Uvod u mikroračunarsku elektroniku

Predavanje VI

Sadržaj predavanja

- Nizovi
- Višedimenzionalni nizovi
- Neograničeni nizovi
- Operacije sa nizovima
- Strukture

Sekvencijalne naredbe

- Kompozitni tipovi podataka sastoje se iz povezane kolekcije elemenata u formi (obliku) bilo nizova ili struktura
- Objekat kompozitnog tipa možemo posmatrati kao celinu ili manipulisati sa njegovim elementima pojedinačno
- U ovom predavanju videćemo kako se definišu kompozitni tipovi i kako se njima manipuliše pomoću operatora i sekvencijalnih naredbi

Nizovi

Nizovi I

- Niz se sastoji iz kolekcije elemenata, pri čemu su svi elementi istog tipa
- Pozicija svakog elementa u nizu određena je skalarnom vrednošću koja se zove *indeks*
- Da bi smo kreirali nizovni objekat u modelu, prvo moramo definisati nizovni tip pomoću deklaracije tipa. Sintaksno pravilo za definisanje nizovnog tipa je

definicija_nizovnog_tipa ⇐ **array** (diskretni_opseg {, . . .}) **of**
indikator_podtipa_elemenata

- Na ovaj način definiše se nizovni tip specificiranjem jednog ili više opsega za indeks (lista diskretnih opsega) i tipa odnosno podtipa kojem pripadaju elementi

Nizovi II

- Diskretni opseg predstavlja podskup vrednosti nekog diskretnog tipa (celobrojni ili nabrojivi tip) i može se specificirati pomoću sledećeg sintaksnog pravila

  diskretni_opseg ⇐ indikator_diskretnog_podtipa | prost_izraz (**to** | **downto**) prost_izraz

- Indikator podtipa može prosto biti ime prethodno deklarisanog tipa i može da uključuje ograničenje opsega da bi ograničio skup vrednosti tog tipa, kao što je prikazano sledećim pravilom

  indikator_podtipa ⇐ oznaka_tipa [**range** prost_izraz (**to** | **downto**) prost_izraz]

Nizovi III

- Za početak počećemo sa jednodimenzionalnim nizovima, kod kojih postoji samo jedan opseg za indeks. Na primer

```
type word is array (0 to 31) of bit;
```

- Svaki element niza *word* je tipa *bit*, a elementi su indeksirani od 0 do 31. Moguća je i sledeća deklaracija

```
type word is array (31 downto 0) of bit;
```

- Razlika je u tome da indeks sada počinje od 31 i završava se sa 0
- Vrednosti indeksa ne moraju biti numeričke. Na primer ako je nabrojivi tip deklarisan kao

```
type controller_state is (initial, idle, active, error);
```

- mogli bi smo deklarisati niz kao

```
type state_counts is array (idle to error) of natural;
```

Nizovi IV

- Ova vrsta deklaracije tipa oslanja se na činjenicu da će tip indeksa biti jasan na osnovu konteksta
- Ako bi postojao više nego jedan nabrojivi tip sa vrednostima *idle* i *error* ne bi bilo moguće razlučiti koji od njih treba koristiti kao tip za indeks
- Da bi se to razjasnilo može se koristiti alternativna forma za specificiranje opsega indeksa u kojoj se neposredno imenuje tip indeksa praćen sa specifikacijom opsega
- Prethodni primer može se dakle zapisati kao

```
type state_counts is array (controller_state range idle to error) of natural;
```

Nizovi V

- Ako želimo da u nizu imamo element za svaku moguću vrednost tipa kojeg je indeks dovoljno je da samo navedemo ime tipa kojeg je indeks bez navođenja opsega. Na primer

```
subtype coeff_ram_address is integer range 0 to 63;
type coeff_array is array (coef_ram_address) of real;
```

- Jednom kada deklarišemo nizovni tip, možemo definisati objekte tog tipa, uključujući konstante, promenljive i signale.

- Na primer, koristeći tipove koje smo malopre deklarisali možemo deklarisati promenljive:

```
variable buffer_register, data_register: word;
variable counters: state_counts;
variable coeff: coeff_array;
```

Nizovi VI

- Svaki od ovih objekata sastoji se iz kolekcije elemenata opisanih odgovarajućom deklaracijom tipa.
- Individualni element može se koristiti u izrazima ili mu se može dodeliti vrednost pomoću naredbe dodele na sledeći način:

```
coeff(0) := 0.0;
```

- Gornja naredba dodele dodeljuje elementu niza *coeff* sa indeksom 0 vrednost 0.0.
- Ako je *active* promenljiva tipa *controller_state*, možemo pisati

```
counters(active) := counters(active)+1;
```

- Nizovni objekat može se koristiti kao jedan složeni objekat. Na primer dodela

```
data_register := buffer_register;
```

- kopira sve elemente niza *buffer_register* u odgovarajuće elemente niza *data_register*.

Nizovi VII

- Desno je prikazan model memorije koja ima kapacitet od 64 realna koeficijenta, inicijalizovana na 0.0
- Tip *coef_ram_address* definisan je malopre
- Arhitekturno telo sadrži proces sa nizovnom promenljivom koja se koristi sa smeštanje koeficijenata
- Kada se proces pokrene, prvo se incijalizuje niz koristeći *for* petlju
- Zatim proces čeka da neki od ulaznih portova promeni svoju vrednost
- Kada je *rd* jednak ‘1’, izlazni port *d_out* uzima vrednost koeficijenta čija je adresa prosleđena pomoću ulaznog porta *addr*
- Kada je *wr* jednak ‘1’, vrednost adrese koristi se da se odredi koji koeficijent je potrebno promeniti

```
entity coeff_ram is
   port (rd, wr: in bit; addr: in coeff_ram_address; d_in: in real;
         d_out: out real);
end entity coeff_ram;

architecture abstract of coeff_ram is
begin
   memory: process is
      type coeff_array is array (coeff_ram_address) of real;
      variable coeff: coeff_array;
   begin
      for index in coeff_ram_address loop
         coef(index) := 0.0;
      end loop;
      loop
         wait on rd, wr, addr, d_in;
         if rd = ‘1’ then
            d_out <= coeff(addr);
         end if;
         if wr = ‘1’ then
            coeff(addr) := d_in;
         end if;
      end loop;
   end process memory;
end architecture abstract;
```

Višedimenzionalni nizovi

Višedimenzionalni nizovi I

- VHDL dozvoljava kreiranje višedimenzionalnih nizova za reprezentaciju matrica ili tabela koje su indeksirane sa više nego jednom vrednošću

- Višedimenzionalni nizovni tip deklariše se tako što se specificira lista opsega indeksa

- Na primer, mogli bi smo uključiti sledeće deklaracije tipova u model konačnog automata:

```
type symbol is ('a', '+', 'd', 'h', digit, cr, error);
type state is range 0 to 6;
type transition_matrix is array (state, symbol) of state;
```

- Opsezi za indekse za svaku dimenziju ne moraju biti istog tipa, niti moraju imati isti smer. Pristup nekom od elemenata višedimenzionalnog niza vrši se navođenjem vrednosti indeksa za taj element.

Višedimenzionalni nizovi II

- U trodimenzionalnoj grafici, tačka u prostoru može se predstaviti koristeći tro-elementni vektor [*x*, *y*, *z*] njenih koordinata

- Transformacije, kao što su skaliranje, rotacija i refleksija, mogu se izvesti množeći vektor sa 3x3 transformacionom matricom da bi smo dobili novi vektor koji predstavlja transformisanu tačku

- Možemo napisati deklaracije tipova za tačke i tranformacione matrice:

```
type point is array (1 to 3) of real;
type matrix is array (1 to 3, 1 to 3) of real;
```

Višedimenzionalni nizovi III

- Možemo iskoristiti ove tipove da definišemo promenljive *p* i *q*, koje će predstavljati tačke i matričnu promenljivu *transform*:

```
variable p, q: point;
variable transform: matrix;
```

- Transformacija se može primeniti na tačku *p* da se dobije transformisana tačka *q* na sledeći način:

```
for i in 1 to 3 loop
    q(i) := 0.0;
    for j in 1 to 3 loop
        q(i) := q(i)+transform(i, j)*p(j);
    end loop;
end loop;
```

Inicijalizacija nizova I

- Videli smo kako se pišu literalne vrednosti za skalarne tipove
- Vrlo često imamo potrebu da koristimo literalne nizovne vrednosti, na primer, da bi smo inicijalizovali promenljivu ili konstantu nizovnog tipa
- Ovo se u VHDL može uraditi pomoću nizovnog aggregate prema sintaksnom pravilu

$$\text{aggregate} \Leftarrow ((\text{[izbori} \Rightarrow\text{] izraz}))\{, \ldots\}$$

Inicijalizacija nizova II

- Pogledajmo prvo jednostavan oblik bez dela sa izborom

- On se jednostavno sastoji od liste elemenata u zagradi, na primer:

```
type point is array (1 to 3) of real;
constant origin: point := (0.0., 0.0, 0.0);
variable view_point: point := (10.0, 20.0, 0.0);
```

- Ova forma koristi pozicionu asocijaciju da bi odredila koja vrednost u listi odgovara kojem elementu u nizu

- Prva vrednost odgovara elementu sa krajnjim levim indeksom, sledeća elementu sa indeksom za jedno mesto u desno, sve do poslednje vrednosti koja odgovara elementu sa krajnjim desnim elementom. Mora postojati odnos jedan prema jedan između vrednosti i elemenata niza.

Inicijalizacija nizova III

- Druga forma koristi imenovanu asocijaciju, kod koje je vrednost indeksa za svaki element eksplicitno napisana pomoću izbora iz sintaksnog pravila
- Izbori mogu biti specificirani na potpuno isti način kao izbori u alternativama *case* naredbe
- Sintaksno pravilo za izbore glasi:

izbori ⇐ **(** prost_izraz | dikretni_opseg | **others)** {| **. . .**}

- Na primer deklaracija i inicijalizacija promenljive može se nalisati kao

variable view_point: point := (1 ⇒ 10.0, 2 ⇒ 20.0, 3 ⇒ 0.0);

Inicijalizacija nizova IV

- Glavna prednost imenovane asocijacije je u tome što nam pruža fleksibilnost u pisanju aggregates za veće nizove
- Da bi smo ovo ilustrovali, vratimo se primeru sa memorijom koeficijenata. Deklaracija tipa bila je

type coeff_array **is array** (coeff_ram_address) **of** real;

- Pretpostavimo da želimo da deklarišemo promenljivu, inicijalizujemo prvih nekoliko elemenata na neke vrednosti različite od nule, a preostale elemente na nulu
- Sledi nekoliko načina na koje je to moguće uraditi:

variable coeff: coeff_array := (0 ⇒ 1.6, 1 ⇒ 2.3, 2 ⇒ 1.6, 3 **to** 63 ⇒ 0.0);

- Ovde smo iskoristili specifikaciju opsega da bi smo većinu elemenata inicijalizovali na vrednost nula

Inicijalizacija nizova V

variable coeff: coeff_array := (0 ⇒ 1.6, 1 ⇒ 2.3, 2 ⇒ 1.6, **others** ⇒ 0.0);

- Ključna reč *others* zamenjuje svaku vrednost indeksa koja prethodno nije navedena. Ako se koristi mora biti zadnji izbor u aggregate.

variable coeff: coeff_array := (0|2 ⇒ 1.6, 1 ⇒ 2.3, **others** ⇒ 0.0);

- Simbol '|' može se koristiti za razdvajanje liste vrednosti indeksa elemenata sa istom vrednošću

- Treba napomenuti da se ne mogu mešati poziciona i imenovana asocijacija u nizovnom aggregate. Na primer, sledeći aggregate je nedozvoljen:

variable coeff: coeff_array := (1.6, 2.3, 2 ⇒ 1.6, **others** ⇒ 0.0); -- GREŠKA!

Inicijalizacija nizova VI

- Drugo mesto na kome možemo koristiti aggregate je mesto objekta kome se dodeljuje vrednost promenljive ili signala
- Sintaksno pravilo za naredbu dodele vrednosti glasi

  naredba_dodele_vrednosti ⇐ [labela:](ime|aggregate) := izraz;

- Ako je cilj aggregate, on mora sadržati ime promenljive na svakoj od pozicija elemenata
- Dalje, izraz na desnoj strani mora dati kompozitnu vrednost istog tipa kao i aggregate
- Svaki element u kompozitnoj vrednosti sa desne strane dodeljuje se odgovarajućoj promenljivoj u ciljnom aggregate
- Isto važi i za naredbu dodele vrednosti signalu

Inicijalizacija nizova VII

- Možemo koristiti dodele ovog tipa da bi smo raspodelili kompozitnu vrednost na odgovarajući broj skalarnih signala
- Na primer, ako imamo promenljivu *flag_reg*, koja je četvoroelementni *bit_vector*, možemo izvršiti sledeću dodelu vrednosti za četiri signala tipa *bit*:

$$(z\_flag, n\_flag, v\_flag, c\_flag) \Leftarrow flag\_reg;$$

- Obzirom da je desna strana *bit_vector*, leva ciljna strana se uzima kao *bit_vector* aggregate
- Krajnji levi element promenljive *flag_reg* dodeljuje se signalu *z_flag*, sledeći element se dodeljuje *n_flag* signalu, itd
- Ovaj oblik višestruke dodele je mnogo kompaktniji od alternative pisanja četiri posebne naredbe dodele vrednosti

Atributi nizova I

- Kao što postoje atributi sklaranih tipova, postoje i atributi nizova; oni pružaju informacije o opsezima indeksa
- Atributi se mogu primeniti na nizovne objekte, kao što su konstante, promenljive i signali, da bi dobili informaciju o tipu tih objekata
- Ako sa A označimo tip niza i objekat, a sa N ceo broj između 1 i broja dimenzija A, VHDL definiše sledeće atribute:

- A'left (N)	Leva granica opsega indeksa dimenzije N od A
- A'right (N)	Desna granica opsega indeksa dimenzije N od A
- A'low (N)	Donja granica opsega indeksa dimenzije N od A
- A'high (N)	Gornja granica opsega indeksa dimenzije N od A
- A'range (N)	Opseg indeksa dimenzije N od A
- A'reverse_range (N)	Obrnuti opseg indeksa dimenzije N od A
- A'length (N)	Dužina opsega indeksa dimenzije N od A
- A'ascending (N)	*true* ako je opseg indeksa dimenzije N od A rastući *false* u suprotnom

Atributi nizova II

- Na primer, ako imamo niz deklarisan kao

type A **is array** (1 **to** 4, 31 **downto** 0) **of** boolean;

- vrednosti nekih atributa bi bile

A'left (1) = 1
A'right (2) = 0
A'range (1) → 1 **to** 4
A'length (1) = 4
A'ascending (1) = true

A'low (1) = 1
A'high (2) = 31
A'reverse_range (2) → 0 **to** 31
A'length (2) = 32
A'ascending (2) = false

- Za sve atribute, ako želimo da dobijemo neku informaciju o prvoj dimenziji (ili ako postoji samo jedna dimenzija) možemo izostaviti broj dimenzije u zagradama, na primer:

A'low = 1

A'length = 4

Atributi nizova III

- Atributi se često koriste prilikom pisanja petlji koje treba da prođu kroz sve elemente niza

- Na primer, ako imamo promenljivu *free_map* koja je niz bitova, možemo napisati *for* petlju koja će brojati broj elemenata sa vrednošću '1', bez prethodnog znanja o stvarnoj veličini niza (broju elemenata):

```
count := 0;
for index in free_map'range loop
   if free_map (index) = '1' then
      count := count+1;
   end if;
end loop;
```

Neograničeni nizovi

Neograničeni nizovi I

- Tipovi nizova sa kojima smo se do sada upoznali zaovu se ograničeni nizovi jer je u definiciji tipa ograničen opseg vrednosti koje može uzeti indeks

- VHDL dozvoljava definisanje neograničenih tipova nizova, kod kojih navodimo samo tip vrednosti indeksa, bez navođenja granica

- Sintaksno pravilo za definisanje ove vrste nizova ima oblik

  definicija_neograničenog_nizovnog_tipa ⇐
  array (**(**oznaka_tipa **range** <>**)**{, . . .}) **of** indikator_podtipa_elemenata

- Simbol '<>', može se zamisliti kao mesto gde bi trebalo da se nalazi opseg indeksa, a koje će biti popunjeno kasnije kada se tip bude koristio

Neograničeni nizovi II

- Primer deklaracije neograničenog nizovnog tipa je

type sample **is array** (natural **range** <>) **of** integer;

- Bitna stvar u vezi sa neograničenim tipovima nizova je da kada deklarišemo objekat tog tipa, moramo da obezbedimo granice opsega indeksa

- Ovo se može uraditi na više načina

- Jedan je da obezbedimo ograničenje kada kreiramo objekat, na primer:

variable short_sample_buf: sample (0 **to** 63);

Neograničeni nizovi III

- U gornjem primeru vrednosti indeksa promenljive *short_sample_buf* su brojevi tipa *natural* u rastućem poretku od 0 od 63

- Drugi način za specifikaciju opsega je da deklarišemo podtip ta neograničeni tip niza. Objekti se onda mogu kreirati koristeći taj podtip, na primer:

```
subtype long_sample is sample (0 to 255);
variable new_sample_buf, old_sample_buf: long_sample;
```

- Ovo je primer novog oblika deklaracije podtipa. Sintaksno pravilo je

indikator_podtipa ⇐ oznaka_tipa [(diskretni_opseg {, . . .})]

- Oznaka tipa je ime neograničenog tipa niza, a diskrenti opseg ograničava indeks na odgovarajući željeni opseg

Neograničeni nizovi IV

- Kada deklarišemo konstantu za neki tip neograničenog niza, moguć je treći način na koji možemo ograničiti vrednosti indeksa
- Do potrebnog opsega možemo doći na osnovu izraza za inicijalizaciju konstane
- Ako je izraz za inicijalizaciju nizovni aggregate napisan pomoću imenovane asocijacije, vrednosti indeksa u aggregate određuju opseg indeksa konstantnog niza. Na primer

  constant looup_table: sample := (1 ⇒ 23, 3 ⇒ -16, 2 ⇒ 100, 4 ⇒ 11);

- opseg indeksa je 1 do 4.
- Ako je izraz zapisan pomoću pozicione asocijacije, vrednost indeksa prvog elementa jeste krajnja leva vrednost za podtip kome pripada indeks. Na primer

  constant beep_sample: sample := (127, 63, 0, -63, -127, -63, 0, 63);

- opseg indeksa je 0 do 7, jer indeks pripada podtipu *natural*. Smer indeksa je rastući, jer je podtip *natural* definisan sa rastućim opsegom.

VHDL predefinisani tip *string*

- VHDL obezbeđuje predefinisan neograničeni nizovni tip koji se zove *string*, deklarisan kao

```
type string is array (positive range <>) of character;
```

- U principu opseg indeksa za ograničeni string može biti bilo rastući ili opadajući pri čemu su granice bilo koji pozitivni celi brojevi

- Međutim najčešće se koristi rastući opseg sa početnom vrednošću indeksa jednakom 1. Na primer:

```
constant LCD_display_len: positive := 20;
subtype LCD_display_string is string (1 to LCD_display_len);
variable LCD_display: LCD_display_string := (others => ' ');
```

VHDL predefinisani tip *bit_vector*

- VHDL takođe obezbeđuje predefinisani nizovni tip *bit_vector*, deklarisan kao

  type bit_vector **is array** (natural **range** <>) **of** bit;

- Ovaj tip se koristi za predstavljanje reči podataka na arhitekturalnom nivou modelovanja

- Na primer, podtip za predstavljanje bajtova podataka mogao bi biti deklarisan kao

  subtype byte **is** bit_vector (7 **downto** 0);

- Alternativno, opseg indeksa možemo definisati i prilikom deklarisanja objekta, na primer:

  variable chanell_busy_register: bit_vector (1 **to** 4);

VHDL predefinisani tip *std_logic_vector*

- Standard_logic paket *std_logic_1164* obezbeđuje neograničeni nizovni tip za vektore sa vrednostima standard_logic

- On je deklarisan kao

  type std_ulogic_vector **is array** (natural **range** <>) **of** std_ulogic;

- Ovaj tip može se koristiti na isti način kao i sličan tip *bit_vector*, sa tom razlikom da pruža više detalja u predstavljanju električnih signala u dizajnu. Možemo definisati podtipove na osnovu standard_logic tipa, na primer:

  subtype std_ulogic **is** std_ulogic_vector (0 **to** 31);

- Ili možemo direktno kreirati objekat tipa standard_ulogic:

  signal csr_offset: std_ulogic_vector (2 **downto** 1);

String i bit-string literali I

- Videli smo da se string literal može iskoristiti za zapisivanje vrednosti koja predstavlja sekvencu karaktera

- String literale možemo koristiti i na mestu nizovnog aggregate za vrednost tipa *string*

- Na primer, možemo inicijalizovati konstantan string na sledeći način:

```
constant ready_message: string := "Ready";
```

- Stringove literala možemo koristiti umesto bilo kojeg jednodimenzionalnog nizovnog tipa čiji su elementi nabrojivog tipa koji uključuje karaktere

- Na primer,

```
variable current_test: std_ulogic_vector (0 to 13) := "ZZZZZZZZZZ----";
```

String i bit-string literali II

- Bit stringovi mogu se koristiti na mestu nizovnih aggregates za zapisivanje vrednosti tipa *bit-vector*

- Na primer, promenljiva *channel_busy_register* definisana malopre može biti inicijalizovana na sledeći način:

```
channel_busy_register := b"0000";
```

- Bit-string literale možemo koristiti umesto bilo kojeg jednodimenzionalnog nizovnog tipa čiji su elementi nabrojivog tipa koji uključuje karaktere '0' i '1,

- Svaki karakter u bit-string literalu predstavljaju jedan, tri ili četiri uzastopna elementa niza, u zavisnosti od toga da li je osnova specificirana unutar literala binarna, oktalna ili heksadecimalna. Na primer

```
constant all_ones: std_ulogic_vector (15 downto 0) := x"FFFF";
```

Neograničeni nizovni portovi I

- Važna primena neograničenog nizovnog tipa jeste u specifikaciji tipa nizovnog porta
- Ovo nam omogućava da interfejs entiteta napišemo u uopštenom obliku, tako da kasnije možemo pripojiti nizovne signale bilo koje veličine ili sa bilo kojim opsegom vrednosti indeksa
- Kada se entitet instancionira, opseg indeksa nizovnog signala spojenog na port se koristi za određivanje opsega porta

Neograničeni nizovni portovi

- Pretpostavimo da želimo da modelujemo familiju I kapija, pri čemu svaka od njih ima različit broj ulaza
- Deklaracija entiteta izgledala bi kao ona prikazana desno
- Ulazni port je neograničenog tipa *bit-vector*
- Arhitekturno telo uljučuje proces koji je osetljiv na promene ulaznog porta
- Kada bilo koji od elemenata promeni svoju vrednost, proces realizuje logičku I operaciju na čitavom ulaznom nizu
- Proces koristi *'range* atribut da bi odredio opseg indeksa za ulazni niz, obzirom da on nije poznat sve do trenutka kada je entitet instancioniran

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

architecture behavioral of and_multiple is
begin
    and_reducer: process (i) is
        variable result: bit;
    begin
        result := '1';
        for index in i'range loop
            result := result and i(index);
        end loop;
        y <= result;
    end process and_reducer;
end architecture behavioral;
```

Operacije sa nizovima

Operacije sa nizovima I

- Iako niz predstavlja kolekciju vrednosti, najčešće operacije nad nizovima izvodimo na pojedinačnim elementima, koristeći operacije opisane u predavanju 2
- Međutim, ako radimo sa jednodimenzionalnim nizovima skalarnih vrednosti, možemo koristiti neke od operatora za izvođenje operacija nad čitavim nizovima
- Prvo, logički operatori (*and*, *or*, *nand*, *nor*, *xor* i *xnor*) mogu se primeniti na dva jednodimenzionalna niza sa elementima tipa *bit* i *boolean*
- Operandi moraju biti iste dužine i istog tipa, a rezultat se izračunava primenjujući operator na parove elementat iz dva niza i dobija se niz iste dužine kao i polazni nizovi
- Elementi su upareni počevši od krajnje leve pozicije u svakom nizu. Element na odgovarajućoj poziciji iz jednog niza je uparen sa elemetom na toj istoj poziciji iz drugog niza.

Operacije sa nizovima II

- Operator *not* takođe se može primeniti na jedan niz sa elementima tipa *bit* ili *boolean*, a kao rezultat dobija se niz iste dužine i istog tipa kao i operand

- Naredne deklaracije i naredbe ilustruju primenu logičkih operatora na *bit* vektore:

```
subtype pixel_row is bit_vector (0 to 15);
variable current_row, mask: pixel_row;

current_row := current_row and not mask;
current_row := current_row xor X"FFFF";
```

Operacije sa nizovima III

- Drugo, operatori pomeranja (*sll*, *srl*, *sla*, *sra*, *ral* i *rar*) mogu se primeniti na jednodimenzionalne nizove tipa *bit* ili *boolean* koji predstavljaju levi operand i celobrojnu vrednost kao desni operand

- Operacija logičkog pomeranja ulevo pomera elemente u nizu za N mesta ulevo (N predstavlja desni operand), popunjavajući prazna mesta sa '0' ili *false* i zanemarujući N krajnjih levih elemenata

- Ako je N negativan broj, elementi se pomeraju udesno. Nekoliko primera za ilustraciju:

  B"10001010" *sll* 3 = B"01010000" B"10001010" *sll* –2 = B"00100010"

- Operacija logičkog pomeranja u desno slično pomera elemente za N pozicija u desno ako je N pozitivan broj, ili u levo ako je N negativan. Na primer:

  B"1001011" *srl* 2 = B"00100101" B"10010111" *srl* –6 = B"11000000"

Operacije sa nizovima IV

- Sledeće dve operacije, aritmetičko pomeranje u levo i u desno, rade slično, ali umesto popunjavanja praznih pozicija sa '0' ili *false*, popunjavaju ih sa kopijom elementa sa kraja koji se napušta, na primer:

  B"01001011" *sra* 3 = B"00001001" B"10010111"*sra* 3 = B"11110010"

  B"00001100" *sla* 2 = B"00110000" B"00010001" *sla* 2 = B"01000111"

- Ako je N negativno, pomeranje se vrši u suprotnom smeru, na primer:

  B"00010001" *sra* –2 = B"01000111"

  B"00110000" *sla* –2 = B"00001100"

- Operator rotacije u levo pomera elemente u nizu N mesta u levo, prebacujući N elemenata sa levog kraja niza u prazna mesta koja se pojavljuju na desnom kraju. Operacija rotacije u desno radi istu stvar, samo u suprotnom smeru. Kao i kod operacija pomeranja negativna vrednost desnog operanda obrće smer rotacije. Neki primeri su:

  B"10010011" *rol* 1 = B"00100111" B"1001011" *ror* 1 = B"11001001"

Operacije sa nizovima V

- Relacioni operatori predstavljaju treću grupu operatora koji se mogu primeniti na jednodimenzionalne nizove
- Elementi niza mogu biti bilo kog diskretnog tipa. Dva operanda ne moraju biti iste dužine, sve dok su im elementi istog tipa.
- Način na koji ovi operatori rade najlakše se može shvatiti kada se primene na stringove karaktera, jer u tom slučaju porede nizove prema case-sensitive dictionary ordering

Operacije sa nizovima VI

- Da bi smo videli kako se dictionary comparison može generalizovati na jednodimenzionalne nizove sa elementima drugih tipova, razmotrimo operator "<" primenjen na dva niza A i B
- Ako A i B imaju dužinu 0, onda je A<B netačno
- Ako A ima dužinu 0, a B neku ne nultu dužinu onda je A<B tačno
- Ako A i B imaju nenulte dužine onda je A<B ako je A(1) <B(1) ili A(1) = B(1), a ostatak A je < od ostatka B
- Ako A ima nenultu dužinu, a B dužinu 0, A<B je netačno
- Poređenje korišćenjem drugih relacionih operatora izvodi se analogno

Operacije sa nizovima VII

- Preostali operator koji se može primeniti na jednodimenzione nizove je operator spajanja (&), koji spaja dva niza
- Na primer, kada se primeni na *bit* vektore, proizvodi novi *bit* vektor sa dužinom koja je jednaka zbiru dužina dva operanda
- Tako je B"0000"&B"1111" jednako B"0000_1111"
- Operator spajanja može se primeniti na dva operanda, od kojih je jedan niz, a drugi jedna skalarni element
- Takođe se može primeniti na dve skalarne vrednosti pri čemu će rezultat biti dužine 2. Evo nekoliko primera:

"abc"&'d' = "abcd"

'w'&"xyz" = "wxyz"

'a'&'b' = "ab"

Rad sa podnizovima I

- Često želimo da radimo sa nekim podskupom elemenata niza, a ne sa celim nizom

- Ovo možemo uraditi koristeći notaciju, u kojoj navodimo levu i desnu vrednost indeksa za deo niza koji želimo koristiti

- Na primer, ako imamo nizove *a1* i *a2* deklarisane kao:

```
type array1 is array (1 to 100) of integer;
type array2 is array (100 downto 1) of integer;

variable a1: array1;
variable a2: array2;
```

Rad sa podnizovima II

- Možemo raditi sa delom niza *a1*(11 **to** 20), koji je niz od 10 elemenata sa vrednostima indeksa od 11 do 20
- Slično, deo niza *a2*(50 **downto** 41) je niz od 10 elemenata ali sa opadajućim opsegom indeksa
- Treba obratiti pažnju da su *a1*(10 **to** 1) i *a2*(1 **downto** 10) nizovi dužine 0, jer su opsezi indeksa jednaki nuli
- Čak štaviše, opsezi specificirani u delu niza moraju imati isti smer kao i u originalnom nizu
- Tako da nije dozvoljeno pisati *a1*(10 **downto** 1) ili *a2*(1 **to** 10)

Rad sa podnizovima III

- Desn je prikazan bihevijalni model mešača bajtova (byte-swapper) koji ima jedan ulazni i jedan izlazni port, oba su tipa *halfword*, koji je deklarisan kao

  subtype halfword **is** bit_vector (0 **to** 15);

- Proces u arhitekturnom telu zamenjuje mesta bajtovima ulaznog porta

- Ovde se može videti kako se delovi nizova mogu koristiti u naredbama dodele vrednosti

```
entity byte_swap is
    port (input: in halfword;
          output: out halfword);
end entity byte_swap;

architecture behavior of byte_swap is
begin
    swap: process (input)
    begin
        output(8 to 15) <= input(0 to 7);
        output(0 to 7) <= input(8 ot 15);
    end process swap;
end architecture behavior;
```

Konverzije nizova I

- U predavanju 2 uveli smo ideju konverzije tipova numeričke vrednosti u neku vrednost *srodnog* tipa
- Vrednost nizovnog tipa takođe se može konvertovati u vrednost nekog drugog nizovnog tipa, isti broj dimenzija i tipove indeksa koji se mogu konvertovati jedan u drugi
- Konverzija tipa prosto formira novi niz željenog tipa, pri čemu je svaki indeks konvertovan u odgovarajući željeni tip
- Da bi smo ilustrovali ideju konverzije tipova nizovnih vrednosti, pretpostavimo da imamo sledeće deklaracije u modelu:

```
subtype name is string (1 to 20);
type display_string is array (integer range 0 to 19) of character;

variable item_name: name;
variable display: display_string;
```

Konverzije nizova II

- Ne možemo direktno dodeliti vrednost tipa *item_name* tipu *display*, obzirom da su tipovi različiti

- Međutim možemo iskoristiti konverziju tipa:

```
display := display_string (item_name);
```

- Ovim se formira novi niz, sa levim elementom sa indeksom 0 i desnim elementom sa indeksom 19, koji je kompatibilan sa ciljnim nizom

- Čest slučaj kod kojeg nema potrebe za konverzijom tipova je dodela vrednosti niza jednog podtipa nizu drugog podtipa, pri čemu su oba podtipa podtipovi istog baznog tipa

- Ova situacija se javlja kada opsezi indeksa ciljnog niza i operanda imaju različite granice ili smerove. VHDL automatski uključuje implicitnu konverziju podtipova prilikom dodele.

Konverzije nizova III

- Na primer ako imamo:

```
subtype big_endian_upper_halfword is bit_vector (0 to 15);
subtype little_endian_upper_halfword is bit_vector (31 downto 16);

variable big: big_endian_upper_halfword;
variable little: little_endian_upper_halfword;
```

- možemo izvršiti sledeće dodele bez potrebe za eksplicitnom konverzijom tipova:

```
big := little;
little := big;
```

Strukture

Strukture I

- Struktura predstavlja kompozitnu vrednost sastavljenu od elemenata koji mogu biti različitih tipova
- Svaki element identifikuje se preko svog imena, koje je jedinstveno unutar strukture
- Ovo ime se koristi za odabir elemenata unutar strukture
- Sintaksno pravilo za definiciju tipa glasi:

definicija_strukturnog_tipa ⇐
record
(identifikator {, . . .}: indikator_podatipa;**)**
{. . .}
end record [identifikator];

Strukture II

- Svako od imena u listama identifikatora deklariše element odgovarajućeg tipa ili podtipa
- Vitičaste zagrade u sintaksnom pravilu znače da obuhvaćeni deo može da se ponavlja neograničen broj puta
- Identifikator na kraju definicije strukturnog tipa, ako se koristi, mora da bude isti kao ime strukture koja se definiše

Strukture III

- Primer za deklaraciju strukturnog tipa i deklaraciju promenljive tog tipa mogao bi biti:

```
type time_stamp is record
        seconds: integer range 0 to 59;
        minutes: integer range 0 to 59;
        hours: integer range 0 to 23;
end record time_stamp;

variable sample_time, current_time: time_stamp;
```

- Vrednost čitave strukture može se dodeliti odjednom koristeći naredbu dodele na primer:

```
sample_time = current_time;
```

Strukture IV

- Takođe možemo pristupiti individualnim elementima u strukturi na sledeći način:

```
sample_hour := sample_time.hours;
```

- U izrazu na desnoj strani naredbe dodele, prefiks pre tačke imenuje promenljivu, a sufiks posle tačke određuje element iz strukture kome pristupamo

- Naravno, moguća je i sledeća situacija:

```
current_time.seconds := clock mod 60;
```

- kada dodeljujemo vrednost individualnim elementima unutar strukture.

Inicijalizacija struktura I

- Strukturni aggregates možemo iskoristiti za zapis literalne vrednosti strukturnog tipa – na primer, za inicijalizaciju strukturne promenljive ili konstante
- Korišćenje strukturnog aggregate je analogno korišćenju nizovnog aggregate za pisanje literalne vrednosti nizovnog tipa
- Strukturni aggregate se formira pisanjem liste elemenata u zagradama
- Aggregate koji koristi pozicionu asocijaciju navodi elemente u istom poratku u kojem se nalaze u deklaraciji strukturnog tipa
- Na primer, ako imamo strukturni tip *time_stamp* od malopre, možemo inicijalizovati konstantu tog tipa na sledeći način:

```
constant midday: time_stamp := (0, 0, 12);
```

Inicijalizacija struktura II

- Možemo koristiti i imenovanu asocijaciju, kod koje identifikujemo svaki element u aggregate pomoću njegovog imena

- Redosled elemenata u ovom slučaju nije bitan. Gornji primer mogao bi biti zapisan kao:

  constant midday: time_stamp := (hours ⇒ 12, minutes ⇒ 0, seconds ⇒ 0);

- Za razliku od nizovnih aggregate, možemo kombinovati imenovanu i pozicionu asocijaciju, pod uslovom da svi imenovani elementi slede iza pozicionih elemenata

- Takođe, možemo koristiti simbol "|" i ključnu reč *others* prilikom pisanja izbora

Inicijalizacija struktura III

- Evo još nekoliko primera koji koriste tipove *instruction* i *time_stamp* deklarisane ranije:

```
constant nop_instr: instruction :=
        (opcode ⇒ addu,
        source_reg1|source_reg2|dest_reg ⇒ 0,
        displacement ⇒ 0);

variable latest_event: time_stamp := (others ⇒ 0);
```

- Za razliku od nizovnih aggregates, ne možemo koristiti opseg vrednosti da bi smo identifikovali elemente u strukturnom aggregate, obzirom da se elementi identifikuju svojim imeno, a ne preko indeksa