za funkcije). Telo potprograma izostavljamo.

- Razlog za ovo je da deklaracija paketa , kao što smo pomenuli ranije, obezbeđuje spoljašnji pogled na objekte koje deklariše, ostavljajući detalje implementacije objekata za telo paketa

- Za objekte kao što su tipovi i signali, kompletna definicija koja je potrebna sadržana je u spoljašnjem pogledu

- Međutim, za potprograme, potrebna nam je samo informacija sadržana u zaglavlju da bi smo bili u mogućnosti da pozovemo potprogram

- Kao korisnici potprograma, ne moramo brinuti kako on postiže svoj efekat ili računa svoj rezultat. Ovo je primer opšteg principa zvanog *sakrivanje informacije* (information hiding): čineći interfejs vidljivim, ali sakrivajući detalje implementacije.

Potprogrami u deklaracijama paketa III

- Da bi smo ilustrovali ovu ideju, pretpostavimo da imamo deklaraciju paketa koja definiše podtip tipa *bit-vector*:

  subtype word32 **is** bit_vector (31 **downto** 0);

- U paket možemo ukjlučiti proceduru koja sabira vrednosti tipa *word32* koje predstavljaju označene cele brojeve. Deklaracija procedure unutar deklaracije paketa bila bi

  procedure add (a, b: **in** word32; result: **out** word32; overflow: **out** boolean);

- Treba primetiti da ne uključujemo ključnu reč *is* ili neku od lokalnih deklaracija ili naredbi potrebnih za izvođenje sabiranja. Te stvari ostavljene su za telo paketa. Sve što uključujemo jeste opis formalnih parametara procedure.

- Slično, mogli smo uključiti funkciju koja izvodi aritmetičko poređenje dve *word32* vrednosti:

  function "<" (a, b: **in** word32) **return** boolean;

- I ovde izostavljamo lokalne deklaracije i naredbe, specificirajući samo formalne parametre i tip rezultata funkcije

Konstante u deklaracijama paketa I

- Isto kao što možemo primeniti princip skrivanja informacije na potprograme deklarisane u paketu, možemo ga primeniti i na konstante deklarisane u paketu
- Eksterni pogled na konstantu čini njeno ime i tip. Ove stvari moramo poznavati da bi smo koristiti konstantu, ali njenu vrednost ne moramo znati.
- Ovo se na prvi pogled može činiti čudnim, ali ako se prisetimo da je razlog za uvođenje konstanti na prvom mestu bio izbegavanje razbacivanja literalnih vrednosti po celom modelu, vidimo da nepoznavanje vrendosti konstante ima smisla
- Specificiranje vrednosti konstante izbegavamo izostavljanjem inicijalizacionog izraza, na primer:

constant max_buffer_size: positive;

- Ovim smo definisali konstantu kao pozitivnu celobrojnu vrednost. Specifikacija stvarne vrednosti ostavljena je za telo paketa. Uz gornju deklaraciju, prateće telo paketa mora sadržati potpunu deklaraciju konstante, na primer:

constant max_buffer_size: positive = 4096;

Konstante u deklaracijama paketa II

- Možemo proširiti specifikaciju paketa sa slajda 7, koja deklariše korisne tipove za model procesora, uključujući deklaracije koje se odnose na obradu instrukcijskih kodova
- Prerađeni paket prikazan je desno
- On sadrži podtip koji predstavlja vrednost instrukcionog koda, funkciju za izdvajanje instrukcionog koda iz instrukcijske reči i veći broj konstanti koje predstavljaju instrukcione kodove za različite instrukcije

```
package cpu_types is
    constant word_size: positive := 16;
    constant address_size: positive := 24;
    subtype word is bit_vector(word_size –1 downto 0);
    subtype address is bit_vector(address_size – 1 downto 0);
    type status_value is (halted, idle, fetch, mem_read,
                          mem_write, io_read, io_write,
                          int_ack);
    subtype opcode is bit_vector(5 downto 0);

    function extract_opcode (instr_word: word)
                    return opcode;

    constant op_nop: opcode := “000000”;
    constant op_breq: opcode := “000001”;
    constant op_brne: opcode := “000010”;
    constant op_add: opcode :=  “000011”;
    …
end package cpu_types;
```

Konstante u deklaracijama paketa III

- Desno je prikazan bihevijalni model procesora koji koristi ove deklaracije
- Proces interpreter instrukcija deklariše promenljivu tipa *opcode* i koristi funkciju *extract_opcode* da bi izdvojio bitove koji predstavljaju instrukciojski kod iz zahvaćene instrukcijske reči
- On zatim koristi konstane iz paketa kao izbore u *case* naredbi da bi dekodirao i izvršio instrukciju specificiranu instrukcijskim kodom

```
architecture behavioral of cpu is
begin
   interpreter: process is
      variable instr_reg: work.cpu_types.word;
      variable instr_opcode: work.cpu_types.opcode;
   begin
      …  -- inicijalizacija
      loop
         …  --  zahvat instrukcije
         instr_opcode := work.cpu_types.extract_opcode(instr_reg);
         case instr_opcode is
            when work.cpu_types.op_nop => null;
            when work.cpu_types.op_breq => …
            …
         end case;
      end loop;
   end process interpreter;
end architecture behavioral;
```

Tela paketa

Tela paketa I

- Svaka deklaracija paketa koja uključuje deklaraciju potrograma ili deklaraciju konstante mora imati odgovarajuće telo paketa u kojem će se nalaziti preostali detalji

- Međutim, ako deklaracija paketa sadrži samo druge vrste deklaracija, kao što su deklaracije tipova, signala ili potpunoo specificiranih konstanti, nije nam potrebno telo paketa

- Sintaksno pravilo za telo paketa slično je onom za interfejs uz dodatak ključne reči *body*:

telo_paketa ⇐

package body identifikator **is**

{deklarativni_deo}

end [**package**][identifikator];

Tela paketa II

- Objekti deklarisani u telu paketa moraju uključiti pune deklaracije svih podprograma definisanih u odgovarajućoj deklaraciji paketa

- Ove potpune deklaracije moraju sadržati zaglavlja podprograma u istom obliku u kakvom se nalaze u deklaraciji paketa, da bi se obezbedilo da implementacija odgovara interfejsu

- Ovo znači da imena, tipovi, modovi i podrazumevane vrednosti za svaki od formanih parametara moraju biti ponovljene u potpunosti

- Dozvoljena su samo dva odstupanja:
  - Prvo, numerički literal može se zapisati drugačije, na primer, u drugoj bazi, pod uslovom da ima istu vrednost
  - Drugo, prosto ime koje se sastoji samo od identifikatora može biti zamenjeno sa selektovanim imenom, pod uslovom da se odnose na isti objekat

Tela paketa III

- Pored potpunih deklaracija objekata iz deklaracije paketa, telo paketa može sadržati deklaracije dodatnih tipova, podtipova, konstanti i potprograma
- Ovi objekti se koriste za implementaciju podprograma definisanih u deklaraciji paketa
- Treba napomenuti da objekti deklarisani u deklaraciji paketa ne mogu biti deklarisani ponovo u telu paketa (osim podprograma i konstanti), jer su oni automatski vidljivi u telu paketa
- Dalje, telo paketa ne može sadržati deklaracije dodatnih signala
- Deklaracije signala mogu se nalaziti samo u deklaraciji interfejsa paketa

Tela paketa IV

- Desno je prikazan okvirni izgled deklaracije paketa i deklaracije tela paketa u kojima se deklarišu preopterećene verzije aritmetičkih operatora za bit-vektor vrednosti
- Funkcije pretpostavljaju da bit vektori reprezentuju označene cele brojeve u binarnoj formi
- U deklaraciji paketa nalaze se samo zaglavlja funkcija. Telo paketa sadrži kompletna tela funkcija.
- Telo paketa takođe sadrži funkciju, *mult_unsigned*, koja nije definisana u deklaraciji paketa. Ona se koristi interno u telu paketa da bi se implementirao operator označenog množenja.

```
package bit_vector_signed_arithmetic is
   function "+" (bv1, bv2: bit_vector) return bit_vector;
   function "-" (bv: bit_vector) return bit_vector;
   function "*" (bv1, bv2: bit_vector) return bit_vector;
   …
end package bit_vecotr_signed_arithmetic

package body bit_vector_signed_arithmetic is
   function "+" (bv1, bv2: bit_vector) return bit_vector is …
   function "-" (bv: bit_vector) return bit_vector is …
   function mult_unsigned (bv1, bv2: bit_vector) return bit_vector is
   …
   begin
      …
   end function mult_unsigned;
   function "*" (bv1, bv2: bit_vector) return bit_vector is
   begin
      if bv1(bv1'left) = '0' and bv2(bv2'left) = '0' then
         return mult_unsigned(bv1, bv2);
      elsif bv1(bv1'left) = '0' and bv2(bv2'left) = '1' then
         return -mult_unsigned(bv1, -bv2);
      elsif bv1(bv1'left) = '1' and bv2(bv2'left) = '0' then
         return -mult_unsigned(-bv1, bv2);
      else
         return mult_unsigned (-bv1, -bv2);
      end if;
   end function "*";
   …
end package body bit_vector_signed_arithmetic;
```

Tela paketa V

- Poslednja tačka koju treba pomenuti na temu paketa vezana je za redosled analize
- Pomenuli smo ranije da je paket posebna dizajn jedinica koja se analizira posebno od ostalih jedinica, kao što su deklaracije entiteta ili arhitekturna tela
- U stvari, deklaracija paketa i njoj pripadajuće telo paketa su posebne dizajn jedinice pa se stoga moraju analizirati odvojeno
- Deklaracija paketa je primarna dizajn jedinica, a telo paketa je sekundarna dizajn jedinica
- Telo paketa zavisi od informacije sadržane u deklaraciji paketa, pa se deklaracija analizira prva
- Dalje, deklaracija se mora analizirati pre bilo koje dizajn jedinice koja se referiše (koristi) na objekte definisane u paketu
- Jednom kada se deklaracija analizira, nije bitno kada će se analizirati telo paketa u odnosu na jedinice koje koriste paket, pod uslovom da se analizira pre nego što se model elaborira

USE naredbe

USE naredbe I

- Videli smo kako se možemo referisati na objekat obezbeđen od strane paketa, pišući njegovo selektovano ime, na primer, *work.cpu_types.status_value*
- Ovo ime odnosi se na objekat *status_value* koji je definisan u paketu *cpu_types* koji se nalazi u biblioteci *work*
- Ako imamo potrebu da se referišemo na ovaj objekat na mnogim mestima u modelu, pisanje selektovanog imena može biti zamorno i može zamagliti smisao modela
- Ranije smo videli da možemo napisati *use* naredbu da bi smo biblioteku učinili direktno vidljivom u modelu, što nam omogućava da izostavimo ime biblioteke kada se referišemo na nju
- Obzirom da je analizirani paket bibliotečka jedinica, *use* naredbu se takođe može primeniti i na pakete

USE naredbe II

- Tako smo mogli koristiti *use* naredbu u modelu sa Slike 8-1

  use work.cpu_types;

- Ovaj *use* naredba dozvoljava nam da pišemo deklaracije unutar modela na jednostavniji način, na primer:

  variable data_word: cpu_types.word;
  variable next_address: cpu_types.address;

- U stvari, *use* naredba je mnogo generalnija nego što ova upotreba nagoveštava i dozvoljava nam da svako ime iz biblioteke ili paketa učinimo direktno vidljivim Pogledajmo potpuno sintaksno pravilo za *use* naredbu:

  use_naredba ⇐ **use** selektovano_ime {, . . .};
  selektovano_ime ⇐
  ime.(identifikator | karakter_literal | simbol_operatora | **all**)

USE naredbe III

- Ako se ovo primeni u gornjem sintaksnom pravilu, vidimo da selektovano ime može biti oblika

  identifikator.identifikator.(identifikator | karakter_literal | simbol_operatora | **all**)

- Jedna mogućnost je da je prvi identifikator ime biblioteke, a drugi je ime paketa unutar biblioteke

- Ovaj oblik nam omogućava da se referišemo direktno na objekte unutar paketa bez potrebe da koristimo potpuno selektovano ime. Na primer, možemo dodatno uprostiti gornje deklaracije ako *use* naredbu napišemo kao

  use work.cpu_types.word, work.cpu_types.address;

- Tada se deklaracije promenljivih mogu napisati kao

  variable data_word: word;

  variable next_address: address;

USE naredbe IV

- *Use* naredbu možemo smestiti u bilo koji deklarativni deo u modelu
- Jedan način kako možemo zamisliti *use* naredbu, je da ona “uvodi” imena navedenih objekata u deo modela koji sadrži *use* naredbu, tako da se ona mogu koristiti bez pisanja imena biblioteke ili paketa. Imena postaju direktno vidljiva posle *use* naredbe.
- Sintaksno pravilo za *use* naredbu pokazuje da možemo napisati ključnu reč *all* umesto imena nekog objekta
- Ovaj oblik je vrlo koristan, jer predstavlja skraćeni način za uvođenje svih imena definisanih u interfejsu paketa.
- Na primer, kao koristimo *IEEE standard_logic* paket kao osnovu za tipove podataka u dizajnu, često je korisno uvesti sve iz tog paketa. Ovo možemo uraditi sa *use* naredbom na sledeći način:

use ieee.std_logic_1164.**all**;

- Ovo znači da model uvodi sva imena definisana u paketu *std_logic_1164* koji se nalazi u biblioteci *ieee*

USE naredbe V

- Desno je prikazana prerađena verzija arhitekturnog tela prikazanog na slajdu 20
- Ono uključuje *use* naredbu koja se odnosi na objekte deklarisane u paketu *cpu_types*
- Ovim se ostatak modela čini daleko čitljivijim
- *Use* naredba nalazi se u deklarativnom delu procesa za interpreter instrukcija
- Tako su imena “importovana” iz paketa direktno vidljiva u nastavku deklarativnog dela i telu procesa

```
architecture behavioral of cpu is
begin
   interpreter: process is
      use work.cpu_types.all;
      variable instr_reg: word;
      variable instr_opcode: opcode;
   begin
      …  -- inicijalizacija
      loop
         …  --  zahvat instrukcije
         instr_opcode := extract_opcode (instr_reg);
         case instr_opcode is
            when op_nop => null;
            when op_breq => …
            …
         end case;
      end loop;
   end process interpreter;
end architecture behavioral;
```

USE naredbe VI

- Poslednja stvar koju treba razmotriti je način na koji *use* naredbe mogu biti uključene na početku dizajn jedinice, kao i u deklarativnim delovima unutar bibliotečke jedinice

- Ranije smo videli kako možemo uključiti biblioteku i *use* naredbu na početku dizajn jedinice, kao što je interfejs entiteta ili arhitekturno telo

- Ova oblast dizajn jedinice zove se njen kontekstni deo (*context clause*)

- Ovo je najverovatnije najuobičajenije mesto za uključivanje *use* naredbi. Imena uvedena ovde su direktno vidljiva u celoj dizajn jedinici.

- Na primer, ako želimo da koristimo *IEEE standard_logic* tip *std_ulogici* u deklaraciji entiteta, mogli bi smo napisati dizajn jedinicu na sledeći način:

```
library ieee; use ieee.std_logic_1164.std_ulogic;
entity logic_block is
    port (a, b: in std_ulogic; y, z: out std_ulogic);
end entity logic_block;
```

USE naredbe VII

- *Library* naredba i *use* naredba zajedno čine kontekstni deo za deklaraciju entiteta u ovom primeru
- *Library* naredba čini kontekst biblioteke dostupnim modelu, a *use* naredba uvodi ime tipa *std_ulogic* deklarisano u paketu *std_logic_1164* u biblioteci *ieee*
- Uključenjem *use* naredbe u kontekstni deo u deklaraciji entiteta, *std_ulogic* ime tipa je dostupno kada se deklarišu portovi entiteta
- Imena importovana sa *use* naredbom na ovaj način su učinjena direktno vidljivim u celom dizajnu nakon *use* naredbe
- Ako je dizajn jedinica primarna jedinica (kao npr. deklaracija entiteta ili deklaracija paketa), vidljivost je proširena na svaku odgovarajuću sekundarnu jedinicu
- Tako da ako uključimo *use* naredbu u primarnu jedinicu, ne moramo da je ponavljamo u sekundarnoj jedinici, jer su imena automatski vidljiva

Generičke konstante

Generičke konstante I

- Modeli koje smo koristili kao primere u prethodnim poglavljima su svi imali fiksno ponašanje i strukturu

- U mnogim slučajevima, ovo predstavlja ograničenje, pa bi bilo zgodno da smo u mogućnosti da pišemo generalnije modele. VHDL obezbeđuje mehanizam, zvani *generics*, za pisanje parametrizovanih modela

- U ovom delu diskutovaćemo *generičke konstante* i pokazaćemo kako se mogu koristiti za pisanje familija modela sa varirajućim ponašanjem i strukturom

Generičke konstante II

- Možemo pisati generičke entitete uključujući generičku interfejs listu u njihovim deklaracijama koja definiše formalne generičke konstante koje parametrizuju entitet

- Prošireno sintaksno pravilo za deklaraciju entiteta koje uključuje *generic* deo je:

deklaracija_entiteta ⇐
entity identifikator **is**
[**generic** (generic_interfejs_lista);]
[**port**(port_interfejs_lista);]
{deklarativni_deo}
[**begin**
{konkurentna_assertion_naredba
| pasivni_konkurentni_poziv_procedure
| pasivna_proces_naredba}]
end [**entity**][identifikator];

Generičke konstante III

- Razlika između ovog i prostijeg pravila je u postavljanju opcione generičke interfejs liste pre port interfejs liste
- Generička interfejs lista je ista kao i svaka druga interfejs lista, ali uz ograničenje da u nju smemo uključiti samo konstantne objekte, koji moraju biti moda *in*
- Obzirom da se oni podrazumevaju za generičku interfejs listu, možemo koristiti uprošćeno sintaksno pravilo:

generička_interfejs_lista ⇐ (identifikator {, . . .}: indikacija_podtipa [:= izraz]) {,...}

Generičke konstante IV

- Jednostavan primer za deklaraciju entiteta sa generičkom interfejs listom je

```
entity and2 is
        generic (Tpd: time);
        port (a, b: in bit; y: out bit);
end entity and2;
```

- Ovaj entitet sadrži jednu generičku konstantu, *Tpd*, predefinisanog tipa *time*

- Vrednost ove generičke konstante može se koristiti unutar tela entiteta i u bilo kojem arhitekturnom telu koje se odnosi na taj entitet

- U ovom primeru namera je da generička konstanta specificira kašnjenje modula, tako da se njena vrednost treba koristiti u naredbi dodele signalu kao kašnjenje

Generičke konstante V

- Arhitekturno telo koji bi realizovalo gore opisano ponašanje moglo bi izgledati:

```
architecture simple of and2 is
begin
    and2_function:
        y <= a and b after Tpd;
end architecture simple;
```

- Vidljivost generičke konstante prostire se od kraja generičke interfejs liste do kraja deklaracije entiteta i takođe obuhvata svako arhitekturno telo koje se odnosi na deklarisan entitet

Generičke konstante VI

- Generičkoj konstanti se dodeljuje vrednost kada se entitet koristi u naredbi instancioniranja komponente

- Ovo se postiže uključivanjem *generic* mape, kao što se vidi iz proširenog sintaksnog pravila za instancioniranje komponente:

naredba_instancioniranja_komponente ⇐
labela:
entity ime_entiteta **[**(identifikator_arhitekture)**]**
[generic map (generic_lista_asocijacije)**]**
[port map (port_lista_asocijacije)**]**;

Generičke konstante VII

- Generička lista asocijacije je ista kao i druge forme listi asocijacije, ali obzirom da su generičke konstane uvek klase konstanta, stvarni argumenti koje obezbeđujemo moraju biti izrazi. Tako da je pojednostavljeno pravilo za generičku listu asocijacije:

  generička_lista_asocijacije ⇐

  ([generic_ime =>](izraz | **open**)){, . . .}

- Kao ilustraciju, pogledajmo naredbu instancioniranja komponente koja koristi *and2* entitet:

```
gate1: entity work.and2 (simple)
    generic map (Tpd => 2 ns)
    port map (a => sig1, b => sig2, y => sig_out);
```

- Generička mapa specificira da ova instanca *and2* modula koristi vrednost 2 ns za generičku konstantu *Tpd*; odnosno, instanca ima propagaciono kašnjenje od 2 ns

Generičke konstante VIII

- Mogli bi smo uključiti drugu naredbu instancioniranja komponente koja koristi *and2* u istom dizajnu, ali sa drugačijom vrednošću za *Tpd* u generičkoj mapi, na primer:

```
gate2: entity work.and2 (simple)
    generic map (Tpd => 3 ns)
    port map (a => a1, b => b1, y => sig1);
```

- Kada se dizajn elaborira imali bi smo dva procesa, jedan koji bi odgovarao instanci *gate1* entiteta *and2* i koji koristi vrednost 2 ns za *Tpd*, i drugi za instancu *gate2* entiteta *and2*, koji koristi vrednost 3 ns za *Tpd*

Generičke konstante IX

- Kao što sintaksno pravilo za generičku interfejs listu pokazuje, možemo definisati veći broj generičkih konstanti različitih tipova i ukjlučiti podrazumevane vrednosti za njih. Jedan primer prikazan je desno.

- U ovom primeru, generička interfejs lista sadrži dve generičke konstante koje parametrizuju propagaciono kašnjenje modula i Bulovu generičku konstantu, *debug*, sa podrazumevanom vrednošću *false*

- Funkcija ove poslednje generičke konstante je da dozvoli dizajnu koji koristi ovaj entitet da aktivira neke operacije vezane za debagovanje

- Ove operacije mogle bi imati formu *report* naredbi unutar *if* naredbe koja testira vrednost konstante *debug*

```
entity control_unit is
    generic (Tpd_clk_out, Tpw_clk: delay_length;
                debug: boolean := false);
    port (clk: in bit;
            ready: in bit;
            control1, control2: out bit);
end entity control_unit;
```

Generičke konstante X

- Imamo istu slobodu u pisanju generičke mape kao i kod drugih listi asocijacija
- Možemo koristiti pozicionu asocijaciju, imenovanu asocijaciju ili njihovu kombinaciju
- Možemo izostaviti stvarne vrednosti za generičke konstante koje imaju definisanu podrazumevanu vrednost, ili možemo eksplicitno iskoristiti podrazumevanu vrednost pišući ključnu reč *open* u generičkoj mapi
- Kao ilustracija, evo tri različita načina za pisanje generičke mape za *control_unit* entiteta:

```
generic map (200 ps, 1500 ps, false);
generic map (Tpd_clk_out => 200 ps, Tpw_clk => 1500 ps);
generic map (200 ps, 1500 ps, debug => open);
```

Generičke konstante XI

- Desno je prikazana deklaracija entiteta i bihevijalno arhitekturno telo za D-flipflop

- Model sadrži generičke konstante:
  - *Tpd_clk_q* koja specificira propagaciono kašnjenje od rastuće ivice takt signala do pojavljivanja signala na izlazu,
  - *Tsu_d_clk* koja specificira vreme uspostavljanja podataka pre pojave ivice takt signala i
  - *Th_d_clk* koja specificira vreme držanja podataka nakon ivice takt signala.

- Vrednosti ovih generičkih konstanti koriste se unutar arhitekturnog tela

```
entity D_flipflop is
   generic (Tpd_clk_q, Tsu_d_clk, Th_d_clk: delay_length);
   port (clk, d: in bit; q: out bit);
end entity D_flipflop;

architecture basic of D_flipflop is
begin
   behavior: q <= d after Tpd_clk_q when clk = '1' and clk'event;

   check_setup: process is
   begin
      wait until clk = '1';
      assert d'last_event >=  Tsu_d_clk
         report "Narušavanje vremena uspostavljanja!";
   end process check_setup;

   check_hold: process is
   begin
      wait until clk'delayed(Th_d_clk) = '1';
      assert d'delayed'last_event >= Th_d_clk
      report "Narušavanje vremena držanja!";
   end process check_hold;
end architecture basic;
```

Parametrizovani modeli

Parametrizovani modeli I

- Drugi način korišćenja generičkih konstanti u entitetima je za parametrizovanje njihove strukture

- Možemo iskoristiti vrednost generičke konstante da bi smo specificirali veličinu nizovnog porta

- Da bi smo videli zašto je ovo korisno, pogledajmo deklaraciju entiteta za registar. Entitet koji koristi neograničeni nizovni tip za svoj ulazni i izlazni port može se deklarisati kao

```
entity reg is
    port (d: in bit_vector; q: out bit_vector; ...);
end entity reg;
```

- Ovo je potpuno legalna deklaracija entiteta, ali ne uključuje ograničenje da ulazni i izlazni portovi *d i q* moraju biti iste veličine

Parametrizovani modeli II

- Tako da bi smo mogli napisati instancioniranje komponente na sledeći način:

```
signal small_data: bit_vector (0 to 7);
signal large_data: bit_vector (0 to 15);
...
problem_reg: entity work.reg
    port map (d => small_data, q => large_data, ...);
```

- Model će biti analiziran i elaboriran bez detekcije greške
- Greška će biti detektovana tek kada registar pokuša da dodeli mali bit vektor vektoru veće veličine
- Ovaj problem možemo izbeći tako što ćemo uključiti generičku konstantu u deklaraciju entiteta koja će parametrizovati veličinu portova

Parametrizovani modeli III

- Možemo iskoristiti generičku konstantu u ograničenjima prilikom deklaracije portova

- Evo kako bi se to moglo primeniti na prethodnu deklaraciju entiteta:

```
entity reg is
    generic (width: positive);
    port (d: in bit_vector (0 to width-1);
            q: out bit_vector (0 to width-1;
            ...);
end entity reg;
```

- U ovoj deklaraciji zahtevamo da korisnik registra specificira željenu veličinu portova za svaku instancu

Parametrizovani modeli IV

- Entitet zatim koristi vrednost veličine kao ograničenje na ulaznom i izlaznom portu, ne dozvoljavajući da njihova veličina bude određena na osnovu signala asociranih sa portovima

- Instancioniranje komponente koristeći ovaj entitet moglo bi izgledati:

```
signal in_data, out_data: bit_vector (0 to bus_size-1);
...
ok_reg: entity work.reg
    generic map (width =>bus_size);
    port map (d => in_data, q => out_data, ...);
```

- Ako bi signali koji se pridružuju portovima u instancioniranju bili različitih veličina, analizator bi prijavio grešku rano u postupku, olakšavajući otklanjanje greške

Parametrizovani modeli V

- Kompletan model za registar, koji sadrži deklaracije entiteta i arhitekturno telo, prikazan je desno
- Generička konstanta se koristi da ograniči širinu ulaznog i izlaznog porta podataka
- Takođe se koristi unutar ahritekturnog tela da odredi veličinu konstantnog bit vektora *zero*
- Ovaj bit vektor predstavlja vrednost koja se dodeljuje registru kada se on resetuje, tako da mora biti iste veličine kao i izlazni port registra

```
entity reg is
    generic (width: positive);
    port (d: in bit_vector(0 to width – 1);
          q: out bit_vector(0 to width – 1);
          clk, reset: in bit);
end entity reg;

architecture behavioral of reg is
begin
    behavior: process (clk, reset) is
        constant zero: bit_vector(0 to width – 1) := (others => '0');
    begin
        if reset = '1' then
            q <= zero;
        elsif clk'event and clk = '1' then
            q <= d;
        end if;
    end process behavior;
end architecture behavioral;
```

Parametrizovani modeli VI

- Možemo kreirati instance entiteta registra u dizajnu, pri čemu će svaka imati različite veličine portova. Na primer:

```
word_reg: entity work.reg(behavioral)
    generic map (width => 32)
    port map (…);
```

- kreira instancu registra sa 32-bitnim portovima. U istom dizajnu, možemo uključiti drugu instancu:

```
subtype state_vector is bit_vector(1 to 5);
state_reg: entity work.reg(behavioral)
    generic map (width => state_vector'length)
    port map (…);
```

- Ova instanca registra ima 5-bitne portove, dovoljno široke da omoguće smeštaj vrednosti podtipa *state_vector*