PROGRAMIRANJE KOMUNIKACIONOG HARDVERA
– Poglavlje 2 –

2 VHDL

VHDL (*VHSIC Hardware Description Language*) programski jezik spada u grupu HDL (*Hardware Description Language*) jezika koji se, kao što im i samo ime kaže, koriste za opisivanje hardverske implementacije. Drugi poznati HDL jezici su Verilog, AHDL, ABEL, MyHDL, JHDL i dr. Ideja koja leži iza HDL jezika je standardizacija opisa hardverskih implementacija, što omogućava lakši i brži razvoj hardvera, lakše i brže testiranje i verifikaciju hardverske implementacije, kao i portabilnost realizovane implementacije na platforme različitih proizvođača. VHDL jezik je primarno namenjen za programiranje programabilnih CPLD i FPGA čipova, kao i za projektovanje ASIC čipova. Glavna konkurencija mu je Verilog jezik koji je takođe veoma popularan. Ovde je zgodno napomenuti da se većina HDL jezika odnosi na opis digitalnih električnih kola, međutim, postoje i jezici koji podržavaju i analogne komponente pored digitalnih komponenti (VHDL i Verilog imaju svoje ekstenzije VHDL-AMS i Verilog-AMS koje podržavaju i analogne komponente), a takođe postoje i HDL jezici koji se koriste za opis veza na štampanim pločama.

Veoma je važno odmah na početku razjasniti da VHDL nije programski jezik u klasičnom smislu kao što su to npr. C, C++, Java i drugi slični programski jezici. Pod programskim jezikom se klasično podrazumeva pisanje koda na višem ili nižem nivou apstrakcije koji se potom kompajliranjem prevodi u asemblerski kod tj. niz komandi koje izvršava procesor. To u suštini znači da se svaki napisani kod izvršava na istom hardveru tj. procesoru. VHDL jezik s druge strane vrši opisivanje hardverske implementacije koja treba da se realizuje i da izvršava određeni skup logičkih (kombinacionih i/ili sekvencijalnih) funkcija. U zavisnosti od toga koji skup logičkih funkcija treba da se izvršava razlikovaće se i implementirani hardver. Kao što ćemo videti, VHDL jezik tekstualno opisuje osnovne delove hardverske implementacije i veze između njih. Svi delovi opisanog hardvera istovremeno postoje i ukoliko je u pitanju npr. FPGA čip ti delovi se razmeštaju po CLB blokovima FPGA čipa. Postoji dosta sličnosti VHDL jezika sa klasičnim programskim jezicima (slične konstrukcije, ključne reči, i sl.) što često početnike zavara da je VHDL isto što i klasičan programski jezik, ali ne treba smetnuti sa uma da VHDL radi 'samo' opisivanje hardvera i samim tim treba programirati tako da se postigne što efikasnija hardverska realizacija. Kod klasičnih programa su tipični kriterijumi performansi broj instrukcija potreban za izvršenje neke funkcije ili količina memorije koja se zauzima, a kod VHDL-a su tipični kriterijumi minimalno zauzimanje fizičkih resursa, minimalno kašnjenje, maksimalna frekvencija koju realizacija podržava. U oba slučaja navedeni kriterijumi mogu biti oprečni, pa se u takvim slučajevima često radi i balansiranje ili se jedan od kriterijuma odabere kao primarni kriterijum.

2.1 Istorijat VHDL-a

Tokom sedamdesetih godina prošlog veka Ministarstvo odbrane SAD se suočilo sa ozbiljnim problemima i troškovima. Ministarstvo odbrane je nabavljalo velike količine električne opreme koja je sadržala integrisana kola od različitih dobavljača/kompanija. Svaka kompanija je imala poseban alat za simulaciju i testiranje opreme, složena uputstva za razumevanje i održavanje opreme. To je značajno otežavalo obuku ljudi koji bi rukovali i održavali takvu raznovrsnu opremu. Dodatni problem je bio veoma brz razvoj tehnologije što je dovodilo do

brzog zastarevanja opreme, pa je trebalo brzo usvajati nove tehnologije i alate vezane uz njih iz kompleksnih uputstava koja su se pri tome razlikovala od kompanije do kompanije. Još jedan problem se javljao u slučaju složenih sistema sastavljenih od komponenti različitih proizvođača, jer su komponente mogle pojedinačno da se testiraju uz podršku kompanija koje su ih proizvodile, ali za čitav sistem nije bilo podrške već se moralo zasebno razvijati okruženje za testiranje, što je bio kompleksan i skup proces. Svi navedeni problemi su dovodili do ogromnih troškova, pa se počela razvijati ideja da se navedeni troškovi smanje tako što bi se uvela standardizacija opreme, tako što bi sva integrisana kola bila opisivana na identičan način, što bi omogućavalo i standardizaciju njihovog ispitivanja i simulacije, a naravno i znatno bržu obuku osoblja zaduženog za korišćenje i održavanje takve opreme.

Ministarstvo odbrane SAD je iniciralo razvoj VHDL jezika 1981. godine sa idejom da sva oprema koju nabavljaju mora da bude opisana VHDL jezikom što bi značajno olakšalo usvajanje takve opreme kao i testiranje i simulaciju takve opreme. U ranoj fazi je tražena podrška industrije i dobijene povratne informacije su korišćene u razvoju VHDL jezika (s obzirom da je Ministarstvo odbrane SAD veliki potrošač ova podrška je bila i dobijena, veliko je pitanje da li bi se dobila podrška industrije da je neko drugi pokrenuo ovu inicijativu). Usled dobre podrške industrije i uopšte uključivanja industrije u ranoj fazi procesa standardizacije (što nije tada bio tipičan slučaj u procesima standardizacije), VHDL jezik je dobio široku podršku i njegova standardizacija je bila veoma uspešna što se vidi po tome da se ovaj jezik i danas koristi i predstavlja najpopularnije rešenje za opis hardverskog dizajna.

Razvoj standardizacije VHDL-a je 1986. godine sa Ministarstva odbrane SAD prebačeno na IEEE organizaciju radi obezbeđivanja još šire podrške od industrije i korisnika. U 1987. je donesen prvi standard koji je ubrzo dopunjen podrškom za signale sa više vrednosti. Naime, prvobitno je standardom predviđena podrška za signale koji imaju samo dve moguće vrednosti 0 i 1, ali se ubrzo videlo da to nije dovoljno, pa je dodata podrška za signale sa više vrednosti (npr. vrednost visoke impedanse) o čemu će biti više reči u narednim sekcijama. Ovaj standard je poznat i kao VHDL-87 standard. U 1993. je izvršena revizija VHDL standarda i ova verzija standarda je trenutno najkorišćenija (poznata pod nazivom VHDL-93). U 2000. i 2002. su izvršene nove revizije VHDL standarda, koje su unele veoma male dopune (ova verzija standarda je poznata pod nazivom VHDL-2002). U 2006. je započeta nova revizija, koja je rezultirala trenutno poslednjom revizijom standarda (poznatom pod nazivom VHDL-2008) koja je donela veći broj dopuna i izmena u odnosu na reviziju VHDL-2002. Nova revizija je zvanično objavljena 2009. godine, ali tek su najnovije verzije razvojnih okruženja poznatih proizvođača programabilnih čipova počele da uvode podršku za ovu poslednju reviziju.

2.2 Osnovna struktura VHDL dizajna

Već je rečeno da je namena VHDL jezika opis hardverske implementacije. Jedan VHDL kod odgovara opisu dizajna jednog hardverskog bloka (modula) koji izvršava određenu logičku funkciju (ili skup logičkih funkcija). Kada se opisuje jedan hardverski modul važne su dve stvari. Prvu stvar predstavljaju interfejsi modula prema okolini i preko kojih se dotični modul povezuje (komunicira) sa drugim modulima ili čipovima. Ovaj deo je bitan za upotrebu modula i njegovu interakciju sa ostalim hardverom u složenim sistemima, jer je korisniku ili projektantu složenog sistema bitna samo funkcija modula, a ne i interna struktura modula tj. kako se zaista izvršava ta funkcija. Za njih je modul poput ’crne kutije’ - nije bitno šta je u ’crnoj kutiji’, već samo šta ona

radi. Druga stvar je interna struktura modula koja definiše princip rada modula, a samim tim i funkciju (ili skup funkcija) koje modul obavlja. Ovaj deo definiše kako se izvršava funkcija koju modul treba da obavi i ona je važna dizajnerima modula, jer je cilj realizovati što optimalniju implementaciju interne strukture modula da bi modul što efikasnije obavljao svoju funkciju. Optimalna implementacija se može odnositi na više aspekata, tipično to su minimalni upotrebljeni hardverski resursi, minimalno kašnjenje modula, maksimalna brzina rada modula i dr.

Struktura VHDL koda jednog modula je organizovana imajući u vidu da su za opis modula važne dve stvari - interfejsi modula i interna struktura modula. Otuda VHDL kod jednog modula ima tri osnovna dela (napomena: VHDL jezik nije *case sensitive* tj. ne razlikuje velika i mala slova, međutim, u okviru ovih skripti će biti usvojena konvencija da se sve ključne reči pišu velikim slovima):

- biblioteka (LIBRARY)
- entitet (ENTITY)
- arhitektura (ARCHITECTURE)

2.2.1. Biblioteka

Uloga biblioteke je slična ulozi *header* fajlova u C programskom jeziku. U C-u su postojali sistemski *header* fajlovi (npr. stdio.h, stdlib.h, math.h...) koji su predstavljali standardne biblioteke koje je korisnik mogao uključiti u svoj kod i koristiti funkcije koje su one nudile. S druge strane su postojali i korisnički definisani *header* fajlovi koje je definisao korisnik za svoje potrebe, a koji su tipično sadržavali delove koda, funkcije, i deklaracije tipova promenljivih koje je korisnik često koristio (ovakve *header* fajlove i dalje mogu koristiti i drugi korisnici za svoje projekte, ako dotični *header* fajlovi sadrže delove koji im odgovaraju). Slično tome, i u VHDL-u postoje sistemske i korisničke biblioteke sa istom svrhom kao prethodno opisani *header* fajlovi u C-u. Najpoznatije sistemske biblioteke u VHDL-u su *std* i *ieee* biblioteka.

Biblioteka u VHDL-u predstavlja skup tzv. paketa (PACKAGE), pri čemu jedan paket sadrži funkcije (FUNCTION), procedure (PROCEDURE), komponente (COMPONENT), definicije tipova promenjivih (TYPE), konstante (CONSTANT) koje se mogu koristiti u VHDL kodu ako se pozove dotična biblioteka. Samo pozivanje biblioteke u VHDL kodu se izvršava na sledeći način:

```
LIBRARY ime_biblioteke;
USE ime_biblioteke.ime_paketa.deo_paketa;
```

Treba uočiti da se kao kod mnogih programskih jezika deklaracija i izraz moraju završiti sa ; čime se signalizira kompajleru da je u pitanju kraj deklaracije ili izraza. Ključna reč LIBRARY vrši pozivanje određene biblioteke, a sa USE se specificira koji deo biblioteke zaista treba uključiti u VHDL kod. U *ime_biblioteke* se stavlja naziv biblioteke koja se želi uključiti u VHDL kod, npr. *ieee*. Ime paketa iz biblioteke koji se želi koristiti se stavlja u *ime_paketa*. Deo paketa navedenog u *ime_paketa* se specificira u *deo_paketa* i samo taj deo paketa će biti uključen u VHDL kod. Ako se za *deo_paketa* stavi *all*, tada se uključuju svi delovi paketa navedenog u *ime_paketa*, npr:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

Tri biblioteke se (gotovo) uvek koriste u svakom VHDL kodu, i to su *ieee*, *std* i *work* biblioteke. Biblioteke *std* i *work* su po difoltu uvek uključene u VHDL kod tj. ne moraju da se eksplicitno pozivaju. Biblioteka *std* je standardna (sistemska) biblioteka koja sadrži osnovne definicije tipova promenjivih, kao i osnovne funkcije koje su definisane odmah na početku standardizacije VHDL-a (VHDL-87). Biblioteka *work* je korisnički definisana biblioteka i ona u stvari predstavlja sve delove projekta u korišćenom razvojnom okruženju (VHDL fajl/fajlove i sve druge fajlove koji se uključe kao deo projekta) i samim tim ni ova biblioteka ne mora eksplicitno da se poziva u VHDL kodu. Ali u slučaju da korisnik definiše pakete u okviru projekta, mora se koristiti njihovo pozivanje upotrebom USE komande unutar VHDL kodova gde se žele iskoristiti ti paketi. Bibloteka *ieee* predstavlja dopunu prvobitnom standardu VHDL jezika (VHDL-87) koja pre svega uključuje podršku signalima sa više od dve vrednosti (pored vrednosti 0 i 1, na raspolaganju su i druge vrednosti poput stanja visoke impedanse). Jedino ova biblioteka se mora eksplicitno pozivati unutar VHDL koda. Detaljnije o tipovima promenjivih će biti u sekciji 2.3, pa će tamo biti i navedeno u kojim bibliotekama su pojedini tipovi definisani.

2.2.2. Entitet

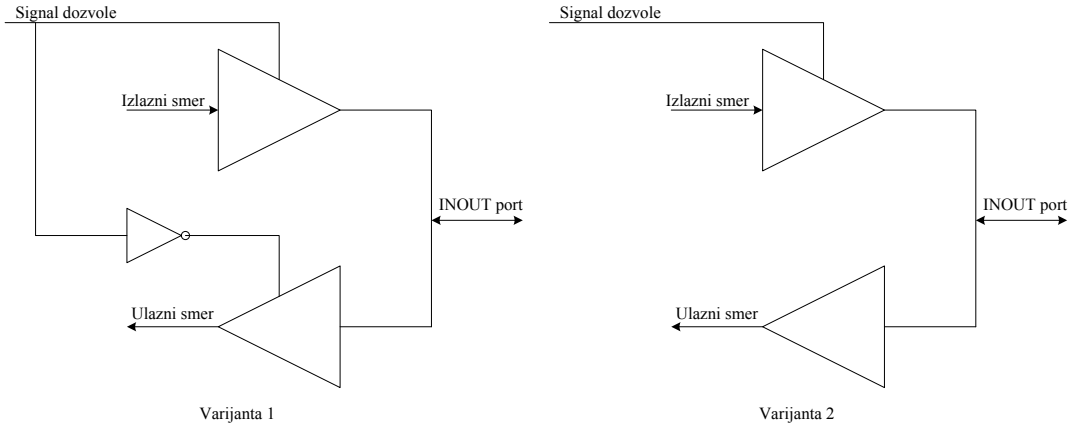
Entitet definiše interfejse hardverskog modula kojeg opisuje VHDL kod. Pored toga entitet može da definiše i parametre hardverskog modula čime se postiže fleksibilnost implementacije i simulacije dizajna. Entitet se definiše na sledeći način:

```
ENTITY ime_entiteta IS
GENERIC
(
    ime_parametra1 : tip := vrednost_parametra;
    ime_parametra2: tip := vrednost_parametra;
    ....
    ime_parametraM : tip := vrednost_parametra
);
PORT
(
    ime_porta1 : mod  tip;
    ime_porta2: mod  tip;
    ....
    ime_portaN : mod  tip
);
END ime_entiteta;
```

Entitet se deklariše ključnom reči ENTITY, zatim sledim naziv entiteta koji se definiše u *ime_entiteta* i potom sledi ključna reč IS. Naziv entiteta je proizvoljan, važno je da počinje slovom i da se ne koristi ključna reč za naziv entiteta (slično ograničenje kao kod većine programskih jezika koje se koristi za nazive npr. promenjivih). Dodatna ograničenja su da nema razmaka u nazivu, da u nazivu nema dva uzastopna *underscore* simbola, da naziv se ne završava sa *underscore* simbolom i da naziv ne sadrži zabranjene simbole poput simbola '?'. Naravno, poželjno je da naziv entiteta bude smislen i da ukaže na funkciju koju obavlja deklarisani entitet. GENERIC deo služi za definisanje parametara entiteta, a PORT deo služi za definisanje interfejsa (tzv. portova u VHDL terminologiji).

U okviru GENERIC dela seređaju parametri pri čemu se definiše naziv parametra iza koga posle dvotačke sledi tip parametra (npr. INTEGER) iza koga posle := sledi difolt vrednost parametra. Deklaracije parametara se odvajaju međusobno sa ; pri čemu je važno uočiti da iza

poslednjeg parametra se ne sme staviti ; čime se označava kraj navođenja parametara. GENERIC deo je opcion, odnosno ne mora da postoji u deklaraciji entiteta.

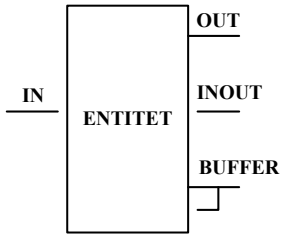


Slika 2.2.2.1. – Upotreba trostatičkih bafera u slučaju INOUT porta

U okviru PORT dela se redaju definicije interfejsa, odnosno portova, tako što se navodi ime porta, potom iza dvotačke se definišu mod porta i tip porta. Mod porta definiše smer porta i postoje četiri moda: IN, OUT, INOUT, BUFFER. IN mod je ulazni mod, OUT mod je izlazni mod. INOUT je ulazno/izlazni mod, odnosno port može biti i ulazni i izlazni, a u toku rada entiteta koji smer je aktivan se određuje dinamički, stoga je neophodno kod ovakvih portova koristiti trostatičke bafere, a takođe se mora definisati kontroler smera. Kontroler smera može biti deo interne strukture entiteta i tada je entitet taj koji diktira smer komunikacije preko INOUT porta, a takođe kontroler smera može biti suprotna strana, odnosno kontroler smera može biti van entiteta i tada je entitet '*slave*' u komunikaciji tj. suprotna strana diktira smer komunikacije. Slika 2.2.2.1 definiše metode za upotrebu trostatičkih bafera kod INOUT porta. Varijanta 1 je 'čistija' varijanta. U ovom slučaju kada signal dozvole aktivira gornji trostatički bafer aktivira se izlazni smer i entitet šalje svoje podatke preko INOUT porta. Istovremeno se deaktivira donji trostatički bafer i ulazni smer ulazi u stanje visoke impedanse. Kada signal dozvole deaktivira gornji trostatički bafer, istovremeno se aktivira donji trostatički bafer čime se aktivira ulazni smer i entitet prima podatke preko INOUT porta. U varijanti 2 razlika je u donjem baferu koji nije trostatički što znači da je donji bafer uvek aktivan. U slučaju aktivacije gornjeg trostatičkog bafera tj. izlaznog smera, tada izlaz donjeg bafer postaje jednak podacima koji se šalju preko INOUT porta (praktično se dobija svojevrsna povratna sprega). U slučaju deaktivacije gornjeg trostatičkog bafera, aktivira se ulazni smer i primaju se spoljašnji podaci preko INOUT porta kao i u varijanti 1. Druga varijanta troši manje resursa jer nema invertora i jedan trostatički bafer je zamenjen prostijim običnim baferom. U suštini, u varijanti 2 se čak ni ne mora koristiti donji bafer, međutim, tipično INOUT port je istovremeno i pin programabilnog čipa, a svi signali koji idu van čipa ili se primaju iz spoljašnje okoline prolaze kroz bafer (ulazni ili izlazni u zavisnosti od smera signala) pa se obično koristi bafer u varijanti 2. Važna napomena je da se uglavnom INOUT portovi ne koriste za komunikaciju između entiteta na istom programabilnom čipu, već se oni koriste samo u komunikaciji entiteta na programabilnom čipu sa nekim drugim (eksternim) čipom, tipično eksternom memorijom. Iz tog razloga je važno odmah uraditi razdvajanje na ulazni i izlazni smer u entitetu sa INOUT portom kao što je prikazano na slici 2.2.2.1. Signal dozvole može biti kontrolisan od strane entiteta i tada je entitet '*master*' u komunikaciji preko INOUT porta i entitet je taj koji određuje smer INOUT porta. U suprotnom,

ako je signal dozvole kontrolisan spolja onda je entitet '*slave*' u komunikaciji preko INOUT porta, i tada entitet ne kontroliše smer INOUT porta.

BUFFER mod je sličan OUT modu u smislu da je to izlazni mod, ali sa dodatkom da se vrednost ovog porta može očitati unutar entiteta. Kod OUT moda, vrednost OUT porta se ne može očitati unutar entiteta, tako da je potrebna upotreba dodatne '*dummy*' promenjive koja će se voditi na OUT port kao izlaz iz entiteta, a '*dummy*' promenjiva će predstavljati vrednost OUT porta koja se sada može očitati unutar entiteta (u nastavku skripti će biti dat primer koji pokazuje ovu situaciju). U praksi BUFFER mod se uglavnom ne koristi često, već se uglavnom koristi OUT mod. U najnovijoj verziji standarda VHDL-2008 mod OUT je takođe dobio mogućnost da se njegova vrednost može očitati unutar entiteta čime je mod BUFFER izgubio na značaju. Generalni prikaz modova je dat na slici 2.2.2.2.



Slika 2.2.2.2. – Modovi portova entiteta

Tip porta definiše tip signala koji se razmenjuje preko porta, to na primer može biti BIT tip gde je u pitanju port koji praktično predstavlja interfejs od jedne žice preko koje se razmenjuju vrednosti 0 i 1. Više detalja o tipovima u sekciji 2.3. Slično kao u GENERIC delu, u PORT delu se deklaracije portova odvajaju sa ; pri čemu se iza poslednjeg porta ne stavlja ; čime se označava kraj deklarisanja portova entiteta. U slučaju da više portova ima identičan mod i tip, oni se mogu navesti u jednom redu na sledeći način:

ime_porta1, ime_porta2, .. ime_portaN: mod tip;

Deklaracija entiteta se završava ključnom reči END koju sledi naziv entiteta i ; kojom se označava kraj deklaracije. Primer deklaracije entiteta koje obavlja funkciju dvoulaznog I kola je:

```
ENTITY dvoulazno_I_kolo IS
        PORT
        (
            x1, x2 : IN BIT;
            y: OUT BIT
        );
END dvoulazno_I_kolo;
```

Kao što vidimo imamo dva ulaza *x1* i *x2*, i jedan izlaz *y*. Svi signali su tipa BIT, jer kod dvoulaznog I kola vrednosti signala treba da uzmu vrednost 0 ili 1. U slučaju da, na primer, želimo da definišemo entitet koji će obavljati '*bit-wise*' I operaciju nad vektorima bita možemo deklarisati takav entitet na sledeći način:

```
ENTITY bitwise_I_kolo IS
        GENERIC
        (
          duz_vektora: INTEGER:=8
        );
        PORT
        (
           x1, x2 : IN BIT_VECTOR(duz_vektora - 1 DOWNTO 0);
           y: OUT BIT_VECTOR(duz_vektora - 1 DOWNTO 0)
```

```
    );
END bitwise_I_kolo;
```

Kao što vidimo, ovde koristimo GENERIC deo da bismo deklarisali fleksibilan entitet za obavljanje *'bit-wise'* I operacije, tako što u GENERIC delu definišemo parametar *duz_vektora* kojim podešavamo dužine ulaznih vektora nad kojima će se vršiti *'bit-wise'* I operacija kao i dužinu rezultujućeg vektora.

Rezime modova portova je dat u tabeli 2.2.2.1.

Tabela 2.2.2.1. – Modovi portova entiteta

Mod	Objašnjenje
IN	Ulazni port
OUT	Izlazni port
BUFFER	Izlazni port sa mogućnošću čitanja vrednosti signala ovog porta u unutrašnjosti entiteta (arhitekturi)
INOUT	Ulazno/izlazni port, neophodna upotreba trostatičkih bafera u unutrašnjosti entiteta (arhitekturi)

2.2.3. Arhitektura

Arhitektura definiše internu strukturu modula kojeg opisuje VHDL kod. Struktura arhitekture je sledeća:

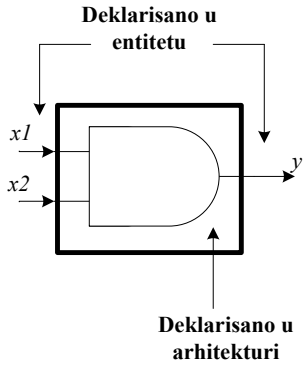
```
ARCHITECTURE ime_arhitekture OF ime_entiteta IS
    deklaracije
BEGIN
    kod
END ime_arhitekture;
```

Deklaracija arhitekture počinje ključnom reči ARCHITECTURE koju sledi naziv arhitekture za kojom sledi ključna reč OF i potom ime entiteta čiju internu strukturu opisuje dotična arhitektura i potom ključna reč IS. Naziv arhitekture treba da poštuje ista pravila kao naziv entiteta, pri čemu je dozvoljeno poklapanje naziva entiteta i arhitekture. Za razliku od naziva entiteta gde je poželjno da ono opiše svrhu entiteta, kod arhitekture to uglavnom nije važno jer to ime je vidljivo samo entitetu pa pošto je naziv entiteta već opisao svrhu onda naziv arhitekture ne mora to da radi. Na primer, autor skripti tipično stavlja naziv *shema* kao naziv arhitekture. Pošto jedan entitet može da ima više arhitekturi, jedino je u tim slučajevima poželjno da naziv arhitekture bude smislen i da ukaže na razlike među definisanim arhitekturama. Međutim, u praksi je veoma retko definisanje više arhitektura za jedan isti entitet, pa se taj slučaj neće obrađivati u okviru ovih skripti.

Nakon definisanja naziva arhitekture i entiteta na koji se ona odnosi, sledi deo za deklaracije u okviru koga se definišu interni signali koji postoje u internoj arhitekturi, ali nisu vidljivi van entiteta čiju internu strukturu opisuje dotična arhitektura. Pored toga ovde se mogu deklarisati funkcije, konstante, drugi entiteti u vidu komponenti koje se uključuju u internu strukturu itd. Više detalja će biti dato u kasnijem delu skripti. Ključna reč BEGIN označava kraj dela za deklaracije i početak dela koji zaista opisuje internu strukturu entiteta (označen sa *kod*). Struktura *koda* će takođe biti objašnjena u nastavku skripti. Sama deklaracija arhitekture se završava ključnom reči END koju sledi naziv dotične arhitekture i ;.

Primer arhitekture za dvoulazno I kolo čiji je entitet deklarisan u sekciji 2.2.2.

```
ARCHITECTURE shema OF dvoulazno_I_kolo IS
BEGIN
        y<=x1 AND x2;
END shema;
```

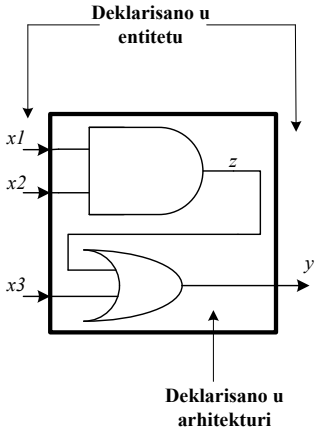


Slika 2.2.3.1. – Dvoulazno I kolo

Kao što vidimo ovde nema dela za deklaracije jer je interna struktura veoma jednostavna. Praktično identična arhitektura važi i za entitet *bitwise_I_kolo* takođe definisanom u sekciji 2.2.2, jedino što bi se morao promeniti naziv entiteta u definiciji arhitekture. Slika 2.2.3.1. prikazuje opisano dvoulazno I kolo i prikazuje šta se definiše u delu koji deklariše entitet, a šta u delu koji deklariše arhitektura. Jasno se vidi sa slike 2.2.3.1 da se u entitetu definišu interfejsi realizovanog modula, a u arhitekturi se definiše interna struktura realizovanog modula.

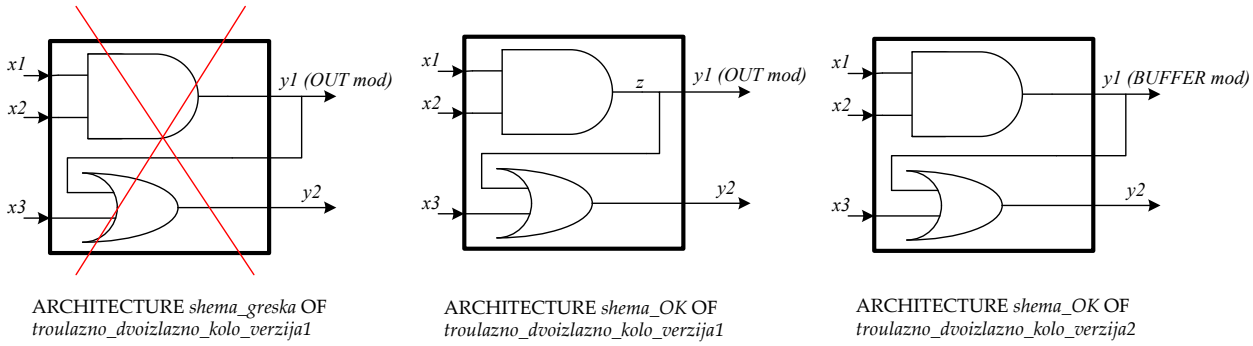
Ako uzmemo malo složenije kolo prikazano na slici 2.2.3.2, njegova deklaracija entiteta, odnosno arhitekture bi bili sledeći:

```
ENTITY troulazno_kolo  IS
PORT
(
        x1, x2, x3 : IN  BIT;
        y: OUT  BIT
);
END troulazno_kolo;
ARCHITECTURE shema OF troulazno_kolo IS
        SIGNAL z : BIT;
BEGIN
        z<= x1 AND x2;
        y<=z OR x3;
END shema;
```



Slika 2.2.3.2. – Troulazno kolo

Ovde kao što vidimo imamo interni signal koji se mora deklarisati u delu za deklaracije arhitekture. Ovaj signal kao što se vidi sa slike 2.2.3.2 nije vidljiv izvan entiteta.



ARCHITECTURE *shema_greska* OF *troulazno_dvoizlazno_kolo_verzija1*

ARCHITECTURE *shema_OK* OF *troulazno_dvoizlazno_kolo_verzija1*

ARCHITECTURE *shema_OK* OF *troulazno_dvoizlazno_kolo_verzija2*

Slika 2.2.3.3. – Primer upotrebe '*dummy*' signala i rada BUFFER moda

Prikažimo i primer koji ukazuje na razliku u korišćenju BUFFER i OUT moda. Cilj je realizovati modul sa tri ulaza i dva izlaza, pri čemu se žele koristiti samo dvoulazna kola u internoj strukturi modula. Izlaz *y1* treba da bude jednak rezultatu logičke I operacije nad ulazima *x1* i *x2*. Izlaz *y2* treba da bude jednak rezultatu logičke ILI operacije nad *y1* i ulazom *x3*. Definicija entiteta *troulazno_dvoizlazno_kolo_verzija1* koristi za oba izlaza OUT mod. Date su dve arhitekture ovog entiteta, jedna neispravna (*shema_greska*) i jedna ispravna (*shema_OK*). Greška u neispravnoj arhitekturi potiče od činjenice da se za formiranje izlaza *y2* koristi vrednost izlaza *y1*, a pošto je izlaz *y1* definisan kao OUT port to nije dozvoljeno, jer se vrednost OUT porta ne može očitati unutar interne strukture (arhitekture) modula. Da bi se prevazišao ovaj problem deklariše se i koristi interni signal *z* koji ima ulogu '*dummy*' signala u ispravnoj arhitekturi *shema_OK*. Ovo je ilustrovano na slici 2.2.3.3. Drugo rešenje je da se u entitetu za izlaz *y1* postavi mod BUFFER, čime bi arhitektura *shema_greska* postala ispravna, jer je sada omogućeno da se čita vrednost *y1* unutar arhitekture entiteta. Ovo je takođe ilustrovano na slici 2.2.3.3, a takođe je dat kod koji definiše entitet (*troulazno_dvoizlazno_kolo_verzija2*) i arhitekturu (*shema_OK*) za ovaj slučaj. Treba primetiti da je kod greške u kodu stavljen komentar koji ukazuje na grešku. Komentari u VHDL kodu se obeležavaju sa -- čime ostatak linije iza ove oznake biva zakomentarisan i kompajler ga ne uzima u obzir. Kao i kod drugih programskih jezika važno je dodavati kvalitetne komentare u kod radi lakše razumljivosti i praćenja koda, ali i lakšeg održavanja koda. Koja varijanta će se koristiti – '*dummy*' signal ili BUFFER mod zavisi samo od programera, jer nakon kompajliranja uglavnom nema razlika. U praksi se uglavnom ređe koristi BUFFER mod i pored pogodnosti koje nudi ovaj mod.

```
ENTITY troulazno_dvoizlazno_kolo_verzija1 IS
PORT
(
        x1, x2, x3 : IN  BIT;
        y1, y2: OUT  BIT
);
END troulazno_dvoizlazno_kolo_verzija1;

ARCHITECTURE shema_greska OF troulazno_dvoizlazno_kolo_verzija1 IS
BEGIN
        y1<= x1 AND x2;
        y2<=y1 OR x3; -- GRESKA jer je y1 OUT port i njegova vrednost se ne može ocitati unutar arhitekture
END shema_greska;

ARCHITECTURE shema_OK OF troulazno_dvoizlazno_kolo_verzija1 IS
        SIGNAL z : BIT;
```

```
BEGIN
        z<= x1 AND x2;
        y1<=z; -- interni signal se može voditi direktno na izlaz, ali istovremeno i
        y2<=z OR x3; -- na ulaz ILI kola koje se nalazi u arhitekturi
END shema_OK;
ENTITY troulazno_dvoizlazno_kolo_verzija2  IS
PORT
(
        x1, x2, x3 : IN  BIT;
        y1: BUFFER  BIT;
        y2: OUT  BIT
);
END troulazno_dvoizlazno_kolo_verzija2;
ARCHITECTURE shema_OK OF troulazno_dvoizlazno_kolo_verzija2 IS
BEGIN
        y1<= x1 AND x2;
        y2<=y1 OR x3; -- OK jer je y1 BUFFER port i njegova vrednost se može ocitati unutar arhitekture
END shema_OK;
```

2.3 Tipovi podataka

U okviru ove sekcije će biti navedeni i ukratko opisani predefinisani tipovi podataka koji se koriste u VHDL-u. Takođe će biti navedeno u kojim bibliotekama su definisani ti tipovi podataka. Posebno će biti naglašeno koji tipovi podataka mogu biti hardverski implementirani, jer kao što je navedeno u uvodnom delu poglavlja jedna od bitnih namena VHDL-a je i olakšavanje procesa testiranja, simulacije i verifikacije hardverskih implementacija. Stoga su, u cilju efikasnijeg postupka simulacije dizajna, definisani i tipovi podataka koji se mogu koristiti samo u okviru simulacije, ali ne mogu da se implementiraju u hardveru. Naravno, tipovi podataka koji se mogu hardverski implementirati se mogu (i moraju) koristiti bez problema u okviru simulacije hardverskog dizajna. Na kraju ove sekcije će biti izloženi načini kako korisnik može da definiše svoje tipove podataka.

2.3.1. Predefinisani tipovi podataka

Predefinisani tipovi podataka koji mogu hardverski da se implementiraju su:

- BIT - Predstavlja jedan signal (liniju) koja može da ima vrednost 0 ili 1 (definisan u paketu *standard* biblioteke *std*). Kada se vrši dodela vrednosti signalu ovog tipa koriste se jednostruki navodnici, na primer:

  ```
  x<='0';
  ```

- BIT_VECTOR - Predstavlja niz signala tipa BIT, ili preciznije sa hardverskog stanovišta predstavlja magistralu (skup linija) gde svaka linija može da ima vrednost 0 ili 1 (definisan u paketu *standard* biblioteke *std*). Ovaj niz može da se deklariše po *little endian* ili *big endian* principu. Deklaracija po *little endian* principu je BIT_VECTOR(*dg* TO *gg*), gde je *dg* najniži indeks tj. donja granica, a *gg* je najviši indeks tj. gornja granica. Deklaracija po *big endian* principu je BIT_VECTOR(*gg* DOWNTO *dg*), gde je *dg* najniži indeks tj. donja granica, a *gg* je najviši indeks tj. gornja granica. U slučaju *litle endian* principa bit sa najvišim indeksom je bit koji se nalazi na desnom kraju i on predstavlja MSB (*Most Significant Bit*), dok u slučaju *big endian* principa bit sa najvišim indeksom je bit koji se nalazi na levom kraju i on

predstavlja MSB. Tako na primer ako se signalu tipa BIT_VECTOR(0 TO 3) dodeli vrednost "1010", MSB bit će dobiti vrednost 0, dok u slučaju signala BIT_VECTOR(3 DOWNTO 0) MSB bit bi dobio vrednost 1. Kada se pišu konkretne vrednosti koje se dodeljuju (kao što je vrednost "1010" u primeru iz prethodne rečenice) prirodnija je upotreba *big endian* jer se vrednost piše s leva na desno kako smo navikli da pišemo, dok se kod *little endian* principa vrednost mora pisati obrnuto tj. naopako što dovodi često do grešaka. Stoga dizajneri najčešće koriste u praksi *big endian* deklaraciju za BIT_VECTOR. Kada se vrši dodela vrednosti signalu tipa BIT_VECTOR koriste se dvostruki navodnici, na primer ako je signal *x* vektor dužine 4:

```
x<="0001";
```

Ako se vrši dodela samo jednom indeksu onda se koriste jednostruki navodnici, na primer:

```
x(2)<= '0';
```

U slučaju kada se vrši dodela opsegu indeksa signala tipa BIT_VECTOR opet se moraju koristiti dvostruki navodnici:

```
x(3 DOWNTO 1)<= "001";
```

Pri tome je važno napomenuti da opseg može biti dužine 1, pri čemu se i dalje moraju koristiti dvostruki navodnici jer je i dalje u pitanju opseg indeksa, na primer:

```
x(3 DOWNTO 3)<= "0";
```

Prethodno navedeni primeri se odnose na upotrebu binarnih vrednosti u dodeli. Pored binarnih mogu se dodeliti i heksadecimalne i oktalne cifre, gde se ispred vrednosti pod dvostrukim navodnicima stavlja X ako je u pitanju heksadcimalni zapis, odnosno O ako je u pitanju oktalni zapis. Na primer:

```
x(7 DOWNTO 0)<=X"1F";--Primer dodele vrednosti u heksadecimalnom zapisu
x(5 DOWNTO 0)<=O"25";--Primer dodele vrednosti u oktalnom zapisu
```

Važno je napomenuti da u slučaju dodele heksadecimalnog zapisa dužina opsega kome se dodeljuje vrednost mora biti deljiva sa 4, a u slučaju oktalnog zapisa deljiva sa 3.

Radi lakše vizualizacije u slučaju dugačkih zapisa se može koristiti '*underscore*' simbol za razdvajanje vrednosti u čitljive blokove, na primer:

```
x<="0001_0111_1100_1010";
```

Važno je naglasiti da kompajleri u pojedinim popularnim razvojnim okruženjima ne dozvoljavaju ovakvo razdvajanje (tretira se kao sintaksna greška) pa je poželjno izbegavati ovu tehniku.

Kada se želi jednom indeksu signala tipa BIT_VECTOR dodeliti vrednost signala tipa BIT, to se radi na sledeći način:

```
y(2)<=x;
```

gde je *x* signal tipa BIT, a *y* signal tipa BIT_VECTOR. U navedenom primeru se indeksu 2 signala *y* tipa BIT_VECTOR dodeljuje vrednost signala *x* tipa BIT. Ako se

želi uraditi suprotno, dodela vrednosti nekog indeksa signala tipa BIT_VECTOR signalu tipa BIT, to se radi na sledeći način:

```
x<= y(2);
```

- STD_LOGIC - Predstavlja proširenje BIT signala (definisan u paketu *std_logic_1164* biblioteke *ieee*). Naime, pokazalo se da logika sa samo dva nivoa (0 i 1) u prvobitnom standardu (VHDL-87) nije dovoljna pa su uvedeni dodatni nivoi signala i definisan je STD_LOGIC tip. STD_LOGIC tip ima sledećih 8 nivoa:
    - 'X' - neodređena vrednost
    - '0' - logička 0
    - '1' - logička 1
    - 'Z' - visoka impedansa
    - 'W' - slab signal neodređene vrednosti
    - 'L' - slab signal logičke 0
    - 'H' - slab signal logičke 1
    - '-' - *don't care* vrednost

  U praksi se najčešće, ipak, koriste samo nivoi 0, 1 i Z jer se ova tri nivoa mogu hardverski implementirati bez ograničenja, dok su ostali nivoi predviđeni za upotrebu pre svega u okviru simulacije gde se od njih najčešće koristi nivo X kao signalizacija nepoznate vrednosti signala. U slučaju direktnog spajanja dva (ili više) signala (linije) na jedan signal (liniju) (npr. izlazi dva I kola se dovedu na jedan od ulaza trećeg I kola) vrši se tzv. razrešavanje konflikta po principima prikazanim u tabeli 2.3.1.1 da bi se odredila vrednost rezultujućeg signala. Važno je naglasiti da direktno spajanje linija treba izbegavati jer to nije dobra praksa, a ako se ipak mora tako nešto uraditi tada treba koristiti trostatičke bafere (napomena: druga poznata metoda koja se koristi za direktno spajanje linija je metoda otvorenog kolektora, ali se ona ne može koristiti u programabilnim čipovima). Takođe, kompajler VHDL koda će prijaviti grešku i prekinuti dalje kompajliranje u slučaju da uoči direktno spajanje linija bez upotrebe na primer trostatičkih bafera. Princip dodele vrednosti signalu tipa STD_LOGIC je identičan onome korišćenom kod tipa BIT, tj. koriste se jednostruki navodnici.

Tabela 2.3.1.1. – Princip razrešavanja konflikta

Signal1 -> / Signal2	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

- STD_LOGIC_VECTOR - Predstavlja niz signala tipa STD_LOGIC, tj. magistralu čiji članovi su signali tipa STD_LOGIC (definisan u paketu *std_logic_1164* biblioteke *ieee*). I ovaj niz se može deklarisati po *little endian* ili *big endian* principu na identičan način kao BIT_VECTOR pri čemu važe isti komentari koji su navedeni kod opisa BIT_VECTOR tipa. Princip dodele vrednosti signalu tipa STD_LOGIC_VECTOR je identičan onome korišćenom kod tipa BIT_VECTOR, koriste se dvostruki navodnici u slučaju kad se dodeljuje kompletna vrednost ili vrednost za opseg indeksa, a kada se dodeljuje vrednost samo jednom indeksu onda se koriste jednostruki navodnici. Dodela vrednosti signala STD_LOGIC tipa jednom indeksu signala tipa STD_LOGIC_VECTOR, i obrnuta dodela se radi identično kao u slučaju BIT i BIT_VECTOR tipova. Takođe važi i princip dodeljivanja vrednosti zapisanih u heksadecimalnom ili oktalnom sistemu kao i kod BIT_VECTOR tipa. Veoma je važno naglasiti da vrednost signala tipa STD_LOGIC (STD_LOGIC_VECTOR) ne može da se dodeli signalu tipa BIT (BIT_VECTOR) i obrnuto, već mora da se radi konverzija o kojoj će kasnije biti reči. Nad signalima tipa STD_LOGIC_VECTOR se mogu vršiti samo logičke operacije, a aritmetičke operacije nisu dozvoljene. Da bi se omogućile aritmetičke operacije mora se koristiti i paket *std_logic_unsigned* ili *std_logic_signed* biblioteke *ieee*, u zavisnosti da li želimo raditi sa nenegativnim celim brojevima ili celim brojevima.
- STD_ULOGIC - Predstavlja proširenje STD_LOGIC signala (definisan u paketu *std_logic_1164* biblioteke *ieee*). Dodat je i deveti nivo signala - 'U' koji se koristi sa signaliziranje konflikata (U - *Unresolved*). Kod STD_LOGIC tipa u slučaju da se na jedan signal direktno vode druga dva (ili više) signala vršilo se razrešavanje konflikta po principima prikazanim u tabeli 2.3.1.1. U slučaju STD_ULOGIC tipa ne radi se razrešavanje konflikta, već u takvoj situaciji rezultujući signal dobija vrednost 'U' tj. nerazrešen. Na ovaj način se lako u simulaciji može utvrditi da su neki signali (linije) međusobno direktno spojeni na rezultujući signal (liniju), pa se lako otkrivaju takve greške tj. neželjene konekcije. Princip dodele vrednosti signalu tipa STD_ULOGIC je identičan onome korišćenom kod tipa BIT, tj. koriste se jednostruki navodnici.
- STD_ULOGIC_VECTOR - Predstavlja niz signala tipa STD_ULOGIC, tj. magistralu čiji članovi su signali tipa STD_ULOGIC (definisan u paketu *std_logic_1164* biblioteke *ieee*). I ovaj niz se može deklarisati po *little endian* ili *big endian* principu na identičan način kao BIT_VECTOR pri čemu važe isti komentari koji su navedeni kod opisa BIT_VECTOR tipa. Princip dodele vrednosti signalu tipa STD_ULOGIC_VECTOR je identičan onome korišćenom kod tipa BIT_VECTOR, koriste se dvostruki navodnici u slučaju kad se dodeljuje kompletna vrednost ili vrednost za opseg indeksa, a kada se dodeljuje vrednost samo jednom indeksu onda se koriste jednostruki navodnici. Dodela vrednosti signala STD_ULOGIC tipa jednom indeksu signala tipa STD_ULOGIC_VECTOR, i obrnuta dodela se radi identično kao u slučaju BIT i BIT_VECTOR tipova. Takođe važi i princip dodeljivanja vrednosti zapisanih u heksadecimalnom ili oktalnom sistemu kao i kod BIT_VECTOR tipa. Veoma je važno naglasiti da vrednost signala tipa STD_ULOGIC (STD_ULOGIC_VECTOR) ne može da se dodeli signalu tipa BIT (BIT_VECTOR) i obrnuto, već mora da se radi konverzija o kojoj će kasnije biti reči. Takođe, vrednost signala tipa STD_ULOGIC_VECTOR ne može da se dodeli signalu tipa STD_LOGIC_VECTOR i obrnuto, već mora da se radi konverzija. Zanimljivo,

dozvoljena je dodela vrednosti signala tipa STD_ULOGIC signalu tipa STD_LOGIC i obrnuto, što je posledica toga da je STD_LOGIC izveden kao podtip tipa STD_ULOGIC, dok su STD_LOGIC_VECTOR i STD_ULOGIC_VECTOR definisani kao posebni tipovi (detaljnije objašnjenje razlike podtipova i tipova će biti objašnjen u sekciji 2.3.2). Nad signalima tipa STD_ULOGIC_VECTOR se mogu vršiti samo logičke operacije, a aritmetičke operacije nisu dozvoljene (i ne mogu se aktivirati preko standardnih biblioteka kao kod STD_LOGIC_VECTOR tipa).

- BOOLEAN - Predstavlja tip koji može imati dve vrednosti *True* i *False* (definisan u paketu *standard* biblioteke *std*). Uglavnom se koristi se pri formiranju uslova.
- INTEGER - Predstavlja ceo 32-bitni broj čije se vrednosti kreću u granicama (-2,147,483,647 do +2,147,483,647). Ovaj tip je definisan u paketu *standard* biblioteke *std*. U principu maksimalan opseg vrednosti INTEGER tipa zavisi od razvojnog okruženja koje se koristi, ali navedeni opseg važi u većini slučajeva. INTEGER u hardverskoj implementaciji u stvari predstavlja 32 linije, gde svaka linija može imati vrednost 0 ili 1. U slučaju da kompajler uvidi da je korišćeni opseg vrednosti signala tipa INTEGER ograničen on može u hardverskoj implementaciji da takav signal implementira sa manje linija. Na primer ako kompajler utvrdi da signal *x* koristi samo vrednosti 0 do 31, signal *x* će biti implementiran sa samo 5 linija (bita). Korisnik sam može da ograniči opseg korišćenih vrednosti u deklaraciji signala sa

  ```
  SIGNAL x: INTEGER RANGE dg TO gg;
  ```

  gde je *dg* donja granica opsega, a *gg* gornja granica opsega. INTEGER tip je pogodan za implementaciju brojača i aritmetičkih funkcija. Signalu tipa INTEGER se dodeljuju vrednosti u decimalnom zapisu, na primer:

  ```
  x<= 55;
  ```

- NATURAL - Predstavlja nenegativan 31-bitni broj čije se vrednosti kreću u granicama (0 do +2,147,483,647). Ovaj tip je definisan u paketu *standard* biblioteke *std*. NATURAL u hardverskoj implementaciji u stvari predstavlja 31 linije, gde svaka linija može imati vrednost 0 ili 1. U slučaju da kompajler uvidi da je korišćeni opseg vrednosti signala tipa NATURAL ograničen on može u hardverskoj implementaciji da takav signal implementira sa manje linija. Korisnik može da ograniči opseg ovog tipa na isti način kao i u slučaju INTEGER tipa. Dodela vrednosti signalu tipa NATURAL se radi na identičan način kao i kod INTEGER tipa.
- POSITIVE - Predstavlja pozitivan 31-bitni broj čije se vrednosti kreću u granicama (1 do +2,147,483,647). Ovaj tip je definisan u paketu *standard* biblioteke *std*. POSITIVE u hardverskoj implementaciji u stvari predstavlja 31 linije, gde svaka linija može imati vrednost 0 ili 1. U slučaju da kompajler uvidi da je korišćeni opseg vrednosti signala tipa POSITIVE ograničen on može u hardverskoj implementaciji da takav signal implementira sa manje linija. Korisnik može da ograniči opseg ovog tipa na isti način kao i u slučaju INTEGER tipa. Dodela vrednosti signalu tipa POSITIVE se radi na identičan način kao i kod INTEGER tipa.
- SIGNED - Predstavlja vektor namenjen aritmetičkim operacijama i definisan je u paketu *std_logic_arith* biblioteke *ieee*. Naime, kao što je već prethodno navedeno STD_LOGIC_VECTOR tip je namenjen samo za logičke operacije, a za uključenje

podrške za aritmetičke operacije potrebna je dodatna biblioteka (tačnije paket). SIGNED predstavlja formu STD_LOGIC_VECTOR tipa namenjenu za aritmetičke operacije, gde SIGNED tip predstavlja takođe vektor bita koji se tretira kao integer tj. ceo broj. Koristi se sistem komplementa 2. Najviši bit određuje da li je broj pozitivan (0) ili negativan (1). U slučaju negativnog broja njegova apsolutna vrednost se računa kao 2^n-x, gde x predstavlja vrednost kompletnog vektora posmatranog kao pozitivan broj, a n predstavlja broj bita koji se koristi u vektoru. Na primer, ako je dužina vektora 4, tada 0101 predstavlja 5, a 1101 predstavlja -3 (2^4-13=3). SIGNED tip ne podržava logičke operacije. Dodela vrednosti se radi na identičan način kao kod STD_LOGIC_VECTOR tipa (dodela vrednosti kompletnom signalu, dodela vrednosti opsegu indeksa, dodela jednom indeksu, dodela vrednosti u heksadecimalnom/oktalnom zapisu, *little endian* i *big endian* princip). Ovaj tip je pogodan ako se u dizajnu rade samo aritmetičke operacije, ali s obzirom na nedostatak podrške za logičke operacije, znatno praktičnije je korišćenje STD_LOGIC_VECTOR tipa gde se sa dodatnim paketom uključuje podrška i aritmetičkim operacijama.

- UNSIGNED - Predstavlja vektor namenjen aritmetičkim operacijama nad nenegativnim celim brojevima i definisan je u paketu *std_logic_arith* biblioteke *ieee*. Važe identične osobine kao kod SIGNED tipa uz razliku da se vektori tretiraju kao nenegativni celi brojevi.

Kao rezime možemo reći da su najkorišćeniji tipovi STD_LOGIC, STD_LOGIC_VECTOR i INTEGER. SIGNED i UNSIGNED tipovi nisu previše praktični jer ne dozvoljavaju logičke operacije, a uključivanjem dodatnog paketa se dobija podrška za aritmetičke operacije kod STD_LOGIC_VECTOR tipa pored postojeće podrške za logičke operacije, što ga čini kompletnijim tipom. STD_ULOGIC_VECTOR nije zgodan zbog nedostatka podrške za aritmetičke operacije. STD_ULOGIC i STD_ULOGIC_VECTOR nisu preterano korisni za upotrebu u odnosu na STD_LOGIC i STD_LOGIC_VECTOR tipove, jer proširenje koje sadrže se može koristiti samo u simulaciji, pri čemu je i ta prednost zanemarljiva. POSITIVE i NATURAL tipovi takođe nisu preterano pogodni jer se INTEGER tip lako može ograničiti na željeni opseg. BIT i BIT_VECTOR tipovi takođe nemaju podršku za aritmetičke funkcije (a takođe ne mogu da se koriste u slučaju implementacije trostatičkih bafera), pa su se stoga STD_LOGIC i STD_LOGIC_VECTOR tipovi nametnuli kao najfleksibilnije rešenje za upotrebu.

VHDL omogućava i upotrebu tipova koji se ne mogu hardverski implementirati (nemaju svoj hardverski ekvivalent) i koji se mogu koristiti za potrebe simulacije ili za potrebe parametrizacije dizajna radi postizanja veće fleksibilnosti. Predefinisani tipovi podataka koji ne mogu hardverski da se implementiraju su:

- REAL - Predstavlja realan broj u opsegu od -10^{38} do $+10^{38}$. Ovaj tip je definisan u paketu *standard* biblioteke *std*.
- CHARACTER - Predstavlja ASCII karakter. Ovaj tip je definisan u paketu *standard* biblioteke *std*. Karakteri koji su vidljivi (*printable*) se pišu pod jednostrukim navodnicima (npr. 'A'), dok se karakteri koji nisu vidljivi (*non printable*) pišu svojim specijalnim nazivom (npr. NUL).

- STRING - Predstavlja niz ASCII karaktera. Ovaj tip je definisan u paketu *standard* biblioteke *std*. Vrednost koja se dodeljuje stringu se mora pisati pod dvostrukim navodnicima (npr. "tekst").
- SEVERITY_LEVEL – Definiše nivo poruke (tj. stepen događaja) u izveštaju koji se generisao u toku simulacije. Koristi se u okviru ASSERT izraza. Ovaj tip je definisan u paketu *standard* biblioteke *std*. Vrednosti ovog tipa su: *note*, *warning*, *error* i *failure* koji određuju stepen događaja o kom se izveštava i na osnovu naziva vrednosti je prilično jasno na šta se odnosi svaka od navedenih vrednosti. Primer upotrebe ASSERT izraza i korišćenja SEVERITY_LEVEL tipa će biti dat kasnije u skripti.
- TIME - Predstavlja jedinicu vremena. Osnovna jedinica je fs (femtosekunda). Ostale vrednosti se grade na osnovu ove osnovne jedinice. Tako je 1ps = 1000fs, 1ns = 1000ps itd. Podržane jedinice su (fs, ps, ns, us, ms, sec, min, hr). Vrednost koja se stavlja ispred jedinice vremena mora biti ceo broj koji se tipično kreće u granicama identičnim granicama INTEGER tipa, ali razvojno okruženje može da definiše i drugačiji opseg. Tipična upotreba ovog tipa je u okviru simulacija: za definisanje vremenskog redosleda događaja, za simuliranje kašnjenja kola u simulaciji, i sl. Ovaj tip je definisan u paketu *standard* biblioteke *std*.
- DELAY_LENGTH - Predstavlja dužinu kašnjenja u jedinicama vremena koje su definisane u tipu TIME. DELAY_LENGTH u stvari predstavlja podtip tipa TIME tako što su dozvoljene samo nenegativne vrednosti, za razliku od TIME tipa gde su moguće i negativne vrednosti. DELAY_LENGTH je definisan u paketu *standard* biblioteke *std*.
- FILE_OPEN_KIND - Predstavlja mod rada otvorenog fajla - čitanje (*read_mode*), pisanje sa prethodnim brisanjem sadržaja fajla (*write_mode*), dopisivanje na kraj fajla (*append_mode*). Koristi se u okviru simulacije za učitavanje vrednosti koje se koriste za stimulisanje određenih situacija ili za ispis vrednosti rada simulacije. Ovaj tip je definisan u paketu *standard* biblioteke *std*.
- FILE_OPEN_STATUS - Predstavlja rezultat otvaranja nekog fajla. Fajl može biti uspešno otvoren (*open_ok*) ili neuspešno otvoren (*status_error*, *name_error*, *mode_error*). Ovaj tip se koristi za proveru ispravnosti otvaranja fajla za potrebe simulacije. Ovaj tip je definisan u paketu *standard* biblioteke *std*.

Rezime predefinisanih podataka je dat u tabeli 2.3.1.2.

Tabela 2.3.1.2. – Predefinisani tipovi podataka

Tip	Objašnjenje	Biblioteka	Mogućnost hardverske implementacije
BIT	Binarni signal	Paket *standard* biblioteke *std*	Da
BIT_VECTOR	Niz (vektor) bita	Paket *standard* biblioteke *std*	Da
STD_LOGIC	Proširenje BIT tipa sa dodatnim stanjima	Paket *std_logic_1164* biblioteke *ieee*	Da
STD_LOGIC_VECTOR	Niz (vektor) STD_LOGIC-a	Paket *std_logic_1164* biblioteke *ieee*	Da
STD_ULOGIC	Proširenje STD_LOGIC tipa sa nerazrešenim stanjem ('U')	Paket *std_logic_1164* biblioteke *ieee*	Da
STD_ULOGIC_VECTOR	Niz (vektor) STD_ULOGIC-a	Paket *std_logic_1164* biblioteke *ieee*	Da
SIGNED	STD_LOGIC_VECTOR ekvivalent za aritmetičke operacije u *signed* režimu	Paket *std_logic_arith* biblioteke *ieee*	Da
UNSIGNED	STD_LOGIC_VECTOR ekvivalent za aritmetičke operacije u *unsigned* režimu	Paket *std_logic_arith* biblioteke *ieee*	Da
BOOLEAN	Logička binarna promenjiva	Paket *standard* biblioteke *std*	Da
INTEGER	Ceo broj	Paket *standard* biblioteke *std*	Da
NATURAL	Ceo nenegativan broj	Paket *standard* biblioteke *std*	Da
POSITIVE	Ceo pozitivan broj	Paket *standard* biblioteke *std*	Da
REAL	Realan broj	Paket *standard* biblioteke *std*	Ne
CHARACTER	ASCII karakter	Paket *standard* biblioteke *std*	Ne
STRING	Niz ASCII karaktera	Paket *standard* biblioteke *std*	Ne
SEVERITY_LEVEL	Nivo značaja generisane poruke u simulaciji	Paket *standard* biblioteke *std*	Ne
TIME	Vreme	Paket *standard* biblioteke *std*	Ne
DELAY_LENGTH	Dužina kašnjenja	Paket *standard* biblioteke *std*	Ne
FILE_OPEN_KIND	Mod rada otvorenog fajla	Paket *standard* biblioteke *std*	Ne
FILE_OPEN_STATUS	Rezultat otvaranja fajla	Paket *standard* biblioteke *std*	Ne

2.3.2. Korisnički definisani tipovi podataka

Pored predefinisanih tipova, sam korisnik takođe može da definiše svoje tipove podataka. Tipično se definišu enumerisani tipovi koji se koriste za dizajniranje konačnih automata (mašina stanja) gde korisnički definisan enumerisani tip sadrži logičke nazive za stanja automata i samim tim čitav dizajn čini lako razumljivim i preglednijim. Druga tipična namena je definisanje nizova i matrica. Definisanje novog tipa se postiže upotrebom ključne reči TYPE i može se uraditi u okviru dela za deklaracije u arhitekturi ili u okviru paketa.

Korisnik može da kreira svoj tip kao podskup INTEGER opsega, što može biti poželjno u slučaju da se identično ograničenje koristi za veći broj signala i naročito ako se koristi u više VHDL kodova istog projekta. Definisanje novog tipa naziva *kratki_integer* koji će da koristi samo opseg 0-31 se radi na sledeći način:

```
TYPE kratki_integer IS RANGE 0 TO 31;
```

Kreiranje novog enumerisanog tipa se radi na sledeći način:

```
TYPE stanje_automata IS (init, mod_citanja, mod_upisa, idle);
```

Definisani tip *stanje_automata* ima četiri vrednosti koje u ovom slučaju deskriptivno opisuju stanja konačnog automata, ukoliko bi se ovako korisnički definisani tip koristio za opisivanje konačnog automata koji sadrži stanja inicijalizacije, modova za čitanje i upis i besposlenog stanja. Naravno, korisnički definisani enumerisani tip se ne mora koristiti samo za opis konačnog automata, ali mu to ipak jeste najčešća namena. Članovi enumerisanog tipa se automatski kodiraju (sem ako korisnik nije drugačije specificirao), pri čemu se koristi minimalan broj bita neophodan za kodiranje svih članova (minimalan pozitivan ceo broj koji je veći ili jednak od $\log_2 N$ gde je N broj članova definisanog enumerisanog tipa). U datom primeru članovi enumerisanog tipa *stanje_automata* bi se kodirali sa dva bita (’00’-*init*, ’01’-*mod_citanja*, ’10’-*mod_upisa*, ’11’-*idle*).

Korisnički definisani tipovi se mogu koristiti i za formiranje nizova i matrica. U suštini pojedini predefinisani tipovi predstavljaju nizove. Tako, na primer, BIT_VECTOR predstavlja u suštini niz bita (tip BIT). Slično važi i za tipove STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR, SIGNED i UNSIGNED (SIGNED i UNSIGNED tipovi u suštini predstavljaju nizove STD_LOGIC-a samo što su namenjeni za implementaciju aritmetičkih, a ne logičkih funkcija). Korisničko definisanje tipa sličnog navedenim *vector* tipovima se radi na sledeći način:

```
TYPE moj_std_logic_vector IS ARRAY (3 DOWNTO 0) OF STD_LOGIC;--ekvivalent za STD_LOGIC_VECTOR(3 DOWNTO 0)
TYPE moj_bit_vector IS ARRAY (0 TO 3) OF BIT;--ekvivalent za BIT_VECTOR(0 TO 3)
```

Definisanje tipa koji predstavlja matricu se može izvršiti na nekoliko načina:

```
TYPE matrica1 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(3 DOWNTO 0);--matrica 4x4 STD_LOGIC-a
TYPE matrica2 IS ARRAY (0 TO 3) OF moj_std_logic_vector;-- matrica 4x4 STD_LOGIC-a
TYPE matrica3 IS ARRAY (0 TO 3, 3 DOWNTO 0) OF STD_LOGIC;--matrica 4x4 STD_LOGIC-a
```

U sva tri slučaja su definisane identične matrice dimenzija 4x4. Red matrice u sva tri slučaja predstavlja *big endian* niz STD_LOGIC-a, pri čemu su *matrica1* i *matrica2* definisane kao niz nizova, a *matrica3* je definisana kao klasična matrica gde se koriste dva indeksa. Broj indeksa u ARRAY delu (u sva tri slučaja) može biti veći (npr. ARRAY (0 TO 3, 0 TO 2, 0 TO 4)), čime se mogu formirati i višedimenzionalne matrice. Dodele vrednosti signalima koji koriste navedene primere tipova matrica, kao i međusobne razmene vrednosti takvih signala se rade na sledeći način:

```
SIGNAL a: moj_std_logic_vector;
SIGNAL x: matrica1;
SIGNAL y: matrica2;
SIGNAL z: matrica3;
....
a<="0110";--OK dodela vrednosti
a(2)<='1';--OK dodela vrednosti indeksu 2
a(2 DOWNTO 0)<="010";--OK dodela vrednosti indeksima 2 do 0
x(1)<="0110";--OK dodela vrednosti redu 1
```

```
x(1)(2 DOWNTO 0)<="010";--OK dodela indeksima 2 do 0 reda 1
x(2)(2)<='1';--OK dodela vrednosti clanu 2,2 matrice
y(1)<="0110";-- OK dodela vrednosti redu 1
y(1)(2 DOWNTO 0)<="010";-- OK dodela indeksima 2 do 0 reda 1
y(2)(2)<='1';-- OK dodela vrednosti clanu 2,2 matrice
z(1, 3 DOWNTO 0)<="0110";-- GRESKA, matrici sa dva indeksa se moze pristupati samo clan po clan
z(1, 2 DOWNTO 0)<="010";-- GRESKA, matrici sa dva indeksa se moze pristupati samo clan po clan
z(2, 2)<='1';-- OK dodela vrednosti clanu 2,2 matrice
z(0 to 3, 0)<="0110";-- GRESKA, matrici sa dva indeksa se moze pristupati samo clan po clan
a<= x(1);--GRESKA razliciti tipovi moj_std_logic_vector i STD_LOGIC_VECTOR(3 DOWNTO 0)
a<= y(1);--OK jer su isti tipovi (moj_std_logic_vector)
a(1)<= x(1)(1);--OK oba tipa STD_LOGIC
a(1)<= y(1)(1);-- OK oba tipa STD_LOGIC
a(1)<= z(1, 1);-- OK oba tipa STD_LOGIC
y(1)<= x(1);--GRESKA razliciti tipovi moj_std_logic_vector i STD_LOGIC_VECTOR(3 DOWNTO 0)
```

Treba primetiti da u slučaju matrice sa dva indeksa (*matrica3*) se može pristupati samo jednom članu matrice, za razliku od druga dva slučaja (*matrica1*, *matrica2*) gde se može pristupiti odjednom čitavom redu ili delu reda. Razlog naravno leži u načinu definisanja matrice. Naime, u slučaju višedimenzionalnih nizova mora se pristupati član po član, odnosno može se indeksirati samo jedan član, ali ne i više članova odjednom (nije dozvoljena upotreba opsega indeksa u slučaju kada postoji više od jedne dimenzije). Pošto je u slučaju *matrica1* i *matrica2* u pitanju niz nizova (oba niza su jednodimenzionalna) onda se može odjednom pristupiti više članova matrice u jednom redu, tj. preciznije rečeno može se pristupati jednom redu, a u okviru reda se može indeksirati više članova reda, kao što smo videli u prethodnim primerima dodele. U slučaju *matrica3* je u pitanju dvodimenzionalni niz (tj. klasična matrica), pa se u obe dimenzije može selektovati samo jedan indeks, čime se selektuje samo jedan član matrice. Mogu se definisati i matrice sa više dimenzija primenom bilo kojeg od navedenih pristupa za formiranje matrice ili kombinacijom navedenih pristupa, ali takav pristup uglavnom treba izbegavati.

Nizu ili matrici se može dodeliti konstantna vrednost svim njihovim članovima unutar samo jednog izraza:

```
a<="0110";-- dodela konstantne vrednosti svim clanovima niza
a<=('0', '1', '1', '0');-- drugi nacin dodele konstantne vrednosti svim clanovima niza
x<=("0110", "1001","1010","0001");-- dodela konstantne vrednosti svim redovima matrice
x<=(('0', '1', '1', '0'), ('0', '1', '1', '0'), ('0', '1', '1', '0'), ('0', '1', '1', '0'));-- drugi nacin dodele konstantne vrednosti svim redovima matrice
y<=("0110", "1001","1010","0001");-- dodela konstantne vrednosti svim redovima matrice
y<=(('0', '1', '1', '0'), ('0', '1', '1', '0'), ('0', '1', '1', '0'), ('0', '1', '1', '0'));-- drugi nacin dodele konstantne vrednosti svim redovima matrice
z<=("0110", "1001","1010","0001");-- dodela konstantne vrednosti svim redovima matrice
z<=(('0', '1', '1', '0'), ('0', '1', '1', '0'), ('0', '1', '1', '0'), ('0', '1', '1', '0'));-- drugi nacin dodele konstantne vrednosti svim redovima matrice
```

Kao što vidimo u sva tri načina definisanja matrice princip dodele konstantne vrednosti kompletnoj matrici je isti. U slučaju kada se matrica definiše kao niz nizova, onda se može dodeliti konstantna vrednost jednom ili više kompletnih redova matrice (kao što je pokazano u narednom primeru), dok to nije moguće kod definisanja matrice kao dvodimenzionalnog niza jer nije dozvoljen opseg indeksa. Nije moguće dodeliti konstantnu vrednost delu reda za više redova odjednom (npr. samo prvom članu svih redova matrice).

```
x(1 TO 2)<=("1001","1010");-- dodela konstantne vrednosti 2. i 3. redu matrice
x(1 TO 2)<=(('0', '1', '1', '0'), ('0', '1', '1', '0'));-- drugi nacin dodele konstantne vrednosti 2. i 3. redu matrice
y(1 TO 2)<=("1001","1010");-- dodela konstantne vrednosti 2. i 3. redu matrice
y(1 TO 2)<=(('0', '1', '1', '0'), ('0', '1', '1', '0'));-- drugi nacin dodele konstantne vrednosti 2. i 3. matrice
```

Nekada je zgodno u dizajnu definisati niz portova entiteta. Međutim, da bi se tako nešto uradilo mora se definisati novi tip, a jedino mesto gde se to može uraditi je paket. U ovom

slučaju definisanje novog tipa se ne može uraditi u arhitekturi jer će se novi tip koristiti u deklaraciji entiteta (znači interfejsa dizajna), a ne u unutrašnjosti (arhitekturi) dizajna. Primer gde se definiše niz ulaznih i izlaznih portova entiteta *linijski_moduli*:

```
--Deklaracija paketa koji definise tip koji ce da se koristi u entitetu za formiranje niza portova
LIBRARY ieee;
USE ieee.std_logic_1164.all;

PACKAGE moj_paket IS
        TYPE niz_portova IS ARRAY (0 to 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
END moj_paket;

--deklaracija entiteta
LIBRARY ieee;
USE ieee.std_logic_1164.all;
--biblioteka work je ukljucena po difoltu, ne treba njen poziv preko LIBRARY kljucne reci
USE work.moj_paket.all;--koristi se samo USE komanda da bi se kreirani paket ukljucio

ENTITY linijski_moduli  IS
PORT
(
    pin: IN niz_portova;
    pout: OUT niz_portova
);
END linijski_moduli;
```

Kao što vidimo, *pin* i *pout* predstavljaju nizove od 4 porta, pri čemu je svaki port tipa STD_LOGIC_VECTOR(7 DOWNTO 0). Naravno, fleksibilnije bi bilo kada bi mogli definisati niz portova gde proizvoljno možemo da definišemo broj članova niza. Da bi se omogućila takva osobina, potrebna je mala izmena u definiciji tipa *niz_portova* paketu iz primera:

```
TYPE niz_portova IS ARRAY (NATURAL RANGE <>) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

Kada se za opseg stavi oznaka <>, to znači da je opseg nedefinisan i da ga korisnik postavlja prilikom deklaracije porta (ili promenjive u arhitekturi ako bi se tu koristio definisani tip). NATURAL označava da je u pitanju opseg nenegativnih brojeva, a da je umesto NATURAL stavljen INTEGER bili bi dozvoljeni i negativni brojevi u opsegu. S obzirom da je opseg nedefinisan mora ispred ključne reči RANGE da ide definicija tipa na koji se odnosi opseg (INTEGER, NATURAL ili POSITIVE). U ovom slučaju je deklaracija ulaza *pin* (isto važi i za izlaz *pout*) u entitetu *linijski_moduli*:

```
pin: IN niz_portova (0 TO 3);
```

Kao što se vidi opseg je sada prisutan prilikom deklaracije ulaza *pin* za razliku od prethodnog primera gde opseg nije bio prisutan jer je već bio definisan u okviru definicije tipa. Princip nedefinisanog opsega se može primeniti i u slučaju definisanja tipova koji bi predstavljali nizove ili matrice, radi postizanja veće fleksibilnosti. Pored nizova mogu se kreirati i matrice portova, ali se one veoma retko koriste u praksi. Izmena u definisanju tipa u paketu bi bila:

```
TYPE matrica_portova IS ARRAY (NATURAL RANGE <>, NATURAL RANGE <>) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

dok bi u entitetu deklaracija jednog ulaza u vidu matrice portova bila:

```
pin: IN matrica_portova (0 TO 3, 0 TO 3);
```

U ovom primeru je takođe korišćen fleksibilan pristup, ali su se u definiciji tipa *matrica_portova* mogli koristiti i unapred definisani opsezi, kao u prvom primeru za definisanje niza portova. Matrica portova se može definisati i kao niz nizova, pri čemu samo jedna dimenzija

(dimenzija koja se odnosi na red matrice) sme imati nedefinisani opseg. Primer takvog definisanja matrice portova je:

```
--Deklaracija paketa koji definise tip koji ce da se koristi u entitetu za formiranje matrice portova
LIBRARY ieee;
USE ieee.std_logic_1164.all;
PACKAGE moj_paket IS
   TYPE niz_portova IS ARRAY (0 to 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
   TYPE matrica_portova IS ARRAY (NATURAL RANGE <>) OF niz_portova;
END moj_paket;
....
   --deklaracija jednog ulaza entiteta kao matrice portova
   pin: IN matrica_portova (0 TO 3);
....
```

Korisnik može definisati i tzv. zapis (RECORD) koji predstavlja kolekciju tipova koji mogu biti isti ili različiti. U suštini ovakva definicija se može koristiti ukoliko želimo da deo dizajna obuhvatimo u jednu zajednički referisanu strukturu radi kreiranja razumljivijeg koda. Primer jednog zapisa koji bi se koristio za grupisanje relevantnih polja zaglavlja IP paketa:

```
TYPE polja_zaglavlja IS RECORD
        ToS : STD_LOGIC_VECTOR(7 DOWNTO 0);
        Dest_IP_Addr : STD_LOGIC_VECTOR(31 DOWNTO 0);
        Src_IP_Addr : STD_LOGIC_VECTOR(31 DOWNTO 0);
        TTL : STD_LOGIC_VECTOR(7 DOWNTO 0);
END RECORD;
SIGNAL zaglavlje: polja_zaglavlja;
....
zaglavlje.ToS<="01101000";
```

Kao što se vidi iz primera, delovima zapisa se pristupa slično kao u C jeziku, iza promenjive koja je tipa dotičnog zapisa se stavlja tačka pa se iza tačke stavlja naziv odgovarajućeg dela (polja) zapisa.

Pored novog tipa, korisnik može da kreira i podtip, pri čemu koristi ključnu reč SUBTYPE (mesta gde se može definisati podtip su ista kao i u slučaju definisanja tipa). Podtip se kreira ako se želi dodatno ograničiti neki već definisani (predefinisani ili korisnički definisani) tip. Razlog za kreiranje podtipa, a ne novog tipa leži u činjenici da se ne mogu vršiti direktne razmene vrednosti između različitih tipova, dok takvo ograničenje ne postoji u slučaju podtipa i izvornog tipa (od kog je dotični podtip nastao). Najbolji primer je mogućnost razmene vrednosti između signala tipa STD_LOGIC i STD_ULOGIC jer je STD_LOGIC definisan kao podtip STD_ULOGIC tipa. S druge strane razmena vrednosti između STD_LOGIC_VECTOR i STD_ULOGIC_VECTOR tipova nije moguća jer su ova dva tipa definisana zasebno. Primeri definisanja podtipova:

```
SUBTYPE kratki_integer1 IS INTEGER RANGE 0 TO 31;
SUBTYPE veoma_kratki_integer IS kratki_integer RANGE 0 TO 7;
SUBTYPE radno_stanje IS stanje_automata RANGE mod_citanja TO idle;
```

Treba primetiti da kod definisanja podtipa mora da se navede izvorni tip, jer u suprotnom bi bilo nemoguće odrediti šta je izvorni tip definisanog podtipa. Razlog za upotrebu podtipa se može jasno videti iz sledećeg primera:

```
SIGNAL x: kratki_integer;
SIGNAL y: kratki_integer1;
SIGNAL z: INTEGER;
....
```

```
z<=x;-- GREŠKA jer signali nisu istog tipa
z<=y;--OK, jer je kratki_integer1 podtip INTEGER tipa
y<=z;-- kod ovakvih dodela se mora paziti da se ne probije opseg podtipa jer izvorni tip ima širi opseg
```

2.3.3. Konverzije tipova

Kao što je rečeno u prethodnoj sekciji, razmena vrednosti i uopšte interoperabilnost između podataka različitih tipova nije moguća. Međutim, često je u dizajnu neizbežna upotreba više tipova pri čemu je potrebno ostvariti razmenu vrednosti između takvih podataka koji su različitog tipa. Da bi se omogućila razmena vrednosti između podataka različitih tipova, mora se koristiti konverzija tipova. U slučaju da su oba tipa koja treba međusobno da razmenjuju vrednosti, poreklom od istog izvornog tipa konverzija se može izvršiti na sledeći način:

```
TYPE longint IS INTEGER RANGE 0 to 100;
TYPE shortint IS RANGE 0 to 10;
TYPE negint IS RANGE -10 to 10;

SIGNAL l:longint;
SIGNAL s:shortint;
SIGNAL n:negint;
.....
l<=longint(s);--OK konverzija iz shortint u longint
s<=shortint(l);--OK konverzija iz longint u shortint, ali treba paziti da vrednost l ne izadje iz opsega shortint
l<=longint(n);--GRESKA jer donja granica negint ne upada u opseg longint
n<=negint(l);-- OK konverzija iz longint u negint, ali treba paziti da vrednost l ne izadje iz opsega negint
s<=shortint(n);--GRESKA jer donja granica negint ne upada u opseg shortint
n<=negint(s);--OK konverzija iz shortint u negint
```

Iz prethodnih primera se može uočiti da se u slučaju tipova porekla od istog izvornog tipa (ovde je to INTEGER) konverzija postiže primenom funkcije istog imena kao i tip u koji želimo prevesti vrednost drugog tipa. Međutim, ova primena je prilično kruta, jer kao što se iz primera vidi, provera da li može da se radi konverzija se vrši samo na osnovu donje granice, tj. proverava se da li donja granica tipa koji se konvertuje upada u opseg tipa koji predstavlja ishod konverzije. Na primer, da je za *longint* stavljena donja granica -20, tada bi *longint* u potpunosti obuhvatao opseg koji zauzima *negint*. Ali, negint(*l*) sada ne bi bio moguć jer nova donja granica za *longint* (-20) nije deo opsega *negint* i samim tim konverzija ne može da se izvrši. U praksi se ovakav tip konverzija retko koristi.

Znatno češće se koriste konverzije između predefinisanih tipova i takve konverzije su uglavnom definisane u paketu *std_logic_arith* biblioteke *ieee*. Te konverzije su:

- CONV_INTEGER(*x*) - konverzija tipa signala *x* u INTEGER tip. Tip signala *x* može biti INTEGER, UNSIGNED, SIGNED, STD_ULOGIC kada se koristi paket *std_logic_arith* biblioteke *ieee*. Tip signala *x* može biti STD_LOGIC_VECTOR kada se koristi paket *std_logic_signed* ili paket *std_logic_unsigned* biblioteke *ieee*.
- CONV_SIGNED(*x, b*) - konverzija tipa signala *x* u SIGNED tip dužine *b* bita. Tip signala *x* može biti INTEGER, UNSIGNED, SIGNED, STD_ULOGIC kada se koristi paket *std_logic_arith* biblioteke *ieee*.
- CONV_UNSIGNED(*x, b*) - konverzija tipa signala *x* u UNSIGNED tip dužine *b* bita. Tip signala *x* može biti INTEGER, UNSIGNED, SIGNED, STD_ULOGIC kada se koristi paket *std_logic_arith* biblioteke *ieee*.

- CONV_STD_LOGIC_VECTOR(*x, b*) - konverzija tipa signala *x* u STD_LOGIC_VECTOR tip dužine *b* bita. Tip signala *x* može biti INTEGER, UNSIGNED, SIGNED, STD_ULOGIC kada se koristi paket *std_logic_arith* biblioteke *ieee*.
- TO_BIT(*x,u*) - konverzija tipa signala *x* u BIT tip. Tip signala *x* može biti STD_ULOGIC kada se koristi paket *std_logic_1164* biblioteke *ieee*. Vrednost *u* je tipa BIT i specificira rezultat konverzije u slučaju kada vrednost signala *x* nije '0', '1', 'H' ili 'L', tj. kada je neodređena situacija. Ako se *u* ne navede onda se podrazumeva da je taj parametar setovan na '0'.
- TO_BITVECTOR(*x,u*) - konverzija tipa signala *x* u BIT_VECTOR tip. Tip signala *x* može biti STD_ULOGIC_VECTOR, STD_LOGIC_VECTOR kada se koristi paket *std_logic_1164* biblioteke *ieee*. Važno je da svi vektori budu iste dužine, pošto se ne specificira dužina rezultujućeg vektora u konverziji. Vrednost *u* je tipa BIT i specificira rezultat konverzije na određenoj poziciji u slučaju kada vrednost odgovarajućeg (na istoj poziciji) bita *x* nije '0', '1', 'H' ili 'L', tj. kada je neodređena situacija. Ako se *u* ne navede onda se podrazumeva da je taj parametar setovan na '0'.
- TO_STDLOGICVECTOR(*x*) - konverzija tipa signala *x* u STD_LOGIC_VECTOR tip. Tip signala *x* može biti STD_ULOGIC_VECTOR, BIT_VECTOR kada se koristi paket *std_logic_1164* biblioteke *ieee*. Važno je da svi vektori budu iste dužine, pošto se ne specificira dužina rezultujućeg vektora u konverziji.
- TO_STDULOGICVECTOR(*x*) - konverzija tipa signala *x* u STD_ULOGIC_VECTOR tip. Tip signala *x* može biti STD_LOGIC_VECTOR, BIT_VECTOR kada se koristi paket *std_logic_1164* biblioteke *ieee*. Važno je da svi vektori budu iste dužine, pošto se ne specificira dužina rezultujućeg vektora u konverziji.
- TO_STDULOGIC(*x*) - konverzija tipa signala *x* u STD_ULOGIC tip. Tip signala *x* može biti BIT kada se koristi paket *std_logic_1164* biblioteke *ieee*.

Signal *x* u navedenim koverzijama može biti i rezultat logičke ili aritmetičke operacije koja za rezultat ima odgovarajući tip signala. Primeri konverzije (napomena - svugde gde se koriste STD_LOGIC_VECTOR signali mora biti uključen paket *std_logic_1164* biblioteke *ieee*):

```
SIGNAL n, m : INTEGER;
SIGNAL x1,x2 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL y1,y2 : UNSIGNED (7 DOWNTO 0);
SIGNAL b1,b2 : BIT_VECTOR(7 DOWNTO 0);
.....
n<=CONV_INTEGER(x1);--mora biti ukljucen std_logic_unsigned(ili signed) paket biblioteke ieee
n<=CONV_INTEGER(y1);--mora biti ukljucen std_logic_arith paket biblioteke ieee
n<=CONV_INTEGER(x1+x2);--mora biti ukljucen std_logic_unsigned(ili signed) paket biblioteke ieee
n<=CONV_INTEGER(x1 OR x2);--mora biti ukljucen std_logic_unsigned(ili signed) paket biblioteke ieee
n<=CONV_INTEGER(y1+y2);--mora biti ukljucen std_logic_arith paket biblioteke ieee
x1<=CONV_STD_LOGIC_VECTOR(n, 8);--mora biti ukljucen std_logic_arith paket biblioteke ieee
x1<=CONV_STD_LOGIC_VECTOR(n + m, 8);--mora biti ukljucen std_logic_arith paket biblioteke ieee
x1<=CONV_STD_LOGIC_VECTOR(y1, 8);--mora biti ukljucen std_logic_arith paket biblioteke ieee
x1<=CONV_STD_LOGIC_VECTOR(y1 + y2, 8);--mora biti ukljucen std_logic_arith paket biblioteke ieee
y1<=CONV_UNSIGNED(n, 8);--mora biti ukljucen std_logic_arith paket biblioteke ieee
y1<=CONV_UNSIGNED(n + m, 8);--mora biti ukljucen std_logic_arith paket biblioteke ieee
```

```
b1<=TO_BITVECTOR(x1);--mora biti ukljucen std_logic_1164 paket biblioteke ieee, u nije naveden ('0' se onda koristi)
b1<=TO_BITVECTOR(x1 AND x2, '1');--mora biti ukljucen std_logic_1164 paket biblioteke ieee,
b1<=TO_BITVECTOR(x1 + x2);--mora biti ukljucen i std_logic_unsigned(ili signed) paket biblioteke ieee
x1<=TO_STDLOGICVECTOR (b1);--mora biti ukljucen std_logic_1164 paket biblioteke ieee
x1<=TO_STDLOGICVECTOR (b1 OR b2);--mora biti ukljucen std_logic_1164 paket biblioteke ieee
```

Rezime konverzija je dat u tabeli 2.3.3.1:

Tabela 2.3.3.1. – Pregled standardnih konverzija

Funkcija	Iz	U	Biblioteka
CONV_INTEGER(*x*)	INTEGER, SIGNED, UNSIGNED, STD_ULOGIC	INTEGER	Paket *std_logic_arith* biblioteke *ieee*
CONV_INTEGER(*x*)	STD_LOGIC_VECTOR	INTEGER	Paket *std_logic_unsigned ili std_logic_signed* biblioteke *ieee*
CONV_SIGNED(*x,b*)	INTEGER, SIGNED, UNSIGNED, STD_ULOGIC	SIGNED	Paket *std_logic_arith* biblioteke *ieee*
CONV_UNSIGNED(*x,b*)	INTEGER, SIGNED, UNSIGNED, STD_ULOGIC	UNSIGNED	Paket *std_logic_arith* biblioteke *ieee*
CONV_STD_LOGIC_VECTOR(*x,b*)	INTEGER, SIGNED, UNSIGNED, STD_ULOGIC	STD_LOGIC_VECTOR	Paket *std_logic_arith* biblioteke *ieee*
TO_BIT(*x,u*)	STD_ULOGIC	BIT	Paket *std_logic_1164* biblioteke *ieee*
TO_BITVECTOR(*x,u*)	STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR	BIT_VECTOR	Paket *std_logic_1164* biblioteke *ieee*
TO_STDLOGICVECTOR(*x*)	BIT_VECTOR, STD_ULOGIC_VECTOR	STD_LOGIC_VECTOR	Paket *std_logic_1164* biblioteke *ieee*
TO_STDULOGICVECTOR(*x*)	BIT_VECTOR, STD_LOGIC_VECTOR	STD_ULOGIC_VECTOR	Paket *std_logic_1164* biblioteke *ieee*
TO_STDULOGIC(*x*)	BIT	STD_ULOGIC	Paket *std_logic_1164* biblioteke *ieee*

2.4 Operatori

U okviru ove sekcije će biti navedeni i opisani predefinisani operatori koji se koriste u VHDL jeziku. Predefinisani operatori se mogu podeliti u sledeće celine: operatori dodele, logički operatori, aritmetički operatori, relacioni operatori, pomerački operatori i operatori konkatanacije. Pored predefinisanih operatora, i sam korisnik može da kreira i definiše svoje operatore.

2.4.1. Operatori dodele

U tabeli 2.4.1.1. su prikazani operatori dodele:

Tabela 2.4.1.1. – Operatori dodele

Operator	Opis
<=	Dodela vrednosti signalu
:=	Dodela vrednosti varijabli, dodela vrednosti konstanti, dodela difolt vrednosti parametru u *generic* delu entiteta, dodela inicijalne vrednosti signalu/varijabli
=>	Dodela vrednosti delovima vektora (koristi se uglavnom u kombinaciji sa ključnom reči OTHERS), mapiranje signala na portove uvedene komponente

Detaljnija razlika između varijabli i signala će biti objašnjena u narednim sekcijama, ali ukratko signali predstavljaju linije (žice) u dizajnu koje povezuju komponente u unutrašnjosti entiteta. Na primer, signal može biti linija koja povezuje izlaz I kola sa ulazom nekog drugog I kola. Pomoću signala se u stvari opisuje hardverski dizajn, zajedno sa odgovarajućim primitivnim komponentama (I kolo, ILI kolo, flip-flop i dr.). Varijable s druge strane treba posmatrati kao pomoćne promenjive koje olakšavaju pisanje koda i kreiranje logičkih funkcija. Varijable nemaju direktan ekvivalent u hardveru kao signali. Pošto postoji očigledna razlika u prirodi signala i varijable, njihovi operatori dodele se razlikuju. Operator dodele := se još koristi i za postavljanje difolt/inicijalnih vrednosti signala, generičkih parametara i konstanti. Pošto je nekad zgodno dodeljivati vrednosti određenim delovima vektora, a za ostatak vektora postaviti zajedničku (istu) vrednost, uveden je operator => za takve situacije. Ovaj operator je veoma pogodan za dodelu iste vrednosti kompletnog vektora uz upotrebu ključne reči OTHERS (vidi primere) u slučaju kada je vektor parametrizovan jer se tada ne može postaviti konstantna vrednost pošto dužina te konstantne vrednosti mora da odgovara setovanju parametra dužine vektora. Isto važi i za veoma duge vektore jer je pisanje konstantne vrednosti sa velikim brojem bita zamorno. Ovaj operator se takođe koristi i za mapiranje komponenti u arhitekturu entiteta. Naime, da bi se olakšao razvoj složenih implementacija, tipično je lakše dizajnirati takav sistem tako što se podeli na nekoliko logičkih celina koje se zasebno realizuju kao entiteti. Realizovani entiteti tada čine komponente u globalnom entitetu koji obuhvata sve te realizovane manje celine. Pošto je uvezene komponente neophodno integrisati u arhitekturu globalnog entiteta, tada se pomoću operatora => vrši mapiranje portova komponenti na signale koji su definisani u arhitekturi globalnog entiteta i na taj način se ostvaruju veze između manjih celina (komponenti) u cilju dobijanja složene implementacije koja obavlja željenu složenu logičku funkciju. Broj nivoa hijerarhije je proizvoljan, odnosno i manji entiteti mogu u sebi sadržati komponente još manjih celina. Na taj način je projektovanje znatno olakšano jer se problem lako može podeliti na manje delove, a potom se realizovani delovi međusobno spajaju u cilju dobijanja složenijeg dizajna. Primeri upotrebe operatora dodele:

```
ENTITY primer IS
GENERIC
(
    N:INTEGER:=4--postavljanje difolt vrednosti parametra N
);
....
SIGNAL x : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL x2 : STD_LOGIC_VECTOR(0 TO 7);
VARIABLE y : INTEGER;
SIGNAL z : STD_LOGIC_VECTOR(3 DOWNTO 0):="1100";--postavljanje inicijalne vrednosti signala z
CONSTANT w : INTEGER:=7; --deklarisanje konstante w i postavljanje vrednosti konstante w
VARIABLE u : INTEGER:=0;-- postavljanje inicijalne vrednosti varijable u

x<= "01101011";--dodela vrednosti signalu (MSB bit je bit krajnje levo ('0'), LSB bit je bit krajnje desno ('1'))
x2<= "01101011";--dodela vrednosti signalu (MSB bit je bit krajnje desno ('1'), LSB bit je bit krajnje levo ('0'))
x<= (0=>'1', OTHERS=>'0');--dodela vrednosti '1' indeksu 0 vektora x, a ostatku vektora se dodeljuje '0' (moraju biti pokriveni svi
indeksi kada se koristi operator dodele =>)
x<= (OTHERS=>'0');--dodela '0' citavom vektoru (veoma prakticno u slucaju parametrizovanog vektora ili dugog vektora)
x<= (0=>'1', 3=>'1', OTHERS=>'0');-- dodela vrednosti '1' indeksima 0 i 3 vektora x, a ostatku vektora se dodeljuje '0'
y:= 15; --dodela vrednosti varijabli
-- primer za mapiranje portova uvezenih komponenti ce bit dat u sekciji koja opisuje komponente
```

2.4.2. Logički operatori

Logički operatori se koriste za logičke operacije (u suštini Bulova algebra), pri čemu tipovi koji mogu da se koriste za ove operacije su BOOLEAN, BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC i STD_ULOGIC_VECTOR (naravno, svi operandi i rezultat logičke operacije moraju biti istog tipa). U slučaju vektora, sve logičke operacije se izvršavaju u *bit-wise* maniru. U tabeli 2.4.2.1. su prikazani logički operatori:

Tabela 2.4.2.1. – Logički operatori

Operator	Opis
NOT	Negacija
AND	Logička I operacija
OR	Logička ILI operacija
NAND	Logička NI operacija
NOR	Logička NILI operacija
XOR	Logička ekskluzivno ILI operacija
XNOR	Inverzna XOR operacija

Operator NOT ima prioritet nad drugim operacijama. Primeri primene logičkih operatora:

```
SIGNAL x1, x2, x3, y1, y2, y3, y4, y5, y6, y7, y8, y9, y10, y11 : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL z1, y12 : STD_LOGIC_VECTOR(0 TO 3);
SIGNAL s1 : STD_LOGIC_VECTOR(1 DOWNTO 0);
....
x1<="0011";
x2<="0101";
x3<="0001";
y1<=x1 XOR x2;--rezultat ce biti "0110"
y2<=x1 XNOR x2;--rezultat ce biti "1001"
y3<=x1 OR x2;--rezultat ce biti "0111"
y4<= x1 OR NOT x2;--rezultat ce biti "1011", pošto se prvo izvršava NOT nad x2 pa tek onda OR
y5<= NOT x1 OR x2;--rezultat ce biti "1101", pošto se prvo izvršava NOT nad x1 pa tek onda OR
y6<= NOT x1 OR NOT x2;--rezultat ce biti "1110", pošto se prvo izvršava NOT nad x1 i nad x2 pa tek onda OR
y7<=x1 AND x2 AND x3;--može više operanada da se stavi u izraz (rezultat ce biti "0001")
y8<=x1 NOR x2 AND NOT x3;--GRESKA u slucaju da rezultat zavisi od redosleda izvrsavanja operacija, moraju se delovi izraza
odvajati u zagrade
y8<=(x1 NOR x2) AND NOT x3;--OK slozenija operacija (rezultat ce biti "1000")
y8<=x1 NOR (x2 AND NOT x3);--OK slozenija operacija (rezultat ce biti "1000")
y9<=x1 AND x2 AND x3;--OK jer rezultat ne zavisi od redosleda operacija
y9<=x1 OR x2 OR x3;-- OK jer rezultat ne zavisi od redosleda operacija (rezultat ce biti "0111")
y9<=x1 XOR x2 XOR x3;-- OK jer rezultat ne zavisi od redosleda operacija (rezultat ce biti "0111")
y9<=x1 XNOR x2 XNOR x3;-- OK jer rezultat ne zavisi od redosleda operacija (rezultat ce biti "0111")
y9<=x1 NAND x2 NAND x3;-- GRESKA jer rezultat zavisi od redosleda operacija, moraju se koristiti zagrade
y9<=x1 NOR x2 NOR x3;-- GRESKA jer rezultat zavisi od redosleda operacija, moraju se koristiti zagrade
y10<= NOT (x1 OR x2 OR x3);--pod NOT operator se može podvesti i složeniji izraz (rezultat ce biti "1000"), NOT se takode izvršava
po bit-wise principu
z1<="1011";
y11<=x1 AND z1;--rezultat ce biti "0011", zbog razlika u smeru vektora radi se na sledeci nacin: y11(3)<=x1(3) AND z1(0),
y11(2)<=x1(2) AND z1(1), y11(1)<=x1(1) AND z1(2), y11(0)<=x1(0) AND z1(1)
y12<=x1 AND z1;--rezultat ce biti "0011", zbog razlika u smeru vektora radi se na sledeci nacin: y12(0)<=x1(3) AND z1(0),
y12(1)<=x1(2) AND z1(1), y12(2)<=x1(1) AND z1(2), y12(3)<=x1(0) AND z1(1)
-- prethodna 2 primera pokazuju da nije dobra praksa mesati smerove vektora jer lako moze doci do konfuzije
x1<="1011";
s1<="01";
```

```
y1<=x1 AND s1;--rezultat ce biti "0001", kao da je raden AND izmedu "1011" i "0001", dimenzija rezultata mora biti jednaka dimenziji veceg operanda, nije dobra praksa raditi bitwise logicke operacije izmedju vektora razlicith duzina
```

Kao što se vidi iz datih primera, u slučaju složenijih operacija kod kojih rezultat zavisi od redosleda izvršavanja, neophodno je koristiti zagrade da bi se označio redosled operacija. Operator NOT u suštini igra ulogu inverznog bafera i tako se zaista i implementira u hardveru. Takođe, važno je uočiti da nije dobra praksa raditi sa vektorima različitih smerova, već se treba odlučiti za jedan smer i njega koristiti u svim vektorima. U suprotnom, lako može doći do grešaka u dizajnu koje bi bile posledica upotrebe različitih smerova u vektorima.

2.4.3. Aritmetički operatori

Aritmetički operatori se koriste za implementiranje osnovnih aritmetičkih funkcija. U tabeli 2.4.3.1. su prikazani aritmetički operatori:

Tabela 2.4.3.1. – Aritmetički operatori

Operator	Opis
+	Sabiranje
-	Oduzimanje
*	Množenje
/	Deljenje
**	Stepenovanje
MOD	Moduo
REM	Ostatak deljenja
ABS	Apsolutna vrednost

Tabela 2.4.3.2. – Tipovi operanada i rezultata operacije sabiranja/oduzimanja

Tip operanda	S	U	I	STDU	STDV	STD
S	S, STDV	S, STDV	S, STDV	S, STDV	X	X
U	S, STDV	U, STDV	U, STDV	U, STDV	X	X
I	S, STDV	U, STDV	I, STDV	X	STDV	X
STDU	S, STDV	U, STDV	X	X	X	X
STDV	X	X	STDV	X	STDV	STDV
STD	X	X	X	X	STDV	X

S - SIGNED, U - UNSIGNED, I - INTEGER, STDU - STD_ULOGIC, STDV - STD_LOGIC_VECTOR, STD - STD_LOGIC, X - nije dozvoljeno

Tipovi podataka koji mogu da se koriste u aritmetičkim operacijama su INTEGER, SIGNED, UNSIGNED i REAL, kao i STD_LOGIC_VECTOR ako se uključe odgovarajuće bibilioteke. Za SIGNED i UNSIGNED tipove mora da se koristi paket *std_logic_arith* biblioteke *ieee*, dok se za STD_LOGIC_VECTOR treba koristiti paket *std_logic_unsigned* ili *std_logic_signed* biblioteke *ieee*. Ne treba zaboraviti da se tip REAL ne može hardverski implementirati. Tabela 2.4.3.2 prikazuje koji tipovi međusobno mogu da se sabiraju/oduzimaju i

koji tipovi mogu da predstavljaju rezultat dotične operacije. Tabela 2.4.3.3. prikazuje koji tipovi međusobno mogu da se množe i koji tipovi predstavljaju rezultat dotične operacije.

Tabela 2.4.3.3. – Tipovi operanada i rezultata operacije množenja

Tip operanda	S	U	I	STDV
S	S, STDV	S, STDV	X	X
U	S, STDV	U, STDV	X	X
I	X	X	I	STDV
STDV	X	X	STDV	STDV

S - SIGNED, U - UNSIGNED, I - INTEGER, STDV - STD_LOGIC_VECTOR, X - nije dozvoljeno

Sabiranje, oduzimanje i množenje nemaju ograničenja što se tiče hardverske implementacije, s tim što treba imati na umu da operacija množenja troši veće hardverske resurse, naročito u slučaju ako treba podržati množenje dužih vektora. Iz tog razloga se često u FPGA čipovima ugrađuju *embedded* množači da bi se poboljšala podrška za obradu signala koja tipično koristi operacije množenja. Takođe, prilikom deklaracije signala treba imati u vidu da rezultat operacije množenja mora imati veći broj bita od operanada. Ostale operacije imaju veoma ograničenu podršku za hardversku implementaciju. Deljenje vektora je moguće samo u slučaju da je broj kojim se deli oblika stepena dvojke jer se tada operacija deljenja svodi na pomeranje udesno i tako se i realizuje (tj. ne koristi se operator deljenja), dok je međusobno deljenje INTEGER tipova dozvoljeno i kao rezultat se uzima ceo broj ispred decimalnog zareza. Stepenovanje je moguće samo ako su ili osnova ili stepen konstantni, pri čemu osnova ima dodatno ograničenje da mora biti stepena dvojke, ako je stepen promenjiva. Takođe, stepen mora biti pozitivan broj. Operator MOD se koristi za izvršavanje operacije moduo. Operator REM se koristi za računanje ostatka deljenja. Operatori MOD i REM se mogu koristiti samo sa INTEGER tipovima. ABS vraća apsolutnu vrednost, ali ova operacija se retko koristi. ABS je definisan za INTEGER tip, kao i za SIGNED i STD_LOGIC_VECTOR tipove, s tim što u slučaju SIGNED operanda, rezultat ABS operacije može biti ili SIGNED ili STD_LOGIC_VECTOR tip. U slučaju primene ABS operatora na STD_LOGIC_VECTOR tip, mora se koristiti paket *std_logic_signed* biblioteke *ieee* (što je i logično jer paket *std_logic_unsigned* je predviđen za rad sa nenegativnim veličinama).

Primeri primene aritmetičkih operatora:

```
SIGNAL x1, x2, y1 : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL x3, y2 : STD_LOGIC_VECTOR(0 TO 3);
SIGNAL x4 : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL y2 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL n1,n2,m1:INTEGER;
....
y1<=x1+x2;
y1<=x1+'1';
y1<=x1-x2;
y2<=x1*x2;
y1<=ABS(x1+x2);
m1<=n1/n2;
m1<=n1**3;
m1<=n1**(-3); GRESKA, stepen mora biti pozitivan broj
```

```
m1<=4**n1;
m1<=(-4)**n1;
m1<=5**n1;--GRESKA, 5 nije stepen dvojke
m1<=n1 MOD n2;
m1<=5 MOD 4;--rezultat je 1
m1<=(-5) MOD 4; --rezultat je 3
m1<=5 MOD (-4); --rezultat je -3
m1<=(-5) MOD (-4); --rezultat je -1
m1<=n1 REM n2;
m1<=5 REM 4;--rezultat je 1
m1<=(-5) REM 4; --rezultat je -1
m1<=5 REM (-4); --rezultat je 1
m1<=(-5) REM (-4); --rezultat je -1
x1<="0011";
x3<="0001";
y1<=x1+x3; --rezultat ce biti "0100", vazno je uociti da je MSB bit rezultata sa leve strane
y2<=x1+x3; -- rezultat ce biti "0100", vazno je uociti da je MSB bit rezultata sa desne strane
-- nema obrtanja pozicija vektora suprotnih smerova da bi se usaglasili, zato aritmeticke operacije nad vektorima suprotnih smerova
treba izbegavati
y1<=x1+x4; -- operandi mogu biti razlicitih dimezija, vazno je da rezultat bude dimenzije veceg operanda
```

2.4.4. Relacioni operatori

Relacioni operatori se koriste za komparaciju vrednosti signala (tipično u okviru postavljanja uslova) i kao rezultat vraćaju BOOLEAN tip. U tabeli 2.4.4.1. su prikazani relacioni operatori:

Tabela 2.4.4.1. – Relacioni operatori

Operator	Opis
=	Jednako
/=	Nejednako
<	Manje
>	Veće
<=	Manje ili jednako
>=	Veće ili jednako

Tipovi podataka koji mogu da se koriste u relacionim operacijama su svi oni koji mogu da se koriste u aritmetičkim ili logičkim operacijama: INTEGER, SIGNED, UNSIGNED, REAL, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR, BIT, BIT_VECTOR, BOOLEAN. Ne treba zaboraviti da se tip REAL ne može hardverski implementirati. Vrednosti koje se porede moraju biti istog tipa, ali ima izuzetaka. SIGNED, UNSIGNED i INTEGER tipovi se mogu međusobno porediti (sve kombinacije između ovih tipova su dozvoljene). STD_LOGIC_VECTOR se može porediti ili sa STD_LOGIC_VECTOR ili sa INTEGER tipom (za poređenje sa INTEGER tipom mora da bude uključen paket *std_logic_signed* ili *std_logic_unsigned* biblioteke *ieee*). Tabela 2.4.4.2 rezimira mogućnost komparacije signala različitog tipa:

Tabela 2.4.4.2. – Mogućnost komparacije između različitih tipova

Tip operanda	S	U	I	STDV
S	OK	OK	OK	X
U	OK	OK	OK	X
I	OK	OK	OK	OK
STDV	X	X	OK	OK

S - SIGNED, U - UNSIGNED, I - INTEGER, STDV - STD_LOGIC_VECTOR, X - nije dozvoljeno, OK - dozvoljeno

Primeri primene relacionih operatora:

```
SIGNAL x1, x2, x3 : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL x4 : STD_LOGIC_VECTOR(0 TO 3);
SIGNAL x5 : STD_LOGIC_VECTOR(2 DOWNTO 0);
SIGNAL y1 : BOOLEAN;
....
y1<=x1>x2;
y1<=x1/=x2;
IF (x1=x2)THEN...
IF( (x1<=x2)AND(x1>x3))THEN...
x1<="0011";
x4<="0001";
y1<=x1>x4; --rezultat ce biti TRUE jer ni ovde nema obrtanja smerova vektora, i ovo pokazuje da nije dobra praksa raditi sa vektorima suprotnih smerova
y1<=x1>x5; --operandi mogu biti razlicitih dimenzija
```

2.4.5. Operatori pomeranja

Operatori pomeranja se koriste za implementaciju pomeračkih registara. U tabeli 2.4.5.1. su prikazani operatori pomeranja:

Tabela 2.4.5.1. – Operatori pomeranja

Operator	Opis	Tip
SLL	Logičko pomeranje ulevo	BIT_VECTOR
SRL	Logičko pomeranje udesno	BIT_VECTOR
SLA	Aritmetičko pomeranje ulevo	BIT_VECTOR
SRA	Aritmetičko pomeranje udesno	BIT_VECTOR
ROL	Rotiranje ulevo	BIT_VECTOR
ROR	Rotiranje udesno	BIT_VECTOR
SHL	Pomeranje ulevo	STD_LOGIC_VECTOR, SIGNED, UNSIGNED
SHR	Pomeranje udesno	STD_LOGIC_VECTOR, SIGNED, UNSIGNED

Tipovi podataka koji mogu da se koriste u operacijama pomeranja su BIT_VECTOR, dok se za STD_LOGIC_VECTOR, SIGNED i UNSIGNED mora uključiti odgovarajuća biblioteka. Za SIGNED i UNSIGNED tipove mora da se koristi paket *std_logic_arith* biblioteke *ieee*, dok se za STD_LOGIC_VECTOR treba koristiti paket *std_logic_unsigned* ili *std_logic_signed* biblioteke *ieee*. Pri tome u slučaju BIT_VECTOR pomeranja izraz se piše u formatu *x* operator *y*, gde je *x* signal tipa BIT_VECTOR koji se pomera, a *y* INTEGER tip koji definiše veličinu pomeraja (za koliko mesta će se izvršiti pomeraj). U slučaju SIGNED, UNSIGNED i STD_LOGIC_VECTOR pomeranja izraz se piše u formatu operator(*x*, *y*), gde je *x* vektor koji se

pomera, a *y* veličina pomeraja. U slučaju STD_LOGIC_VECTOR tipa i operandi i rezultat moraju svi biti STD_LOGIC_VECTOR tipa. Dok u slučaju SIGNED i UNSIGNED tipa, pomereni vektor (*x*) može biti ili SIGNED ili UNSIGNED tipa, ali veličina pomeraja mora biti UNSIGNED tipa. Tip rezultata mora da se poklapa sa tipom operanda *x*.

Kao što vidimo iz tabele 2.4.5.1, u slučaju BIT_VECTOR tipa razlikujemo logičko i aritmetičko pomeranje ulevo, odnosno udesno. U slučaju logičkog pomeranja upražnjena mesta se popunjavaju nulama. U slučaju aritmetičkog pomeranja ulevo, na upražnjena mesta se upisuje vrednost krajnjeg bita sa desne strane. U slučaju aritmetičkog pomeranja udesno, na upražnjena mesta se upisuje vrednost bita sa krajnje leve strane.

U slučaju pomeranja ulevo/udesno vektora tipa STD_LOGIC_VECTOR, kada se koristi paket *std_logic_unsigned*, ili vektora tipa UNSIGNED, vrši se logičko pomeranje ulevo/udesno. U slučaju pomeranja ulevo/udesno vektora tipa STD_LOGIC_VECTOR, kada se koristi paket *std_logic_signed*, ili vektora tipa SIGNED, vrši se logičko pomeranje ulevo, i aritmetičko pomeranje udesno.

Primeri primene operatora pomeranja:

```
SIGNAL x1, y1 : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL x2 : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL x3: STD_LOGIC_VECTOR(0 TO 7);
SIGNAL z1, y2 : BIT_VECTOR(7 DOWNTO 0);
SIGNAL y3 : BIT_VECTOR(0 TO 7);
SIGNAL n : INTEGER;
....
y1<=SHL(x1, x2);
y1<=SHL(x1, "0010");
--ukljucen std_logic_unsigned paket (identicno ponasanje ima UNSIGNED tip)
y1<=SHL("10111011", "0010");--rezultat je "11101100"
y1<=SHR("10111011", "0010");--rezultat je "00101110"
--ukljucen std_logic_signed paket (identicno ponasanje ima SIGNED tip)
y1<=SHL("10111011", "0010");--rezultat je "11101100"
y1<=SHR("10111011", "0010");--rezultat je "11101110"
y1<=SHR("00111011", "0010");--rezultat je "00001110"
---------------------------------------------
x3<="11011101";
y1<=SHL(x3, "0010");-- rezultat je "01110100", nema obrtanja smerova vektora
y2<=z1 ROL 2;
y2<=z1 SLL n;
y2<="10111010" SLL 2; --rezultat je "11101000"
y3<="10111010" SLL 2; --rezultat je "11101000", ali je MSB sa desne strane
y2<="10111010" SRL 2; --rezultat je "00101110"
y2<="10111010" SLA 2; --rezultat je "11101000"
y2<="10111011" SLA 2; --rezultat je "11101111"
y2<="10111010" SRA 2; --rezultat je "11101110"
y2<="10111010" ROL 2; --rezultat je "11101010"
y2<="10111010" ROR 2; --rezultat je "10101110"
```

2.4.6. Operatori konkatanacije

Operatori konkatanacije se koriste za konkatanaciju signala u duži signal (praktično lepljenje signala). U tabeli 2.4.6.1. su prikazani operatori konkatanacije:

Tabela 2.4.6.1. – Operatori konkatanacije

Operator	Opis
&	Konkatanacija

Tabela 2.4.6.2. – Tipovi operanada i rezultata operacije konkatanacije

Tip operanda	S	U	SL	SUL	SV	SUV	B	BV
S	S	X	S	S	X	X	X	X
U	X	U	U	U	X	X	X	X
SL	S	U	U, S, SV, SUV	U, S, SV, SUV	SV	SUV	X	X
SUL	S	U	U, S, SV, SUV	U, S, SV, SUV	SV	SUV	X	X
SV	X	X	SV	SV	SV	X	X	X
SUV	X	X	SUV	SUV	X	SUV	X	X
B	X	X	X	X	X	X	BV	BV
BV	X	X	X	X	X	X	BV	BV

S - SIGNED, U - UNSIGNED, SL - STD_LOGIC, SUL - STD_ULOGIC, SV - STD_LOGIC_VECTOR, SUV - STD_ULOGIC_VECTOR, B - BIT, BV - BIT_VECTOR, X - nije dozvoljeno

Tipovi podataka koji mogu da se koriste u operaciji konkatanacije su BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, SIGNED, UNSIGNED, STD_ULOGIC, STD_ULOGIC_VECTOR. Tabela 2.4.6.2 daje pregled koji tipovi se mogu međusobno spajati i koji tip predstavlja rezultat konkatanacije. Iz tabele 2.4.6.2 se može lepo uočiti odnos tipova i podtipova, tj. nemogućnost saradnje vektora različitih tipova koji potiču od STD_LOGIC/STD_ULOGIC tipova, ali svi oni mogu da rade konkatanaciju sa STD_LOGIC/STD_ULOGIC tipovima. Takođe, posledica je i da rezultat konkatanacije između STD_LOGIC/STD_ULOGIC tipova može da bude bilo koji od izvedenih tipova (SIGNED, UNSIGNED, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR). Takođe, može se vršiti konkatanacija karaktera i stringova u rezultujući string, što može biti zgodno u simulaciji za ispis poruka.

Primeri primene operatora konkatanacije:

```
SIGNAL x1, x2: STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL x3: STD_LOGIC;
SIGNAL x4: STD_LOGIC_VECTOR(0 TO 3);
SIGNAL y1 : STD_LOGIC _VECTOR(7 DOWNTO 0);
SIGNAL y2 : STD_LOGIC _VECTOR(4 DOWNTO 0);
SIGNAL y3 : STD_LOGIC _VECTOR(8 DOWNTO 0);
SIGNAL y4 : STD_LOGIC _VECTOR(0 TO 7);
....
y1<=x1& x2;
y2<=x1& x3;
y3<= x1& x2& x3;
y3<= x1& x3& x2;
y3<= x1& '1'& x2;
y3<= "1010" & x3& x2;
y1<= "1010" & "0010";--rezultat je "10100010", MSB bit ima vrednost '1' (MSB je na levoj strani)
y1<= "1010" & '0' & "010";--rezultat je "10100010", MSB bit ima vrednost '1' (MSB je na levoj strani)
y4<= "1010" & "0010";--rezultat je "10100010", MSB bit ima vrednost '0' (MSB je na desnoj strani)
y4<= "1010" & '0' & "010";--rezultat je "10100010", MSB bit ima vrednost '0' (MSB je na desnoj strani)
-- MSB je najvisi indeks vektora, a LSB je najnizi indeks vektora
```

```
x1<="1011";
x4<="0111";
y1<=x1& x4;--rezultat je "10110111", nema obrtanja vektora da bi se usaglasili smerovi
```

2.5 Atributi

Atributi se koriste za opisivanje svojstava vektora (nizova), odnosno signala. U slučaju nizova, atributi se koriste za određivanje opsega indeksa niza, vrednosti najvišeg/najnižeg indeksa i dr. U slučaju signala, atributi se koriste za određivanje aktivnosti signala, poput promene vrednosti signala, pojave aktivne vrednosti, trajanja iste vrednosti signala i dr. U naredne dve podsekcije će biti opisani atributi koji se koriste za opisivanje nizova, odnosno atributi koji se koriste za oisivanje signala.

2.5.1. Atributi za opis vektora (nizova)

Tabela 2.5.1.1. – Atributi za opis vektora

Naziv atributa	Opis atributa	STD_LOGIC_VECTOR (7 DOWNTO 0)	STD_LOGIC_VECTOR (0 TO 4)	STD_LOGIC_VECTOR (5 DOWNTO 2)
LOW	Najniži indeks	0	0	2
HIGH	Najviši indeks	7	4	5
LEFT	Indeks krajnjeg levog člana	7	0	5
RIGHT	Indeks krajnjeg desnog člana	0	4	2
LENGTH	Dužina vektora	8	5	4
RANGE	Opseg indeksa	7 DOWNTO 0	0 TO 4	5 DOWNTO 2
REVERSE_RANGE	Obrnuti opseg indeksa	0 TO 7	4 DOWNTO 0	2 TO 5

Atributi za opis vektora se koriste za lako određivanje osnovnih osobina vektora poput dužine vektora, vrednosti najvišeg i najnižeg indeksa, vrednosti indeksa krajnjeg levog/desnog člana, opsega indeksa i obrnutog opsega indeksa. Razlog za uvođenje atributa je nejedinstven način definisanja vektora. Na primer, vektori se deklarišu u jednom od dva moguća smera - *big endian* ili *little endian*. Takođe, najniži indeks ne mora da bude 0, iako se u praksi koristi najčešće vrednost 0 kao najniži indeks. Naravno, čak i da se vektori uvek jedinstveno definišu i dalje bi dužina vektora bila parametar koji bi zavisio od deklaracije konkretne promenjive. Tabela 2.5.1.1. daje nazive atributa za opis vektora, kao i njihovo objašnjenje i konkretne primere. Iz tabele se vidi da se pristupanjem atributu vektora dobija odgovarajući parametar vektora u vidu INTEGER-a ili opsega INTEGER-a.

Pristupanje atributu se vrši tako što se iza naziva vektora stavlja apostrof iza kojega ide naziv atributa, na primer (*x* je vektor npr. BIT_VECTOR):

```
x'LOW
```

Najčešća upotreba atributa je u okviru petlji jer omogućavaju fleksibilno definisanje petlji. Umesto da se deklariše konkretan broj iteracija u petlji, može se iskoristiti atribut RANGE za definisanje i broja iteracija i vrednosti promenjive koja se koristi za indeks iteracije čime se

postiže veća fleksibilnost dizajna. Na sličan način se može iskoristiti kombinacija LOW i HIGH atributa.

```
FOR i IN RANGE (x'LOW TO x'HIGH) LOOP.... --(petlja se krece od najnižeg do najvišeg indeksa vektora)
FOR i IN RANGE (x'HIGH DOWNTO x'LOW) LOOP.... --(petlja se krece od najvišeg do najnižeg indeksa vektora)
FOR i IN x'RANGE LOOP.... --(petlja se krece kroz kompletan opseg vektora u opadajucem ili rastucem redosledu indeksa u zavisnosti kako je vektor definisan - kao big endian ili little endian)
FOR i IN RANGE (0 TO x'LENGTH) LOOP.... --(ovo je u redu ako je najniži indeks vektora 0)
```

Postoje i atributi za opisivanje enumerisanih tipova. Neki od tih atributa su identični već opisanim atributima poput LOW, HIGH, LEFT i RIGHT. Pošto se u praksi ne koriste često atributi za opisivanje enumerisanih tipova, ovi atributi neće biti razmatrani u okviru ovih skripti.

2.5.2. Atributi za opis signala

Atributi za opis signala se koriste za opis i praćenje stanja signala poput promene vrednosti signala, dužine stabilne vrednosti signala i sl. Lista atributa za opis signala je data u tabeli 2.5.2.1.

Tabela 2.5.2.1. – Atributi za opis signala

Naziv atributa	Opis atributa
EVENT	Vraća TRUE ako se dogodila promena vrednosti signala
STABLE	Vraća TRUE ako se nije dogodila promena vrednosti signala
ACTIVE	Vraća TRUE ako signal ima vrednost '1'
QUIET <vreme>	Vraća TRUE ako nije bilo promene vrednosti signala za specificirano vreme
LAST_EVENT	Vraća vreme proteklo od trenutka poslednje promene vrednosti signala
LAST_ACTIVE	Vraća vreme proteklo od trenutka poslednje aktivne vrednosti signala ('1')
LAST_VALUE	Vraća vrednost signala pre poslednjeg EVENT-a tj. promene vrednosti signala

Pored u tabeli navedenih atributa, postoji još nekoliko atributa, među kojima i atribut kojim se dobija informacija o tipu signala, ali pošto se oni ne koriste često neće biti opisani u okviru ovih skripti. Kao što se vidi iz opisa atributa u tabeli 2.5.2.1, većina atributa je namenjena upotrebi u okviru simulacije, i nije ih hardverski moguće implementirati, sem EVENT i STABLE atributa. Kod STABLE atributa se može specificirati i dužina intervala, slično kao kod QUIET atributa. EVENT atribut se veoma često koristi i to u sprezi sa takt signalom za određivanje nailaska uzlazne ili silazne ivice, što određuje novi ciklus u sekvencijalnoj logici. Pristupanje vrednosti atributa za opis signala se radi na identičan način kao kod pristupa vrednosti atributa za opis vektora - iza naziva signala se stavlja apostrof iza kojega se navodi atribut kojem želimo da pristupimo. Veoma česte konstrukcije koje se koriste u okviru sekvencijalne logike za označavanje uzlazne ivice takta su:

```
IF (clk'EVENT AND clk='1')THEN... -- definisanje uzlazne ivice takta (clk je cesto korišcen naziv za signal takta)
IF (clk'EVENT AND clk='0')THEN... -- definisanje silazne ivice takta
IF (NOT(clk'STABLE) AND clk='1')THEN... -- drugi nacin definisanja uzlazne ivice takta
IF (NOT(clk'STABLE) AND clk='0')THEN... -- drugi nacin definisanja silazne ivice takta
WAIT UNTIL(clk'EVENT AND clk='1')... -- treci nacin definisanja uzlazne ivice takta
WAIT UNTIL(clk'EVENT AND clk='0')... -- treci nacin definisanja silazne ivice takta
IF RISING_EDGE(clk) THEN... -- cetvrti nacin definisanja uzlazne ivice takta (ovaj metod koristi funkciju, a ne atribut signala)
IF FALLING_EDGE(clk) THEN... -- cetvrti nacin definisanja silazne ivice takta (ovaj metod koristi funkciju, a ne atribut signala)
```

2.6 Konkurentni (paralelni) kod

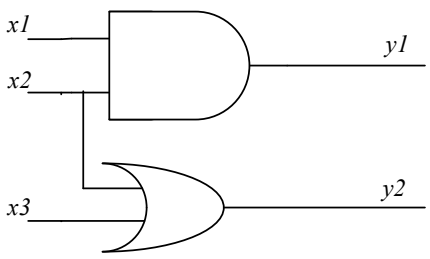
U okviru ove sekcije počinje opis konstrukcija koje se koriste za opisivanje hardverskih celina unutar arhitekture entiteta. Postoje dva načina koja se koriste za opis unutrašnje strukture (arhitekture) entiteta - konkurentni i sekvencijalni način, pri čemu se oba načina mogu koristiti uporedo unutar iste arhitekture. Svi delovi opisani u arhitekturi entiteta, bez obzira na koji način su opisani - konkurentni ili sekvencijalni način, su međusobno konkurentni u smislu da kad se implementiraju u hardveru oni će istovremeno postojati i izvršavati svoje funkcije.

Konkurentni kod podrazumeva da se svi izrazi implementiraju i izvršavaju paralelno tj. istovremeno. Otuda kod ovog koda nije dozvoljeno istom signalu dva (ili više) puta izvršiti dodelu vrednosti. Konkurentni kod je prirodan za opisivanje hardvera jer uzima u obzir činjenicu da se svi izrazi (koji predstavljaju hardverske delove) izvršavaju istovremeno, što i jeste poenta hardvera tj. svi hardverski delovi istovremeno postoje i izvršavaju svoje funkcije. Konkurentni kod se koristi za opisivanje kombinacione logike. Kao što ćemo videti u okviru ove sekcije, postoji i način da se konkurentnim kodom opiše i sekvencijalna logika, ali treba izbegavati takav pristup.

S druge strane sekvencijalni kod podrazumeva da se izrazi sekvencijalno izvršavaju, slično kao kod proceduralnih programskih jezika poput C jezika. Ovaj način je često prirodan za kreiranje neke logičke funkcije koju hardver treba da obavi. Međutim, iako je očigledna suštinska razlika između konkurentnog koda i sekvencijalnog koda, hardver koji se kreira na osnovu sekvencijalnog koda će takođe raditi paralelno kao i hardver koji se implementira na osnovu konkurentnog koda. Naravno, za razliku od konkurentnog koda, u slučaju sekvencijalnog koda ne mora doći do direktnog preslikavanja izraza u odgovarajući hardver. U stvari, kompajler će protumačiti sekvencijalni kod, utvrditi koje funkcije sve obavlja (preciznije odnose između signala) i tek potom će da uradi formiranje hardvera koji je u stvari opisan sekvencijalnim kodom. Ovo će biti razjašnjeno čitaocu u sledećoj sekciji primerima sekvencijalnog koda koji će biti obrađen u sledećoj sekciji. Sekvencijalni kod se koristi za opis sekvencijalne logike, a može da se koristi i za opis kombinacione logike.

2.6.1. Upotreba operatora u konkurentnom kodu

Svi operatori navedeni u sekciji 2.4 se mogu koristiti za formiranje konkurentnog koda. Konkurentni kod se tada sastoji od niza izraza u kojima se koriste operatori navedeni u sekciji 2.4. Kao primer uzmimo kolo prikazano na slici 2.6.1.1 koje sadrži samo osnovne logičke funkcije I i ILI.

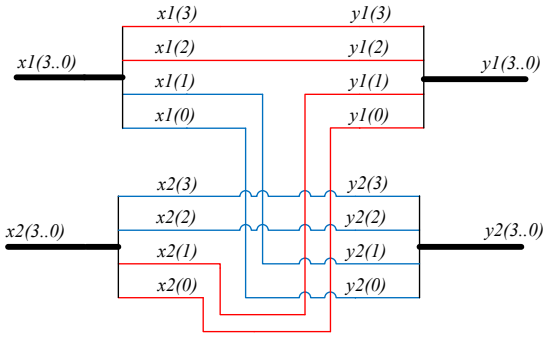


Slika 2.6.1.1. – Kolo koje sadrži jedno I i jedno ILI kolo

Pomoću logičkih operatora možemo u potpunosti opisati ovo kolo sledećim konkurentnim kodom:

```
y1<=x1 AND x2;
y2<=x2 OR x3;
```

U slučaju da želimo da realizujemo kombinacionu funkciju koja meša dva ulazna vektora dužine 4, tako što spaja gornju polovinu jednog vektora sa donjom polovinom drugog vektora, i obrnuto (slika 2.6.1.2), tada možemo da koristimo samo operator dodele, ili možemo da koristimo i operator konkatanacije.



Slika 2.6.1.2. – Mešanje vektora

```
--------------------samo operator dodele--------------------
y1(3 DOWNTO 2)<=x1(3 DOWNTO 2);
y1(1 DOWNTO 0)<=x2(1 DOWNTO 0);
y2(3 DOWNTO 2)<=x2(3 DOWNTO 2);
y2(1 DOWNTO 0)<=x1(1 DOWNTO 0);
--------------------operator konkatanacije--------------------
y1<=x1(3 DOWNTO 2) & x2(1 DOWNTO 0);
y2<=x2(3 DOWNTO 2) & x1(1 DOWNTO 0);
```

Kao što vidimo, čak i u slučaju veoma jednostavnih funkcija, njihova realizacija tj. opisivanje se može uraditi na više načina. U gornjem primeru oba slučaja su u potpunosti ravnopravna tj. njihova hardverska realizacija će biti istovetna slici 2.6.1.2. Ovo je veoma dobra osobina VHDL jezika jer omogućava višestruke načine kodiranja iste funkcije, čime programer može lako da nađe svoj stil pisanja VHDL koda, a da pri tome ne utiče negativno na performanse konačne realizacije.

2.6.2. WHEN konstrukcija

WHEN konstrukcija omogućava selekciju izraza koji želimo da aktiviramo, pri čemu selekciju određujemo odgovarajućim uslovima ili vrednošću signala selektora. Postoje dve WHEN konstrukcije na raspolaganju: WHEN/ELSE konstrukcija i WITH/SELECT/WHEN konstrukcija.

Struktura WHEN/ELSE konstrukcije je sledeća:

```
rezultat <= vrednost_1 WHEN uslov_1 ELSE
            vrednost_2 WHEN uslov_2 ELSE
            ....
            vrednost_n-1 WHEN uslov_n-1 ELSE
            vrednost_n;
```

Struktura WITH/SELECT/WHEN konstrukcije je sledeća:

```
WITH selektor SELECT
        rezultat <= vrednost_1 WHEN vrednost_selektora_1,
                    vrednost_2 WHEN vrednost_selektora_2,
                    ....
                    vrednost_n WHEN vrednost_selektora_n;
```

U slučaju WITH/SELECT/WHEN konstrukcije moraju biti pokrivene sve vrednosti selektora. Iz tog razloga se često koristi ključna reč OTHERS da pokrije sve preostale situacije, koje prethodno nisu pokrivene (to je naročito zgodno u slučaju STD_LOGIC baziranih signala i vektora jer oni imaju i druge vrednosti pored '0' i '1'):

```
vrednost_n WHEN OTHERS;
```

Takođe, kada ne želimo da izvršimo dodelu vrednosti, možemo iskoristiti ključnu reč UNAFFECTED, kojom signaliziramo da ne želimo da izvršimo dodelu vrednosti, već želimo da zadržimo trenutnu vrednost rezultujućeg signala. Treba imati na umu da to znači da realizovanoj logici dodajemo memoriju u vidu leča. Na primer, često za situacije koje nisu pokrivene željenim vrednostima selektora i ne želimo da menjamo vrednost rezultujućeg signala:

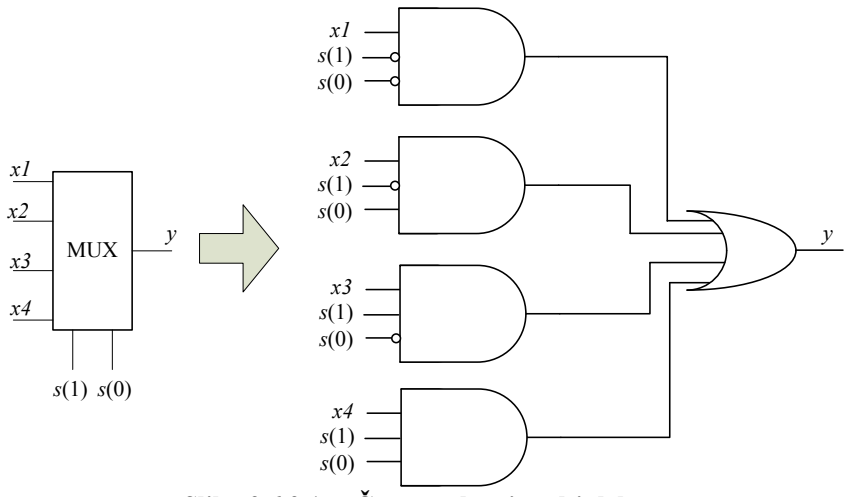
```
UNAFFECTED WHEN OTHERS;
```

Kod navođenja vrednosti selektora u WITH/SELECT/WHEN konstrukciji može se navesti više vrednosti odjednom u obliku opsega ili skupa vrednosti, ako, naravno, dotične vrednosti daju isti rezultat:

```
WITH broj SELECT--broj i rezultat su deklarisani kao INTEGER
rezultat <= 0 WHEN 0 TO 5, --navodenje u vidu opsega (moze i za vektore, ali treba izbegavati tu varijantu)
        1 WHEN 6|7|9|10, --navodenje u vidu više vrednosti (skup vrednosti)
        ....
        UNAFFECTED WHEN OTHERS;
```

Logičko kolo prikazano na slici 2.6.1.1 možemo opisati i primenom dve WHEN/ELSE konstrukcije:

```
y1 <= '1' WHEN ((x1='1')AND(x2='1')) ELSE
        '0';
y2 <= '0' WHEN ((x2='0')AND(x3='0')) ELSE
        '1';
-------Alternativna WHEN/ELSE konstrukcija za y1------------
y1 <= '0' WHEN ((x1='0')OR(x2='0')) ELSE
        '1';
-------Alternativna WHEN/ELSE konstrukcija za y2------------
y2 <= '1' WHEN ((x2='1')OR(x3='1')) ELSE
        '0';
```



Slika 2.6.2.1. – Četvoroulazni multiplekser

Razmotrimo četvoroulazni multiplekser prikazan na slici 2.6.2.1. Mutliplekser možemo opisati pomoću operatora:

```
y<=(NOT s(1) AND NOT s(0) AND x1)OR(NOT s(1) AND s(0) AND x2)OR(s(1) AND NOT s(0) AND x3)OR(s(1) AND s(0) AND x4);
```

Multiplekser možemo takođe opisati i sa WHEN/ELSE konstrukcijom:

```
y <= x1 WHEN (s="00") ELSE
        x2 WHEN (s="01") ELSE
        x3 WHEN (s="10") ELSE
        x4;
```

Multiplekser možemo takođe opisati i sa WITH/SELECT/WHEN konstrukcijom koja možda i najprirodnije opisuje funkciju multipleksera.

```
WITH s SELECT
y <= x1 WHEN "00",
        x2 WHEN "01",
        x3 WHEN "10",
        --x4 WHEN "11"; --ovo sme ako je s deklarisan kao BIT_VECTOR(1 DOWNTO 0)
        x4 WHEN OTHERS; --ovo mora ako s nije deklarisan kao BIT_VECTOR(1 DOWNTO 0), vec kao STD_LOGIC_VECTOR ili slican tip
```

Primer multipleksera dodatno potvrđuje činjenicu da VHDL omogućava raznoliko kodiranje iste funkcije. Takođe, treba primetiti još jednu važnu osobinu VHDL koda. Naime, opisivanjem multipleksera preko operatora, praktično je opisana struktura multipleksera preko najprostijih delova: I, ILI i NE funkcija (kola). S druge strane, preko WHEN konstrukcija je opisano ponašanje funkcije multipleksera bez zalaženja u njegovu strukturu. To je takođe veoma bitna osobina VHDL jezika, koja omogućava da vršimo programiranje (dizajniranje) hardverske celine tako što opisujemo njeno ponašanje, a potom kompajler vrši prevođenje tog opisa ponašanja u stvarnu hardversku realizaciju koja odgovara strukturnom opisu dotične hardverske celine. To značajno pojednostavljuje projektovanje složenih logičkih funkcija jer je dovoljno da opišemo njihovo ponašanje, a ne složenu strukturu takvih funkcija.

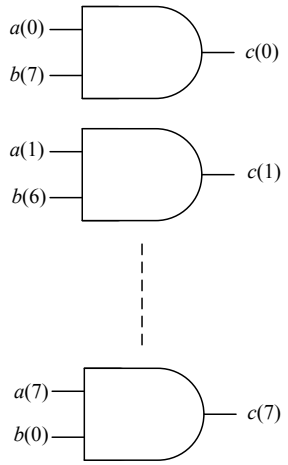
2.6.3. GENERATE konstrukcija

Razlikujemo dve GENERATE konstrukcije - FOR/GENERATE i IF/GENERATE. FOR/GENERATE konstrukcija se koristi za formiranje FOR petlje kojom se generiše kod koji se ponavlja, da bi se izbeglo višestruko pisanje identičnog koda. IF/GENERATE konstrukcija se koristi za generisanje koda ako je ispunjen određeni uslov. Tipično se uslov definiše koristeći GENERIC parametre čime se preko GENERIC dela entiteta, utiče na sastav unutrašnjosti entiteta. U slučaju IF/GENERATE konstrukcije nije dozvoljen ELSE deo. Takođe, GENERATE konstrukcije mogu da se gnezde, na primer unutar FOR/GENERATE konstrukcije se može naći druga FOR/GENERATE konstrukcija ili IF/GENERATE konstrukcija, i obrnuto. Uglavnom nije dobra praksa imati više od jednog gneždenja GENERATE konstrukcija.

Struktura FOR/GENERATE konstrukcije je sledeća:

```
labela: FOR iterativna_promenjiva IN opseg GENERATE
        konkurentni kod
END GENERATE;
```

Važno je napomenuti da opseg mora biti statički definisan, odnosno ne može biti dinamički definisan tako što bi se u toku izvršavanja logičke funkcije koja uključuje ovu konstrukciju opseg dinamički podešavao. Razlog je prost. VHDL kod opisuje hardver koji se implementira. Konkurentni kod opisuje hardversku implementaciju, a FOR/GENERATE konstrukcija je zadužena za replikaciju identičnog hardvera u više kopija, međutim, broj kopija mora biti unapred određen tj. jednom kad se te kopije postave u hardver njihov broj ne može više da se menja. Kao primer FOR/GENERATE konstrukcije uzmimo I funkciju između dva vektora dužine osam prikazane na slici 2.6.3.1.



Slika 2.6.3.1. – Primer I funkcije između vektora suprotnih smerova

Jedan način za pisanje ove funkcije je direktno pisanje pomoću operatora:

```
c(0)<=a(0) AND b(7);
c(1)<=a(1) AND b(6);
.....
c(7)<=a(7) AND b(0);
```

Kao što vidimo imamo identične izraze koji se razlikuju samo u indeksima, i sa slike 2.6.3.1 se vidi da su u pitanju identična dvoulazna I kola koja se javljaju osam puta. Stoga možemo da formiramo FOR/GENERATE konstrukciju koja će formirati replike dvoulaznih I kola (odnosno gore navedenih izraza kada se koristi samo operator) kroz iteracije:

```
I_kola: FOR i IN 0 to 7 GENERATE
        c(i)<=a(i) AND b(7-i);
END GENERATE;
```

Struktura IF/GENERATE konstrukcije je sledeća:

```
labela: IF uslov GENERATE
        konkurentni kod
END GENERATE;
```

Kao primer za IF/GENERATE možemo uzeti sledeći slučaj. Na primer želimo da imamo izbor između strukture prikazane na slici 2.6.3.1 i strukture koja bi sadržala ILI kola umesto I kola. Tada u GENERIC parametre entiteta možemo staviti parametar *izbor* koji bi bio tipa BIT, pa bi *izbor* mogao imati vrednosti '0' ili '1'. Sa '0' bi birali varijantu sa I kolima, a sa '1' varijantu sa ILI kolima. Tada bi IF/GENERATE konstrukcija bila:

```
if_labela1: IF (izbor='0') GENERATE
        I_kola: FOR i IN 0 to 7 GENERATE
                c(i)<=a(i) AND b(7-i);
        END GENERATE;
END GENERATE;
if_labela2: IF (izbor='1') GENERATE
        ILI_kola: FOR i IN 0 to 7 GENERATE
                c(i)<=a(i) OR b(7-i);
        END GENERATE;
END GENERATE;
```

Pošto se ne može koristiti ELSE u IF/GENERATE konstrukciji, morali smo koristiti dve IF/GENERATE konstrukcije. Takođe, ovaj primer pokazuje gneždenje FOR/GENERATE konstrukcije u IF/GENERATE konstrukciji. Unutar GENERATE konstrukcija se mogu staviti i procesi. Procesi, kao što ćemo videti u sekciji 2.7, sadrže sekvencijalni kod, ali posmatrani kao celina se tretiraju kao konkurentni kod prema ostatku koda arhitekture.

2.6.4. BLOCK konstrukcija

Razlikujemo dve BLOCK konstrukcije - jednostavnu i zaštićenu blok konstrukciju. Jednostavna blok konstrukcija se koristi za grupisanje koda u blokove radi bolje preglednosti koda. Zaštićena blok konstrukcija sadrži tzv. zaštitni uslov, a unutar zaštićenog bloka postoje nezaštićeni i zaštićeni delovi konkurentnog koda. Nezaštićeni delovi konkurentnog koda se uvek izvršavaju (u prevodu direktno se implementiraju), dok se zaštićeni deo koda izvršava samo kad je ispunjen zaštićen uslov (u prevodu ovaj deo koda se ne implementira direktno, već u kombinaciji sa zaštićenim uslovom). Zaštićena blok konstrukcija se može iskoristiti za kreiranje sekvencijalnog koda, ali takav pristup treba izbegavati. Dozvoljeno je gneždenje blok konstrukcija unutar drugih blok konstrukcija.

Struktura jednostavne BLOCK konstrukcije je sledeća:

```
labela: BLOCK
        deklarativni deo
BEGIN
        konkurentni kod
END BLOCK labela;
```

Možemo videti da postoji i deklarativni deo u BLOCK konstrukciji, pa imajući to u vidu možemo reći i da je BLOCK konstrukcija svojevrsna podarhitektura. Kao primer možemo uzeti

WHEN/ELSE varijantu realizacije četvoroulaznog multipleksera i staviti taj kod u jedan BLOCK:

```
blok_mux: BLOCK
BEGIN
    y <= x1 WHEN s="00" ELSE
                x2 WHEN s="01" ELSE
                x3 WHEN s="10" ELSE
                x4;
END BLOCK blok_mux;
```

Na ovaj način smo izdvojili kod koji definiše jedan multiplekser u jedan blok. Na sličan način možemo druge celine u arhitekturi entiteta izdvojiti u blokove čime postižemo pregledniji kod jer odmah možemo videti gde se šta nalazi. Naravno, veoma sličan efekat smo mogli postići i ubacivanjem komentara u kod. Uglavnom se BLOCK konstrukcija ne koristi često, ali opet sve zavisi od stila pisanja VHDL programera. Unutar BLOCK (običnih i zaštićenih) konstrukcija se mogu staviti i procesi. Procesi, kao što ćemo videti u sekciji 2.7, sadrže sekvencijalni kod, ali posmatrani kao celina se tretiraju kao konkurentni kod prema ostatku koda arhitekture.

Struktura zaštićene (GUARDED) BLOCK konstrukcije je sledeća:

```
labela: BLOCK (zaštićeni uslov)
        deklarativni deo
BEGIN
        konkurentni kod koji sadrži zaštićene i nezaštićene delove
END BLOCK labela;
```

Kod zaštićenih delova se mora koristiti ključna reč GUARDED. Na primer, ako želimo realizovati D-FF pomoću zaštićene blok konstrukcije, kod bi izgledao:

```
dff: BLOCK ((clk'EVENT)AND(clk='1'))
BEGIN
        q<=GUARDED d;
END BLOCK dff;
```

U slučaju da želimo da dodamo i sinhroni reset flip-flopu kod bi izgledao:

```
dff_reset: BLOCK ((clk'EVENT)AND(clk='1'))
BEGIN
        q<=GUARDED d WHEN reset='0' ELSE
        '0';
END BLOCK dff_reset;
```

Kao što smo videli iz prethodna dva primera, konkurentnim kodom se može kreirati i sekvencijalna logika, ali još jednom da ponovimo, takav pristup ipak treba izbegavati, jer sekvencijalni kod koji se obrađuje u sledećoj sekciji predstavlja prirodan način za opisivanje sekvencijalne logike.

2.7 Sekvencijalni kod

Sekvencijalni kod za razliku od konkurentnog koda se izvršava sekvencijalno, redosledom kako su izrazi navedeni. Međutim, ne sme se zaboraviti da se sekvencijalni kod takođe implementira u vidu hardverskih celina koje istovremeno postoje i izvršavaju svoje funkcije, kao u slučaju konkurentnog koda. Sekvencijalni kod opisuje ponašanje logičkog kola koje treba da obavlja neku logičku funkciju, a često je lakše opisati ponašanje logičkog kola nego

samu strukturu logičkog kola. Sekvencijalni kod se može koristiti za opisivanje i sekvencijalne i kombinacione logike. Postoje tri celine koje sadrže sekvencijalni kod: PROCESS, FUNCTION i PROCEDURE. Unutar svake od ovih celina se mogu koristiti sledeće konstrukcije: operatori, IF, WAIT, CASE i LOOP. Konstrukcije pomoću operatora su identične onima u konkurentnom kodu, uz bitnu razliku da se izrazi tumače sekvencijalno, a ne paralelno kao u konkurentnom kodu. U konkurentnom kodu se mogu koristiti samo signali (SIGNAL), dok se u sekvencijalnom kodu pored signala mogu koristiti i varijable (VARIABLE), pa će prvo biti objašnjena razlika između signala i varijabli.

2.7.1. Signali i varijable

Signali predstavljaju realne linije (žice) u dizajniranom hardveru koje spajaju delove hardvera (tj. komponente), na primer linija koja povezuje izlaz jednog I kola i ulaz drugog I kola bi predstavljala jedan signal. Signali se koriste i u konkurentnom kodu i u sekvencijalnom kodu. Mogu se deklarisati u paketima (PACKAGE), entitetu (ENTITY) i deklarativnom delu arhitekture (ARCHITECTURE). U entitetu signali u stvari predstavljaju portove, a već smo videli kako se deklarišu portovi entiteta. Deklaracija signala u paketu i arhitekturi se radi na sledeći način:

```
SIGNAL naziv_signala : tip_signala:=inicijalna_vrednost_signala;
```

Postavljanje inicijalne vrednosti signala prilikom deklaracije signala je opciono (odnosno deo *:=inicijalna_vrednost_signala* može da se izostavi). Opciono u tipu signala može da se navede opseg kao, na primer, u slučaju INTEGER tipa. Primeri deklarisanja signala:

```
SIGNAL broj : INTEGER RANGE 0 TO 31:=0;
SIGNAL vektor : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL bitsignal : STD_LOGIC:='0';
```

Signali su globalne promenjive, tako da u slučaju deklarisanja u paketu, signal je globalan za sve entitete koji koriste dotičan paket, u slučaju deklarisanja u entitetu signal je globalan za sve arhitekture dotičnog entiteta, i u slučaju deklarisanja u deklarativnom delu arhitekture, signal je globalan u dotičnoj ahitekturi. Entitet mora da sadrži portove (sem u slučaju entiteta koji se koristi za testiranje i koji se neće implementirati, što ćemo videti u budućim poglavljima) tako da se u entitetu sigurno deklarišu signali. S druge strane, češće je deklarisanje signala u arhitekturi, nego u paketu, jer u slučaju deklarisanja u paketu mora da se vodi računa da ne dođe do greške u radu sa signalom ako se on koristi za više entiteta, što može da bude veoma komplikovano.

Posebna vrsta signala su konstante (CONSTANT) koje se deklarišu u paketu ili deklarativnom delu arhitekture, pri čemu se deklarisanje konstante podjednako vrši u paketu i deklarativnom delu arhitekture, tj. nema razlike kao u slučaju signala. Konstanta se može deklarisati i u entitetu, ali to se uglavnom nikad ne radi. Deklaracija konstante u paketu i arhitekturi se radi na sledeći način:

```
CONSTANT naziv_konstante : tip_konstante:= vrednost_konstante;
```

Prilikom deklaracije konstante mora da se navede i vrednost konstante. Primeri definisanja konstante:

```
CONSTANT konst_vektor : STD_LOGIC_VECTOR(7 DOWNTO 0):= "00001111";
CONSTANT konst_broj : INTEGER:=1;
```

U slučaju upotrebe signala unutar procesa, promena njegove vrednosti nije trenutna, nego se obavlja tek nakon svih izraza u procesu tj. nakon što se dođe do kraja procesa. Na primer, neka signal *x* tipa INTEGER ima vrednost 3, a signal *y* tipa INTEGER ima vrednost 5. Ako napišemo unutar sekvencijalnog koda sledeća dva izraza:

```
x <=7;
y <=x;
```

Tada će signal *y* da dobije vrednost 3, a signal *x* vrednost 7, i to tek na kraju procesa. To znači da je signal *y* dobio staru vrednost signala *x* bez obzira na dodelu nove vrednosti signalu *x*, iz prostog razloga što će i *x* i *y* dobiti novu vrednost tek na kraju procesa. Isto važi i za funkcije i procedure pošto i one predstavljaju sekvencijalni kod (tako da ćemo u ostatku ove sekcije koristiti samo proces kao primer). Ova osobina predstavlja značajnu razliku u odnosu na proceduralne jezike poput C-a, jer tamo je promenjiva odmah dobijala novu vrednost i dalje se radilo sa tom novom vrednošću. Ovo svojstvo signala u sekvencijalnom kodu često zna da zbuni početnike tako da je to jedna od pojedinosti na koje treba obratiti pažnju. Takođe, veoma važna napomena - ako se u arhitekturi koristi više procesa, tada nije dozvoljena dodela istom signalu unutar dva (ili više) različita procesa, jer to dovodi do konflikta koju dodelu treba uzeti kao konačnu. Kao što se iz primera moglo videti, operator dodele za signal je <=.

Varijabla predstavlja lokalni signal koji se definiše u deklarativnom delu procesa, funkcije ili procedure, i varijabla je vidljiva samo unutar tela procesa, funkcije, procedure u čijem deklarativnom delu je deklarisana. Deklarisanje varijable se radi na sličan način kao deklarisanje signala samo uz upotrebu ključne reči VARIABLE:

```
VARIABLE naziv_varijable : tip_varijable:=inicijalna_vrednost_varijable;
```

Postavljanje inicijalne vrednosti varijable prilikom deklaracije varijable je opciono (odnosno deo :=*inicijalna_vrednost_varijable* može da se izostavi), ali treba imati u vidu da se postavljanje inicijalne vrednosti varijable koristi samo u simulaciji, i ne može hardverski da se implementira (kompajler će ignorisati ovu vrednost). Razlog za tu situaciju je što varijable nemaju svoj ekvivalent u linijama (žicama), kao signali. Opciono, u tipu varijable može da se navede opseg kao na primer u slučaju INTEGER tipa. Primeri deklarisanja varijable:

```
VARIABLE broj : INTEGER RANGE 0 TO 31:=0;
VARIABLE vektor : STD_LOGIC_VECTOR(7 DOWNTO 0);
VARIABLE bitvarijabla : STD_LOGIC:='0';
```

Varijable predstavljaju ekvivalent promenjivih iz proceduralnih jezika poput C-a. Naime, vrednosti dodeljene varijabli odmah postaju aktivne. Uzmimo sada dve modifikovane varijante prethodnog primera, pri čemu treba primetiti da je operator dodele za varijablu :=.

```
----------Varijanta 1----------
x :=7;
y <=x;
----------Varijanta 2----------
x :=7;
y :=x;
```

U varijanti 1, *x* je deklarisan kao varijabla, a *y* kao signal. Varijabla *x* odmah dobija vrednost 7, dok će *y* dobiti takođe vrednost 7, ali na kraju procesa (tj. ako se negde do kraja procesa radi sa signalom *y* radi se sa njegovom starom vrednošću). U varijanti 2 i *x* i *y* su deklarisani kao varijable. To znači da će i *x* i *y* imati vrednost 7, i ako se kasnije do kraja procesa još negde koriste *x* i *y* onda se radi sa njihovim novim vrednostima, a to je 7. Ovo svojstvo

varijabli često zna da zavede početnike da koriste pretežno varijable, ali takav princip treba izbegavati jer varijable za razliku od signala nemaju direktan ekvivalent u linijama (žicama). Zbog svoje prirode da trenutno menjaju vrednost one u stvari predstavljaju ekvivalent kombinacionoj logici, pa se može desiti da se preteranom upotrebom varijabli dobije rešenje lošijih performansi zbog prevelike količine nepotrebne kombinacione logike. Varijable treba koristiti samo tamo gde zaista moraju da se koriste, tj. kada je veoma teško opisati logičku funkciju upotrebom samo signala. Kao primer možemo uzeti deo koda koje definiše logičko kolo koje treba da prebroji jedinice unutar vektora:

```
----------Varijanta 1 GRESKA-----------
suma <=0;--suma je deklarisana kao signal
FOR i IN x'RANGE LOOP
  IF(x(i)= '1') THEN
     suma<=suma+1;
  END IF;
END LOOP;
broj_jedinica<=suma;
----------Varijanta 2 OK-----------
suma :=0; --suma je deklarisana kao varijabla
FOR i IN x'RANGE LOOP
  IF(x(i)= '1') THEN
     suma:=suma+1;
  END IF;
END LOOP;
broj_jedinica<=suma;
```

U datim primerima se koriste konstrukcije sekvencijalnog koda koje će kasnije biti detaljnije objašnjene, ali prilično je jasno šta se dešava u napisanom kodu. U varijanti 1 se koristi signal *suma* na nepravilan način. Naime, u petlji se želi postići inkrementiranje ovog signala za svaki član vektora koji je jednak '1'. Međutim, s obzirom da je *suma* deklarisana kao signal, *suma* će imati ili vrednost 0 ako je vektor sačinjen od nula, ili staru vrednost inkrementiranu za jedan ako postoji jedna ili više jedinica u vektoru. Kao što znamo, signal dobija vrednost tek na kraju procesa. Prva dodela (vrednost 0) signalu *suma* je na početku pre petlje i to je trenutno poslednja dodela signalu *suma*. U samoj petlji kad se naiđe na jedinicu u vektoru vrši se nova dodela vrednosti signalu *suma*, i nova vrednost, koja će biti dodeljena signalu *suma* na kraju procesa, je za jedan inkrementirana stara vrednost signala *suma* (sa kojom je ušao u proces). Svaki put kad se naiđe na jedinicu će biti izvršena ova dodela. Otuda, na kraju procesa će signal *suma* imati ili vrednost 0 ili staru vrednost inkrementiranu za 1. Očigledno, varijanta 1 ne radi kako treba. Važno je primetiti da za razliku od konkurentnog koda, u sekvencijalnom kodu su dozvoljene višestruke dodele nove vrednosti istom signalu, a poslednje izvršena dodela u procesu predstavlja novu vrednost signala koju će on dobiti na kraju procesa. Naravno, višestruka dodela je moguća samo ako se vrši unutar istog procesa/funkcije/procedure. U drugoj varijanti, *suma* je deklarisana kao varijabla i to znači da sve dodele vrednosti postaju odmah aktivne. Tako, varijabla *suma* ulazi u petlju sa vrednošću 0. Na svaku jedinicu u vektoru se inkrementira vrednost varijable *suma* i ta vrednost postaje odmah aktivna i sa njom se ulazi u novu iteraciju petlje. Na kraju, kao rezultat u signal *broj_jedinica* ispravno ispisujemo broj jedinica u vektoru koji je predstavljen vrednošću varijable *suma*. Nova vrednost signala *broj_jedinica* će naravno biti aktivna na kraju procesa. Na ovaj način smo ilustrovali kako treba upotrebljavati varijable, jer napisati kod koji obavlja funkciju brojanja jedinica u vektoru nije nimalo jednostavno samo pomoću signala.

2.7.2. Proces

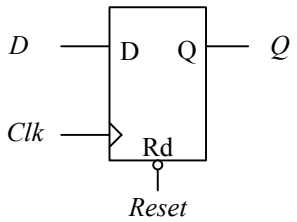
Proces (PROCESS) predstavlja svojevrstan ekvivalent BLOCK konstrukciji u konkurentnom kodu. U telu arhitekture nekog entiteta, sekvencijalni kod mora biti grupisan unutar procesa. Pri tome, u okviru jedne arhitekture može biti definisano i više procesa, pri čemu je jedino važno voditi računa da se jednom signalu ili portu entiteta ne sme dodeljivati vrednost u različitim procesima. Sintaksa procesa je:

```
labela: PROCESS (lista osetljivosti)
        deklarativni deo
BEGIN
        sekvencijalni kod
END PROCESS labela;
```

Labela procesa je opciona i ne mora da se koristi. U deklarativnom delu procesa se deklarišu varijable koje se koriste unutar procesa i one se ne mogu koristiti tj. nisu vidljive van procesa. Lista osetljivosti predstavlja okidač na koji se telo procesa (sekvencijalni kod) počinje izvršavati. To znači da svaki put kad se promeni vrednost signala iz liste osetljivosti krenuće izvršenje sekvencijalnog koda. Ovako tumačeći, proces deluje kao svojevrsna procedura iz programskog jezika C koja bi se pokretala svaki put kad neki signal iz liste osteljivosti promeni svoju vrednost. Međutim, treba uvek imati na umu da VHDL služi za opisivanje hardverske strukture, pri čemu pruža fleksibilnost programeru da opiše samu strukturu hardvera ili da opiše ponašanje hardvera (naravno uvek se u dizajnu mogu kombinovati oba pristupa). Upravo lista osetljivosti pruža priliku VHDL programeru da ne vodi računa o svim detaljima strukture svih delova dizajna nego da se koncentriše samo na ponašanje kola tj. opis funkcije koju kolo treba da obavi. Posmatrajmo veoma jednostavan primer, običan D-FF koji ima i mogućnost asinhronog reseta. Kreirajmo proces koji opisuje funkciju takvog flip-flopa. Treba primetiti da labela procesa nije korišćena u primeru.

```
PROCESS (reset,clk)
BEGIN
        IF(reset='1')THEN
                Q<='0';
        ELSIF(clk'EVENT AND clk='1')THEN
                Q<=D;
        END IF;
END PROCESS;
```

Opis ovakvog D-FF je sledeći: kada se dogodi promena vrednosti signala *reset* ili *clk* izvršiti sledeće operacije: ako *reset* ima vrednost '1' na izlaz *Q* izbaciti vrednost '0', a ako *reset* nema vrednost '1', tada ako se desila uzlazna ivica takta prosledi na izlaz *Q* vrednost sa ulaza *D*, u suprotnom ništa ne raditi. Na osnovu ovog opisa formira se flip-flop prikazan na slici 2.7.2.1.



Slika 2.7.2.1. – Struktura D-FF iz primera

Kao što se vidi, opis flip-flopa iz primera se preslikava u realan hardver, u ovom slučaju konkretan flip-flop. Treba napomenuti da se lista osetljivosti može izostaviti ili se može navesti

samo deo liste osetljivosti. U tim slučajevima će kompajler izbaciti upozorenje da su pojedini signali izostavljeni iz liste osetljivosti, ali će kompajliranje da se korektno odradi do kraja (naravno pod uslovom da nema drugih grešaka). U slučaju Xilinx-ovog ISE razvojnog okruženja je ipak poželjno navoditi listu osetljivosti jer u određenim situacijama kada se kreira kombinaciona logika sa većim brojem hijerarhijskih nivoa može doći do grešaka u konačnom rezultatu kompajlera i kao rezultat se može dobiti neispravan dizajn.

Pokažimo i primer D-FF sa sinhronim resetom. U tom slučaju opis takvog flip-flopa je:

```
PROCESS (clk)
BEGIN
        IF(clk'EVENT AND clk='1')THEN
                IF(reset='1')THEN
                        Q<='0';
                ELSE
                        Q<=D;
                END IF;
        END IF;
END PROCESS;
```

Primetimo da lista osetljivosti sada sadrži samo signal takta jer se sve akcije (promene) sada dešavaju samo na uzlaznu ivicu signala takta, odnosno sve promene su sinhrone. Napomenimo još da u slučaju kada proces implementira kombinacionu logiku, onda svi ulazni signali te kombinacione logike treba da se navedu u listi osetljivosti. Pošto najveći broj sekvencijalnih kola radi sinhrono i tipično ima opciju reseta, ova dva primera sa D-FF daju prikaz kako se kreira sinhrona sekvencijalna logika sa asinhronim, odnosno sinhronim resetom. Asinhroni reset očigledno ima najviši prioritet i zato ide kao prvi IF uslov, dok sinhroni reset mora da radi po diktatu sinhronog signala (takta) i zato mora da se podvuče pod IF uslov koji opisuje uzlaznu ivicu takta. U najčešćem slučaju, u akcijama koje se obavljaju na reset se vrši inicijalizacija svih delova sekvencijalnog kola na inicijalne (početne) vrednosti. S obzirom da smo u ovoj sekciji videli jednu VHDL definiciju sinhronog signala (takta), zgodno je sada napomenuti da u gotovo svim programabilnim čipovima nije dozvoljeno definisati logiku koja radi na obe ivice takta, jer kao što smo videli u poglavlju 1, flip-flopovi u FPGA programabilnim čipovima se mogu konfigurisati da rade samo na uzlaznu ili samo na silaznu ivicu takta. Otuda se sledeća konstrukcija ne može implementirati u hardveru iako je sintaksno i funkcionalno tačna:

```
IF(clk'EVENT AND clk='1')THEN--uzlazna ivica
        brojac<= brojac +1;
ELSIF(clk'EVENT AND clk='0')THEN--silazna ivica
        brojac <= brojac +1;
END IF;
```

Prikazana konstrukcija pokušava da implementira logiku koja bi vršila brojanje ivica takta, tako što bi inkrementirala vrednost brojača na svaku ivicu takta. Međutim, pošto u većini programabilnih čipova nema flip-flopova koji mogu da rade na obe ivice takta istovremeno, ova logika kod takvih čipova ne može da se implementira u hardveru. Naravno, unutar jedne arhitekture je moguće definisati procese od kojih bi neki bili aktivni na uzlaznu ivicu takta, a drugi na silaznu ivicu takta. U tom slučaju deo dizajnirane logike će raditi na uzlaznu ivicu takta, a drugi deo dizajnirane logike će raditi na silaznu ivicu takta. Međutim, treba biti oprezan u takvim situacijama jer je maksimalno dozvoljeno fizičko kašnjenje između signala koji rade na uzlaznu ivicu i signala koji rade na silaznu ivicu takta dvostruko manje od maksimalnog kašnjenja u slučaju kada cela logika radi samo na jedan tip ivice takta (kašnjenje od pola periode takta naspram kašnjenja od cele periode takta). Napomenimo još da u okviru istog procesa može

da se istovremeno (uporedo) implementira i sekvencijalna i kombinaciona logika. Tada se u listi osetljivosti trebaju navesti svi ulazni signali kombinacione logike, signal takta i signali asinhronog reseta i dozvole za takt (ako postoje).

2.7.2.1. Operatori

Svi operatori navedeni u sekciji 2.4 se mogu koristiti za formiranje sekvencijalnog koda unutar procesa. Sekvencijalni kod se tada sastoji od niza izraza u kojima se koriste operatori navedeni u sekciji 2.4. Kao primer iskoristimo kolo sa slike 2.6.1.1 (sa jednim I i jednim ILI kolom). Prikažimo i konkurentni i sekvencijalni kod ovog kola:

```
--Konkurentni kod--
y1<=x1 AND x2;
y2<=x2 OR x3;
--Sekvencijalni kod--
PROCESS (x1,x2,x3)
BEGIN
        y1<=x1 AND x2;
        y2<=x2 OR x3;
END PROCESS;
```

U slučaju sekvencijalnog koda opis kola mora da se stavi unutar procesa. U okviru liste osetljivosti moraju da se navedu sva tri ulazna signala (*x1*, *x2*, *x3*) jer promena bilo kog od ovih signala može dovesti do promene nekog od izlaznih signala. Kao što vidimo, kod unutar procesa i konkurentni kod su identični, odnosno upotreba operatora je gotovo ista. Suštinska razlika je u prirodi konkurentnog i sekvencijalnog koda. Naime, u konkurentnom kodu svi izrazi se tumače istovremeno, a kod sekvencijalnog koda se tumače sekvencijalno pa su čak dozvoljene i višestruke dodele istom signalu jer se kao nova vrednost uzima poslednja dodela pre dostizanja kraja procesa. Primer modifikovanog sekvencijalnog koda koje i dalje opisuje isto kolo:

```
--Modifikovani sekvencijalni kod--
PROCESS (x1,x2,x3)
BEGIN
        y1<=x1 OR x3;--ovo je redudantna linija jer je sledeca linija ponistava
        y1<=x1 AND x2;
        y2<=x2 OR x3;
END PROCESS;
```

Dodatna linija u modifikovanom kodu nema nikakav efekat. Postoje dve dodele signalu *y1*, ali pošto se obe bezuslovno izvršavaju uvek će biti izvršena druga dodela, tako da prva dodela neće ni biti implementirana u hardveru.

Pokažimo i da se unutar jedne arhitekture mogu mešati konkurentni i sekvencijalni kod, tako što ćemo opisati kolo iz primera sa mešavinom konkurentnog i sekvencijalnog koda:

```
--Mesavina konkurentnog i sekvencijalnog koda--
y1<=x1 AND x2;
PROCESS (x2,x3)
BEGIN
        y2<=x2 OR x3;
END PROCESS;
```

Za kraj pokažimo još da se unutar jedne arhitekture može formirati više procesa koristeći još uvek isto kolo iz primera.

```
--Proces 1--
PROCESS (x1,x2)
BEGIN
        y1<=x1 AND x2;
END PROCESS;
--Proces 2--
PROCESS (x2,x3)
BEGIN
        y2<=x2 OR x3;
END PROCESS;
```

2.7.2.2. IF konstrukcija

IF konstrukcija je veoma popularna u sekvencijalnom kodu, što smo mogli i videti u primeru opisa D-FF u sekciji 2.7.2, gde se koristila za opis uslova uzlazne ivice signala takta i opis uslova asinhronog/sinhronog reseta. Sintaksa IF konstrukcije je:

```
IF (uslov1) THEN
        sekvencijalni kod;
ELSIF (uslov2) THEN
        sekvencijalni kod;
ELSIF (uslov3) THEN
        sekvencijalni kod;
.........
ELSE
        sekvencijalni kod;
END IF;
```

U IF konstrukciji ELSIF i ELSE deo su opcioni. Npr. može da postoji samo ELSE odmah iza IF, ili mogu da postoje samo jedan ili više ELSIF uslova, a da na kraju nema i ELSE uslova. Takođe, može postojati i samo IF, bez ELSE i ELSIF delova.

Kao primer IF konstrukcije možemo uzeti jedno dvoulazno I kolo:

```
PROCESS (x1,x2)
BEGIN
        IF((x1='1')AND(x2='1'))THEN
                y<='1';
        ELSE
                y<='0';
        END IF;
END PROCESS;
```

Drugi primer IF konstrukcije je kreiranje funkcije multipleksera 4 u 1:

```
PROCESS (x1,x2,x3,x4,sel)
BEGIN
        IF(sel="00")THEN
                y<=x1;
        ELSIF(sel="01")THEN
                y<=x2;
        ELSIF(sel="10")THEN
                y<=x3;
        ELSE
                y<=x4;
        END IF;
END PROCESS;
```

Zbog sličnosti IF konstrukcije sa IF konstrukcijama drugih programskih jezika, IF konstrukcija se ponekad koristi neoptimalno, jer preterano korišćenje ELSIF opcija može da

kreira dug niz logičkih kola na jednom putu, međutim, više reči o ovoj temi će biti u sledećoj sekciji 2.7.2.3.

2.7.2.3. CASE konstrukcija

CASE konstrukcija predstavlja ekvivalent WITH/SELECT/WHEN konstrukciji. Sintaksa CASE konstrukcije je:

```
CASE (promenjiva) IS
        WHEN (vrednost1) => sekvencijalni kod;
        WHEN (vrednost2) => sekvencijalni kod;
        ........
END CASE;
```

Sekvencijalni kod se izvršava samo kod WHEN dela koji sadrži vrednost koja je jednaka trenutnoj vrednosti *promenjiva*. Odnosno, ponašanje nije identično CASE konstrukciji u C jeziku, što navodimo radi izbegavanja potencijalne zabune ako je čitalac navikao da programira u C jeziku.

I ovde kao kod WITH/SELECT/WHEN konstrukcije moraju biti pokrivene sve vrednosti promenjive na osnovu koje se formira CASE, tako da je i ovde zgodna upotreba ključne reči OTHERS za pokrivanje svih preostalih uslova:

```
WHEN OTHERS => sekvencijalni kod;
```

Takođe, ako u nekoj selekciji ne treba ništa izvršiti tada se koristi ključna reč NULL koja je ekvivalent ključnoj reči UNAFFECTED iz konkurentnog koda.

```
WHEN (vrednostx) => NULL;
```

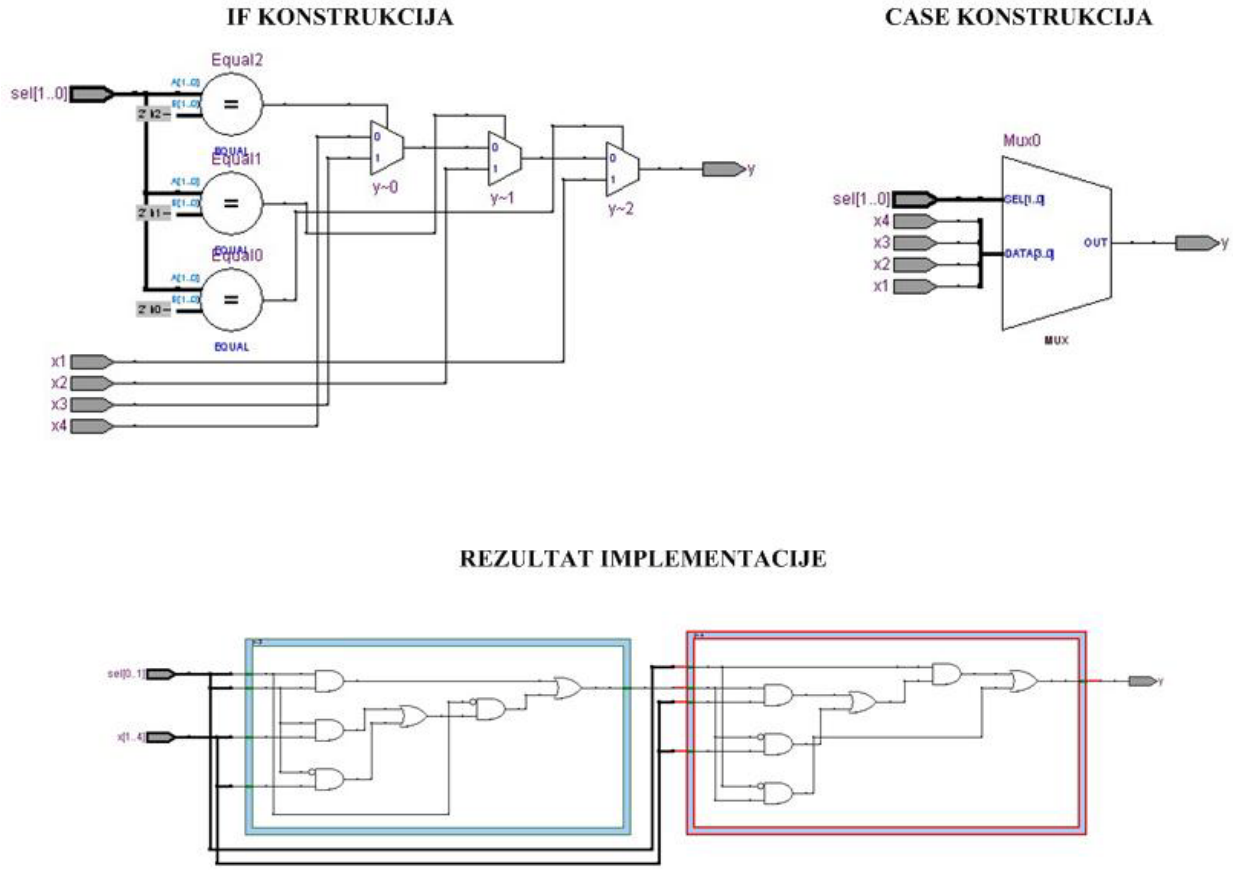
Isto kao kod WITH/SELECT/WHEN konstrukcije, moguće je odjednom pokriti više vrednosti ili preko opsega vrednosti ili navođenjem skupa vrednosti:

```
CASE (broj) IS --broj je deklarisan kao INTEGER
        WHEN 0|4|5 => sekvencijalni kod;-- navodjenje skupa vrednosti
        WHEN 1 TO 3 => sekvencijalni kod;-- navodjenje opsega vrednosti (nije dozvoljeno za vektore)
        WHEN OTHERS => sekvencijalni kod;
END CASE;
```

Kreirajmo multiplekser 4 u 1 pomoću CASE konstrukcije:

```
CASE (sel) IS
        WHEN "00"=> y<=x1;
        WHEN "01"=> y<=x2;
        WHEN "10"=> y<=x3;
        WHEN OTHERS => y<=x4;
END CASE;
```

Funkcionalno, multiplekser kreiran pomoću CASE konstrukcije i multiplekser kreiran pomoću IF konstrukcije (iz prethodne sekcije) su identični. Međutim, strukturno, oni su različiti. CASE konstrukcija sve uslove tretira ravnopravno tj. ravno, a IF konstrukcija sa svojim ELSIF delovima uslove tretira po prioritetima tj. hijerarhijski, pri čemu redolsed navođenja uslova određuje redosled prioriteta.



Slika 2.7.2.3.1. – Struktura multipleksera u IF i CASE varijanti

Na slici 2.7.2.3.1 je u gornjoj polovini slike prikazana logička struktura multipleksera u obe varijante (CASE i IF), a u donjoj polovini slike je prikazana ostvarena implementacija na čipu Cyclone II familije kompanije Altera. Kao što vidimo, CASE varijanta ima bolju logičku strukturu sa manjim kašnjenjem, dok IF varijanta koristi više multipleksera u cilju ostvarivanja prioriteta i očigledno IF varijanta ima složeniju logičku strukturu sa većim kašnjenjem. Današnji kompajleri bi uglavnom trebali da se izbore sa ovom razlikom pa da čak i u slučaju IF varijante, kreiraju strukturu koja odgovara CASE varijanti da bi se dobilo kvalitetnije rešenje. Tako je rezultat realne implementacije multipleksera na čipu Cyclone II familije identičan u oba slučaja i prikazan je u donjoj polovini slike 2.7.2.3.1. Međutim, i sam dizajner bi ipak morao da vodi računa o ovoj razlici i da izabere varijantu koja bolje odgovara njegovim potrebama - uglavnom je CASE varijanta sa svojom ravnom strukturom bolja opcija. Tako da u dizajnu uvek treba proveriti da li se kreirana IF konstrukcija može zameniti boljom CASE varijantom. Napomenimo samo da sličan odnos u konkurentnom kodu imaju WHEN/ELSE i WITH/SELECT/WHEN konstrukcije, čiji su ekvivalenti u sekvencijalnom kodu IF i CASE konstrukcije, respektivno.

2.7.2.4. LOOP konstrukcija

LOOP konstrukcija predstavlja petlju i zgodna je za upotrebu kod delova koda koji se ponavljaju, a mogu da se kompaktnije napišu u vidu petlje. Razlikujemo FOR/LOOP i WHILE/LOOP varijante. FOR/LOOP varijanta se koristi kada je broj ciklusa petlje konstantan. WHILE/LOOP petlja se koristi kada broj ciklusa petlje nije konstantan, već zavisi od postavljenog uslova, tačnije petlja se izvršava sve dok se postavljeni uslov ne naruši. Očigledno,

FOR/LOOP petlja je zgodna za kompaktnije pisanje delova koda koji se ponavljaju (naravno ako je moguće te delove smestiti u petlju), dok sa WHILE/LOOP petljom treba biti oprezan jer njeno nedeterminističko ponašanje obično podrazumeva i složeniji hardver u konačnoj implementaciji.

Sintaksa FOR/LOOP varijante je:

```
labela: FOR iterativna_promenjiva IN opseg LOOP
    sekvencijalni kod
END LOOP labela;
```

Labela u FOR/LOOP petlji je opciona i može da se izostavi. Takođe je važno naglasiti da opseg mora da bude statički definisan. Primer primene FOR/LOOP petlje je već dat u sekciji 2.7.1.

Sintaksa WHILE/LOOP varijante je:

```
labela: WHILE uslov LOOP
    sekvencijalni kod
END LOOP labela;
```

Labela u WHILE/LOOP petlji je opciona i može da se izostavi. Pošto petlja mora da se završi u jednom prolazu procesa, uslov za izlazak iz petlje treba da bude baziran na varijabli, jer kao što znamo signal menja svoju vrednost tek na kraju procesa. Otuda bi uslovi bazirani na signalima potencijalno mogli da dovedu do zaglavljivanja u petlji, što, u stvari, rezultira dizajnom koji ne može da se implementira u hardveru. Primer jedne WHILE/LOOP petlje koja opisuje osam dvoulaznih I kola:

```
i:=0;
WHILE (i<8) LOOP
        c(i)<=a(i) AND b(i);
        i:=i+1;
END LOOP;
```

WHILE/LOOP konstrukcija treba da se koristi samo kada je zaista neophodna i kada ne postoji drugi način da se opiše ponašanje logike.

Za nasilan izlazak iz petlje se koristi ključna reč EXIT. Po izvršavanju ove komande automatski se izlazi iz petlje. Ova komanda se može iskoristiti za kreiranje koda koji će da izvrši funkciju određivanja jedinice na najnižoj poziciji tj. indeksu:

```
pozicija <=8;
FOR i IN 0 TO 7 LOOP---ako je vektor duzine 8
  IF(x(i)= '1') THEN
     pozicija<=i;
     EXIT;--nasilan izlazak iz petlje kad smo nasli najnizu jedinicu
  END IF;
END LOOP;
```

Za preskakanje preostalih izraza do kraja iteracije koristi se ključna reč NEXT. NEXT je ekvivalent komande CONTINUE iz programskog jezika C. Kao primer uzmimo sledeći kod:

```
FOR i IN 0 TO 7 LOOP---ako je vektor duzine 8
  NEXT WHEN (x(i)= '0');
  ---alternativno pisanje NEXT komande----
  --- IF(x(i)= '0') THEN
  ---       NEXT;
  --- END IF;
   c(i) <= x(i) XOR y(i);
   d(i) <= x(i) OR y(i);
```

```
END LOOP;
```

Prikazani kod izvršava jednu XOR i jednu OR operaciju samo na onim pozicijama vektora na kojima vektor *x* ima logičku '1', u suprotnom te dve operacije se preskaču.

Obe ključne reči i EXIT i NEXT se mogu pisati samostalno:

```
EXIT;
NEXT;
```

ili uz upotrebu WHEN uslova:

```
EXIT WHEN uslov;
NEXT WHEN uslov;
```

2.7.2.5. WAIT konstrukcija

WAIT konstrukcija je specifična sekvencijalna konstrukcija, gde se izvršava čekanje dok se određeni uslov ne ispuni. U slučaju da se koristi WAIT konstrukcija, tada lista osetljivosti procesa mora biti prazna. Postoje tri varijante WAIT konstrukcije: WAIT UNTIL, WAIT ON i WAIT FOR.

WAIT UNTIL konstrukcija se koristi za čekanje dok se ne ispuni određeni uslov, pri čemu je uslov određen vrednošću jednog signala. Sintaksa WAIT UNTIL konstrukcije je:

```
WAIT UNTIL vrednost signala;
```

WAIT UNTIL konstrukcija je pogodna za formiranje sinhrone logike koja radi na uzlaznu ili silaznu ivicu takta. Ova konstrukcija mora ići na početak procesa tj. mora biti prva linija procesa.

```
PROCESS
BEGIN
        WAIT UNTIL (clk'EVENT AND clk='1');
        IF(reset='1')THEN
                .....(1)
        ELSIF(clk'EVENT AND clk='1')THEN
                .....(2)
        END IF;
END PROCESS;
```

WAIT UNTIL konstrukcija u datom primeru sinhrone logike obezbeđuje da se na početku procesa čeka uzlazna ivica takta, i tek kad ona naiđe ide se dalje u proces. Tada se ispituje signal reseta i ako je on aktivan izvršava se deo koda pod (1), u suprotnom se izvršava deo koda pod (2). Treba primetiti da je zbog WAIT UNTIL konstrukcije u pitanju sinhroni reset. Takođe, treba primetiti da je proces bez definisane liste osetljivosti, čime se postiže da se odmah po završetku procesa ponovo ulazi u njega i odmah nailazi na WAIT UNTIL konstrukciju kojom se obezbeđuje čekanje na uzlaznu ivicu takta.

WAIT ON konstrukcija je malo fleksibilnija konstrukcija jer dozvoljava reakciju (čekanje) na više signala, tako da se može koristiti i za formiranje sinhrone i asinhrone logike. Isto kao i WAIT UNTIL konstrukcija, i WAIT ON konstrukcija treba da ide na početak procesa. Sintaksa WAIT ON konstrukcije je:

```
WAIT ON signal1, signal2, ..., signalN;
```

Primer logike koja radi na signal takta, ali koja ima asinhroni reset je:

```
PROCESS
BEGIN
        WAIT ON reset, clk;
        IF(reset='1')THEN
                .....(1)
        ELSIF(clk'EVENT AND clk='1')THEN
                .....(2)
        END IF;
END PROCESS;
```

WAIT ON konstrukcija u datom primeru obezbeđuje da se na početku procesa čeka promena signala reseta ili takta, i tek kada se dogodi takva promena ide se dalje u proces. Tada se ispituje signal reseta i ako je on aktivan izvršava se deo koda pod (1), u suprotnom ispituje se da li je došla uzlazna ivica takta i ako jeste izvršava se deo koda pod (2). U ovom primeru imamo asinhroni reset. Takođe, treba primetiti da je proces bez definisane liste osetljivosti, čime se postiže da se odmah po završetku procesa ponovo ulazi u njega i odmah nailazi na WAIT ON konstrukciju kojom se obezbeđuje čekanje na promenu signala navedenih u WAIT ON konstrukciji. Praktično lista signala navedena u WAIT ON konstrukciji je ekvivalent liste osetljivosti procesa. WAIT UNTIL i WAIT ON konstrukcije u suštini zamenjuju funkciju liste osetljivosti procesa. Koju varijantu će programer koristiti, listu osetljivosti procesa ili WAIT konstrukciju zavisi samo od stila programera, pošto su obe varijante ekvivalentne i nema razlike u konačnoj implementaciji. Napomenimo da WAIT ON konstrukcija nije podržana u nekim razvojnim okruženjima. Takođe, s obzirom da se sinhrona logika veoma jednostavno implementira upotrebom IF konstrukcije, primena WAIT ON i WAIT UNTIL konstrukcija i nije od velikog značaja, ali opet to zavisi od stila pisanja programera.

WAIT FOR konstrukcija se ne može implementirati u hardveru (za razliku od prve dve navedene WAIT konstrukcije) i koristi se samo za potrebe simulacije. WAIT FOR konstrukcija izaziva čekanje za specificirano vreme. Sintaksa WAIT FOR konstrukcije je:

```
WAIT FOR vreme;
```

Na primer, ako želimo čekanje od 5ns:

```
WAIT FOR 5ns;
```

WAIT FOR konstrukcija je zgodna za potrebe testiranja jer pomoću nje može da se definiše redosled događaja u željenim vremenskim intervalima u okviru simulacije. Na primer, ako želimo da signal *x* ima vrednost '1' u trajanju od 5ns, pa potom vrednost '0' u trajanju od 10ns pre nego što se ponovo vrati na vrednost '1':

```
x<='1';
WAIT FOR 5ns;
x<='0';
WAIT FOR 10ns;
x<='1';
```

2.7.3. Funkcije i procedure

Funkcije (FUNCTION) i procedure (PROCEDURE) takođe predstavljaju celine u okviru kojih se piše sekvencijalni kod. Međutim, postoji bitna razlika u odnosu na proces. Proces se piše unutar tela arhitekture, dok se funkcije i procedure stavljaju u deklarativni deo arhitekture ili u pakete. Proces u stvari igra ulogu glavnog koda, a funkcije i procedure igraju ulogu potprograma. U funkcije i procedure se stavljaju opisi onih konstrukcija koje se koriste na više mesta u dizajnu ili koje će se koristiti i u drugim projektima. Takođe se mogu koristiti i za dobijanje

kompaktnijeg i preglednijeg koda u telu arhitekture. Ako se funkcija/procedura koristi samo u arhitekturi jednog entiteta, onda nema potrebe kreirati paket, već se dotična funkcija/procedura može deklarisati i u deklarativnom delu te arhitekture. U ovom slučaju se funkcija/procedura koristi za dobijanje kompaktnijeg, preglednijeg i jasnijeg koda u telu arhitekture. Ako će se funkcija/procedura koristiti u više različitih entiteta onda je bolje kreirati jedan paket kog će svi entiteti potom koristiti. Ako nameravamo da funkciju/proceduru koriste i drugi dizajneri onda je takođe bolje deklarisati je u paketu. Kao proces, i funkcije i procedure sadrže sekvencijalni kod i mogu upotrebljavati sve konstrukcije kao i proces (operatori, IF, CASE, LOOP), sem WAIT konstrukcija. WAIT konstrukcije se ne mogu koristiti u funkcijama i procedurama.

2.7.3.1. Funkcije

Da bi se kreirala funkcija neophodna je njena deklaracija koja definiše ulazne argumente i izlazni argument, i sadrži opis tj. telo funkcije. Sintaksa deklaracije funkcije je:

```
FUNCTION naziv_funkcije (lista ulaznih argumenata) RETURN izlazni_argument IS
        deklarativni deo
BEGIN
        sekvencijalni kod
END naziv_funkcije;
```

Funkcija može imati samo jedan izlazni argument, a nula, jedan ili više ulaznih argumenata. Slučaj sa nula ulaznih argumenata nema veliki praktičan značaj. Ulazni argumenti funkcije mogu biti konstante i signali, dok varijable ne mogu da se koriste kao ulazni argumenti. Ulazni argument se navodi u vidu ključne reči koja definiše tip argumenta - signal ili konstanta (SIGNAL, CONSTANT), zatim sledi naziv argumenta iza koga sledi tip podatka argumenta. Naziv argumenta i tip podatka argumenta su odvojeni dvotačkom. Između susednih argumenata se stavlja tačka zarez kao simbol razdvajanja dva susedna ulazna argumenta. Ukoliko se izostavi ključna reč koja definiše tip argumenta, argument će se tretirati kao konstanta. Ukoliko više argumenata ima isti tip argumenta i isti tip podatka argumenta, oni se mogu nabrojati razdvajajući nazive tih argumenata međusobno zarezom. Izlazni argument se navodi u vidu tipa podatka koji funkcija treba da vrati kao rezultat. U deklarativnom delu se mogu deklarisati varijable koje se koriste lokalno unutar funkcije. Ne mogu se deklarisati signali u deklarativnom delu funkcije što je logično jer bi ti signali ažurirali svoju vrednost tek na kraju funkcije (što znači da ne bi imali nikakvu upotrebnu vrednost unutar funkcije), a tada bi bili i uništeni jer bi imali lokalni karakter (vidljivi samo na nivou funkcije). Stoga nema nikakvog smisla njihovo deklarisanje unutar funkcije, pa je otuda i zabranjeno. Sekvencijalni kod predstavlja telo funkcije. Važno je napomenuti da u navođenju tipova podataka ulaznih i izlaznih argumenata ne treba navoditi opsege ni kod INTEGER tipova ni kod vektor tipova, čak i ako su definisani ti opsezi. Na primer, vektor možemo navesti kao STD_LOGIC_VECTOR, tj. ne dodajemo i opseg tog vektora. Takođe, ni komponente se ne mogu instancirati unutar funkcije. O komponentama će biti reči nešto kasnije. Primer funkcije koja vrši određivanje broja jedinica u vektoru:

```
FUNCTION broj_jedinica (SIGNAL x: STD_LOGIC_VECTOR) RETURN INTEGER IS
        VARIABLE suma:INTEGER;
BEGIN
        suma :=0;
        FOR i IN x'RANGE LOOP
                IF(x(i)='1') THEN
                        suma:=suma+1;
                END IF;
        END LOOP;
```

```
        RETURN suma;
END broj_jedinica;
```

Kao što vidimo iz primera, rezultat funkcije se vraća upotrebom ključne reči RETURN uz koju se dodaje rezultat koji treba funkcija da vrati. Čak i ako znamo unapred opsege vektora sa kojima ćemo raditi (npr 7 DOWNTO 0), svejedno ih ne možemo staviti u deklaracije argumenata funkcije. Deklarisana funkcija se poziva unutar sekvencijalnog ili konkurentnog koda preko svog naziva:

```
izlaz <= broj_jedinica (ulaz); --ova linija može biti deo konkurentog ili sekvencijalnog koda
IF(broj_jedinica (z)>5)THEN.... --može da se koristi poziv funkcije u konkurentnim ili sekvencijalnim  konstrukcijama (ovde je u pitanju sekvencijalna konstrukcija)
```

Kao primer pokažimo i funkciju koja radi komparaciju dva broja:

```
FUNCTION komparacija (SIGNAL x,y: INTEGER) RETURN STD_LOGIC_VECTOR IS
BEGIN
        IF(x<y)THEN
                RETURN "01";
        ELSIF(x>y)THEN
                RETURN "10";
        ELSE
                RETURN "11";
        END IF;
END komparacija;
```

U slučaju kada su brojevi različiti vraća se vrednost "01" ili "10", gde pozicija jedinice signalizira koji broj je veći, dok vraćena vrednost "11" signalizira da su poređeni brojevi jednaki. Iako je iz tela funkcije očigledno da je opseg vektora, koji predstavlja rezultat funkcije, uvek dva, svejedno se opseg ne navodi u tipu podatka izlaznog argumenta.

U okviru deklaracije varijabli, opseg vektora ili integera se može, ali i ne mora navoditi. Važno je napomenuti da neka razvojna okruženja ne podržavaju ovu mogućnost i zahtevaju da se opseg varijable definiše. Kao primer uzmimo funkciju koja izvršava *bit-wise* XOR operaciju nad dva vektora u međusobno obrnutom redosledu.

```
--varijabla nema definisan opseg---
FUNCTION bitwiseXOR (SIGNAL x,y: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
        VARIABLE rezultat: STD_LOGIC_VECTOR;
BEGIN
        FOR i IN x'RANGE LOOP
                rezultat(i):=x(i) XOR y(x'HIGH-i);
        END LOOP;
        RETURN rezultat;
END bitwiseXOR;
--varijabla ima definisan opseg---
FUNCTION bitwiseXOR (SIGNAL x,y: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
        VARIABLE rezultat: STD_LOGIC_VECTOR(7 DOWNTO 0);
BEGIN
        FOR i IN 0 TO 7 LOOP
                rezultat(i):=x(x'LOW+i) XOR y(x'HIGH-i);
        END LOOP;
        RETURN rezultat;
END bitwiseXOR;
```

U slučaju kada smo varijabli definisali opseg, a samim tim i dužinu, važno je da ulazni i izlazni argumenti funkcije takođe budu iste dužine, jer u suprotnom će doći do greške u kompajliranju dizajna. Očigledno, onog momenta kada varijabli ili nekom delu sekvencijalnog

koda u telu funkcije zadamo neke fiksne vrednosti poput opsega varijable, broja iteracija petlje i sl. time utičemo i na signale koji se mogu koristiti kao ulazni i izlazni argumenti funkcije, tj. oni se moraju deklarisati u skladu sa postavljenim ograničenjima unutar funkcije. Slično u primeru funkcije *komparacija*, u telu funkcije smo praktično naglasili da signal kome se dodeljuje rezultat funkcije mora biti STD_LOGIC_VECTOR dužine dva.

2.7.3.2. Procedure

Procedura, za razliku od funkcije, može da vrati više rezultata. Otuda se deklaracija njenih argumenata nešto razlikuje u odnosu na deklaraciju argumenata funkcije. Sintaksa deklaracije procedure je:

```
PROCEDURE naziv_procedure (lista argumenata) IS
        deklarativni deo
BEGIN
        sekvencijalni kod
END naziv_procedure;
```

Lista argumenata se navodi u sličnom maniru kao kod funkcija uz sledeće razlike: kod svakog argumenta se mora navesti i mod argumenta, argumenti procedure mogu biti i varijable. Mod argumenta određuje smer argumenta: ulazni (IN), izlazni (OUT) ili ulazno/izlazni (INOUT). Ukoliko se ne navede tip argumenta, onda se za ulazne argumente podrazumeva da je u pitanju konstanta, a za izlazne i ulazno/izlazne se podrazumeva da je u pitanju varijabla. Kao i u slučaju funkcija, ni kod procedura nije dozvoljena deklaracija signala u deklarativnom delu procedure, a nije dozvoljeno ni instanciranje komponenti u telu procedure. Doduše, dozvoljeno je deklarisanje signala, ali samo ako se procedura deklariše unutar deklarativnog dela procesa, što baš i nema neku veliku praktičnu primenu. Takođe, u procedurama se ne mogu koristiti ni izrazi za detektovanje ivice signala (na primer IF izrazi za uzlaznu ivicu takta koje smo već imali u nekim prethodnim primerima u okviru ovog poglavlja). Za razliku od funkcija, kod procedura argumenti poput vektora se navode sa opsegom, a takođe i integer se može navesti u vidu opsega. Procedura se poziva u vidu samostalnog izraza u okviru konkurentnog ili sekvencijalnog koda:

```
naziv_procedure (x1,x2,x3,y); --ova linija može biti deo konkurentog ili sekvencijalnog koda
IF(z=5)THEN naziv_procedure (x1,x2,x3,y); .... --izraz može da bude i unutar neke konstrukcije
```

Na osnovu toga što se prilikom navođenja argumenata, navodi i mod argumenta, procedure možemo posmatrati i kao mini entitete, koji imaju ulogu da zamene deo arhitekture pozivom procedure radi postizanja preglednijeg koda. Kreirajmo procedure koje obavljaju uloge funkcije *broj_jedinica* i funkcije *komparacija* definisanih u prethodnoj sekciji. Treba primetiti da smo u slučaju prve procedure vektor *x* deklarisali sa parametrizovanom gornjom granicom. Važno je deklarisati navedeni parametar u GENERIC delu entiteta ako se deklarisanje procedure radi unutar deklarativnog dela arhitekture, odnosno kao CONSTANT ako se deklarisanje procedure vrši u paketu.

```
PROCEDURE broj_jedinica (SIGNAL x: IN STD_LOGIC_VECTOR (N-1 DOWNTO 0); SIGNAL z: OUT INTEGER) IS
        VARIABLE suma:INTEGER;
BEGIN
        suma :=0;
        FOR i IN x'RANGE LOOP
                IF(x(i)='1') THEN
                        suma:=suma+1;
                END IF;
```

```
        END LOOP;
        z<= suma;
END broj_jedinica;

PROCEDURE komparacija (SIGNAL x,y: IN INTEGER; SIGNAL z: OUT STD_LOGIC_VECTOR(1 DOWNTO 0)) IS
BEGIN
        IF(x<y)THEN
                z<= "01";
        ELSIF(x>y)THEN
                z<= "10";
        ELSE
                z<= "11";
        END IF;
END komparacija;
```

Očigledno je da je primarna namena ulaznih argumenata da učestvuju u formiranju vrednosti izlaznih argumenata koji predstavljaju (iznose) rezultat rada procedure. Ulazno/izlazni argumenti imaju osobine i ulaznih i izlaznih argumenata, odnosno učestvuju u formiranju vrednosti izlaznih argumenata, ali i predstavljaju (iznose) rezultat rada procedure.

2.8 Komponente

Kao što smo već naveli u ovom poglavlju, entitet zajedno sa opisom svoje arhitekture čini jednu hardversku celinu. Međutim, u praksi se veoma često razvijaju veoma složene hardverske implementacije koje treba da izvršavaju velik broj funkcija, pri čemu na razvoju radi veći broj inženjera. U takvom slučaju veoma je nepraktično kompletan dizajn smestiti samo u jedan entitet, čak i uz upotrebu funkcija, procedura i biblioteka. Razlog je što više ljudi radi u paraleli na različitim delovima implementacije, a takođe što bi VHDL kod takvog entiteta bio nepregledan, i samim tim i težak za održavanje. VHDL jezik omogućava upotrebu hijerarhijskog dizajniranja implementacije, što se na jednostavan način postiže upotrebom komponenti. Hijerarhijsko dizajniranje podrazumeva da se prvo realizuju entiteti najnižeg hijerarhijskog nivoa, gde svaki entitet obavlja svoju funkciju ili skup funkcija. Zatim se entiteti u vidu komponenti uključuju u arhitekturu hijerarhijski višeg entiteta i međusobno povezuju. Ovo zatim može da se ponavlja sve dok se ne dođe do najvišeg (top) entiteta koji predstavlja koren opisa hardverske implementacije. Na ovaj način se postiže pregledniji kod i lakše održavanje koda jer su funkcije razdvojene po sopstvenim entitetima i modifikacija jedne funkcije zahteva samo izmenu jednog entiteta ako se menja samo arhitektura entiteta ili grupe entiteta ako se menjaju i portovi entiteta. S druge strane, i razvoj implementacije je brži jer timovi mogu da rade u paraleli nezavisno jedni od drugih, jedino je važno da su unapred definisani interfejsi (portovi) entiteta radi kasnijeg uvezivanja tih entiteta u hijerarhijski višim nivoima implementacije, kao i protokol komunikacije preko portova entiteta. Takođe, testiranje i verifikacija implementacije su olakšani i ubrzani jer se uvek prvo mogu testirati i verifikovati funkcije pojedinačno pa tek onda grupe funkcija, odnosno kompletna implementacija. Sve ovo treba da naglasi važnost komponenti u složenijim projektima.

Komponente se kreiraju od entiteta. Svaki entitet se piše u svom VHDL fajlu. Kada želimo neki entitet da uvezemo u drugi entitet (koji predstavlja viši hijerarhijski nivo) moramo da ga prvo deklarišemo kao komponentu. Deklaracija komponente je veoma slična deklaraciji entiteta. Deklaracija se vrši ili u paketu ili u deklarativnom delu arhitekture entiteta u koji uvozimo komponentu. Sintaksa deklaracije komponente je:

```
COMPONENT ime_komponente IS
GENERIC
(
    ime_parametra1 : tip := vrednost_parametra;
    ime_parametra2: tip := vrednost_parametra;
    ....
    ime_parametraM : tip := vrednost_parametra
);
PORT
(
    ime_porta1 : mod  tip;
    ime_porta2: mod  tip;
    ....
    ime_portaN : mod  tip
);
END COMPONENT;
```

Kao što vidimo sintaksa je gotovo identična sintaksi deklaracije entiteta. Razlika je što se umesto ključne reči ENTITY koristi ključna reč COMPONENT i što se na kraju koda END umesto naziva komponente stavlja opet ključna reč COMPONENT. U većini razvojnih okruženja se ključna reč IS može izostaviti kod deklarisanja komponente. Takođe se mogu vrednosti parametara izostaviti iz GENERIC dela, s obzirom da će konkretne vrednosti biti definisane prilikom upotrebe (instanciranja) komponente.

Kada se komponenta deklariše, treba se uvezati sa ostalim signalima u arhitekturi entiteta u koji je uvezena komponenta. Taj proces se zove instanciranje komponente. U okviru arhitekture se može instancirati proizvoljan broj instanci iste komponente. Takođe, u entitet se može uvesti proizvoljan broj različitih komponenti. Postoje dva načina za instanciranje komponente. Prvi način, nominalno mapiranje, podrazumeva da se upotrebom uparenih parova izvrši mapiranje internih signala arhitekture entiteta ili samih portova entiteta na portove uvezene komponente, odnosno mapiranje konstantnih vrednosti ili *generic* parametara samog entiteta na *generic* parametre komponente:

```
naziv_instance: ime_komponente
GENERIC MAP
(
    ime_parametra1 => mapirani_parametar1,
    ime_parametra2=> mapirani_parametar2,
    ....
    ime_parametraM => mapirani_parametarM
)
PORT MAP
(
    ime_porta1 => mapirani_signal1,
    ime_porta2=> mapirani_ signal2,
    ....
    ime_portaN => mapirani_ signalN
);
```

Naziv instance mora biti jedinstven na nivou arhitekture jer se preko njega razlikuju različite instance komponente. Takođe, signali moraju da budu identičnog tipa kao portovi uvezene komponente na koje su mapirani. Isto važi i za mapirane parametre. Ove napomene važe i za drugi način mapiranja.

Drugi način, poziciono mapiranje, vrši mapiranje na portove komponenti tako što se samo naređaju parametri, odnosno signali koji se mapiraju na komponentu. Pri tome se

podrazumeva da je redosled parametara, odnosno, portova na koje se radi mapiranje identičan onome koji je naveden u deklaraciji komponente. Ovaj način omogućuje manje pisanog koda prilikom instanciranja komponente, ali je nepregledniji i češće se dešavaju greške prilikom mapiranja. Preporuka autora skripti je da se koristi nominalno mapiranje prilikom instanciranja komponenti jer se tako dobija pregledniji kod koji je lakše održavati. Sintaksa pozicionog mapiranja je:

```
naziv_instance: ime_ komponente
GENERIC MAP
(
    mapirani_parametar1,
    mapirani_parametar2,
    ....
    mapirani_parametarM
)
PORT MAP
(
    mapirani_signal1,
    mapirani_ signal2,
    ....
    mapirani_ signalN
);
```

Kreirajmo sledeći primer. Entitet treba da ima dva ulazna vektora *x1* i *x2* i tri izlazna STD_LOGIC signala *l*, *e* i *g*. Funkcija entiteta je da proveri koji od ulaznih vektora sadrži veći broj jedinica u sebi. Ako *x1* sadrži veći broj jedinica onda treba da se aktivira izlaz *g*, ako *x2* sadrži veći broj jedinica onda treba da se aktivira izlaz *l*, i ako oba vektora imaju isti broj jedinica onda treba da se aktivira izlaz *e*. Kreiraćemo i poseban entitet koji ima funkciju brojanja jedinica u vektoru, i taj entitet ćemo uvesti kao komponentu u entitet koji vrši poređenje dva vektora. VHDL kod primera je:

```
-----Prvi VHDL fajl----------
-----definiše entitet brojac_jedinica i njegovu arhitekturu-------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY brojac_jedinica IS
GENERIC
(
        N : INTEGER:=16
);
PORT
(
        x : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        broj_jedinica : OUT INTEGER
);
END brojac_jedinica;
ARCHITECTURE shema of brojac_jedinica IS
BEGIN
        PROCESS(x)
                VARIABLE suma: INTEGER;
        BEGIN
                suma :=0; --suma je deklarisana kao varijabla
                FOR i IN x'RANGE LOOP
                        IF(x(i)= '1') THEN
                                suma:=suma+1;
                        END IF;
```

```
                END LOOP;
                broj_jedinica<=suma;
        END PROCESS;
END shema;
-----Drugi VHDL fajl----------
-----definiše entitet komparator i njegovu arhitekturu-------
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY komparator IS
GENERIC
(
        N : INTEGER:=8
);
PORT
(
        x1,x2 : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
        l,e,g : OUT STD_LOGIC
);
END komparator;
ARCHITECTURE shema of komparator IS
        --deklaracija komponente brojac_jedinica koju uvozimo
        COMPONENT brojac_jedinica IS
        GENERIC
        (
                N : INTEGER:=16
        );
        PORT
        (
                x : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
                broj_jedinica : OUT INTEGER
        );
        END COMPONENT;
        --deklarišemo interne signale u koje cemo upisati broj jedinica vektora i koje cemo da poredimo
        SIGNAL broj1, broj2: INTEGER;
BEGIN
        --instanciramo komponentu za brojanje jedinica vektora x1
        instanca1: brojac_jedinica
        GENERIC MAP
        (
                N =>N
        )
        PORT MAP
        (
                x => x1,
                broj_jedinica => broj1
        );
        --instanciramo komponentu za brojanje jedinica vektora x2
        instanca2: brojac_jedinica
        GENERIC MAP
        (
                N =>N
        )
        PORT MAP
        (
                x => x2,
                broj_jedinica => broj2
        );
        --konkurentni kod za poredenje broja jedinica oba vektora
```

```
        e<='1' WHEN (broj1=broj2) ELSE
                '0';
        g<='1' WHEN (broj1>broj2) ELSE
                '0';
        l<='1' WHEN (broj1<broj2) ELSE
                '0';
END shema;
```

Pošto imamo dva vektora, instancirane su dve komponente, po jedna za svaki vektor. Izlazni portovi instanciranih komponenti se međusobno porede da bi se odredile vrednosti izlaza kompletne realizovane logike (izlazi top entiteta *komparator*). WHEN/ELSE konstrukcije su iskorišćene za formiranje sva tri izlaza. Naravno, mogao se kreirati i ekvivalentan sekvencijalni kod koji bi radio identičnu funkciju kao dati konkurentni kod. Treba primetiti da je difolt vrednost parametra *N* u entitetu *brojac_jedinica* 16. Međutim, pošto je u entitetu *komparator* izvršeno mapiranje GENERIC parametra entiteta *komparator*, onda će *brojac_jedinica* biti implementiran sa vrednošću parametra *N*=8. Ovo pokazuje praktičnu stranu parametrizacije dizajna, gde se dizajn bez ikakvih izmena lako prilagođava različitim situacijama. Napomenimo da je svejedno gde će u telu arhitekture biti instancirane komponente (na početku, u sredini ili na kraju). Tipično se instanciranje komponenti radi ili na kraju ili na početku tela arhitekture, radi bolje preglednosti koda.

2.9 Paketi

Funkcija paketa je slična funkciji h fajlova u C programskom jeziku. U okviru paketa se mogu čuvati definicije funkcija, procedura, komponenti, korisnički definisanih tipova i podtipova podataka, kao i konstanti čime paket postaje zajedničko mesto sa kog se svi oni mogu lako pozivati u okviru entiteta u slučaju potrebe. U principu se u okviru paketa mogu definisati i signali, ali takvu praksu treba izbegavati. Što se tiče komponenti, entitet koga one predstavljaju se i dalje definiše u svom posebnom VHDL fajlu, a u okviru paketa se čuva deklaracija komponente opisana u prethodnoj sekciji. Otuda, komponentu ima smisla čuvati u paketu samo ako se ona koristi u više entiteta, jer se tada deklariše samo na jednom mestu u paketu, a u svim entitetima gde se uvozi se samo instancira, ali se ne deklariše u entitetu ako se koristi paket koji sadrži deklaraciju te komponente. Paket se sastoji iz deklaracije paketa i tela paketa, pri čemu se telo paketa koristi samo u slučaju da paket sadrži funkcije i/ili procedure. Sintaksa paketa i tela paketa je:

```
PACKAGE naziv_paketa IS
        deklarativni deo
END naziv_paketa;

PACKAGE BODY naziv_paketa IS
        opisi funkcija i procedura
END naziv_paketa;
```

Paket (PACKAGE) sadrži deklaracije svega što paket sadrži - korisnički definisani tipovi i podtipovi podataka, konstante, komponente, kao i deklaracije funkcija i procedura, ali bez tela funkcija i procedura. Telo paketa (PACKAGE BODY) sadrži kompletne opise (sadržaj) funkcija i procedura koje su deklarisane u paketu. Ako paket ne sadrži nijednu proceduru i funkciju, onda telo paketa ne postoji. Kreirani paket se po difoltu u projektu poziva preko *work* biblioteke, sem ako se ne kompajlira u neku drugu biblioteku. Primer paketa koji sadrži samo PACKAGE deo:

```
PACKAGE kratak_paket IS
	TYPE stanje_automata IS (init, mod_citanja, mod_upisa, idle);
	CONSTANT konstanta: STD_LOGIC_VECTOR(1 DOWNTO 0):= "10";
	COMPONENT brojac_jedinica IS
	GENERIC
	(
		N : INTEGER
	);
	PORT
	 (
		x : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
		broj_jedinica : OUT INTEGER
	);
	END COMPONENT;
END kratak _paket;
```

Primer paketa koji sadrži oba dela i PACKAGE i PACKAGE BODY:

```
PACKAGE dug_paket IS
	TYPE stanje_automata IS (init, mod_citanja, mod_upisa, idle);
	CONSTANT konstanta: STD_LOGIC_VECTOR(1 DOWNTO 0):= "10";
	COMPONENT brojac_jedinica IS
	GENERIC
	(
		N : INTEGER
	);
	PORT
	(
		x : IN STD_LOGIC_VECTOR(N-1 DOWNTO 0);
		broj_jedinica : OUT INTEGER
	);
	END COMPONENT;
	FUNCTION broj_jedinica (SIGNAL x: STD_LOGIC_VECTOR) RETURN INTEGER;
	PROCEDURE komparacija (SIGNAL x,y: IN INTEGER; SIGNAL z: OUT STD_LOGIC_VECTOR(1 DOWNTO 0));
END dug_paket;

PACKAGE BODY dug_paket IS
	FUNCTION broj_jedinica (SIGNAL x: STD_LOGIC_VECTOR) RETURN INTEGER IS
		VARIABLE suma:INTEGER;
	BEGIN
		suma :=0;
		FOR i IN x'RANGE LOOP
			IF(x(i)='1') THEN
				suma:=suma+1;
			END IF;
		END LOOP;
		RETURN suma;
	END broj_jedinica;
	PROCEDURE komparacija (SIGNAL x,y: IN INTEGER; SIGNAL z: OUT STD_LOGIC_VECTOR(1 DOWNTO 0)) IS
	BEGIN
		IF(x<y)THEN
			z<= "01";
		ELSIF(x>y)THEN
			z<= "10";
		ELSE
			z<= "11";
		END IF;
	END komparacija;
END dug_paket;
```

Navedeni paketi bi se uključili sa:

```
USE work.kratak_paket.all;
USE work.dug_paket.all;
```

Pošto se koristi biblioteka *work*, ne mora se koristiti njen poziv sa

```
LIBRARY work;
```

jer svaki projekat po difoltu uključuje biblioteku *work*, što je već rečeno na početku drugog poglavlja.