Glava 3

VHDL – PRIMENA U SINTEZI

Akronim **VHDL** potiče od ***V****ery High Speed Integrated Circuits* ***H****ardware* ***D****escription* ***L****anguage* i označava jezik za opis hardvera veoma brzih integrisanih kola. Nastao je iz potrebe da se olakša pisanje projektne dokumentacije i omogući simulacija digitalnih kola prilikom projektovanja. Standardizaciju je iniciralo i finansiralo Ministarstvo odbrane SAD-a. VHDL je 1987. godine usvojen kao standard: IEEE Std. 1076-1987, kraće nazvan VHDL-87. Priključivanje godine uz naziv ima smisla kada se zna da IEEE komitet za standardizaciju svakih pet godina razmatra primedbe koje stižu od korisnika VHDL-a, kako bi povećala efikasnost jezika. Sledeća varijanta jezika definisana je standardom IEEE Std. 1076-1993 (VHDL-93), a 1999 razvijena je i proširena varijanta koja podržava opis analognih i kola sa mešovitim signalima. Zvanično, ova varijanta jezika definisana je standardom IEEE Std. 1076.1-1999, dok je neformalni, ali češći naziv **VHDL AMS** (**VHDL** ***A****nalogue and* ***M****ixed* ***S****ignals*). Kasnije korekcije bile su ciljem da se ispoštuju zahtevi, odnosno primedbe korisnika i to 2000, 2002, 2007. i 2008, 2009. i 2011.

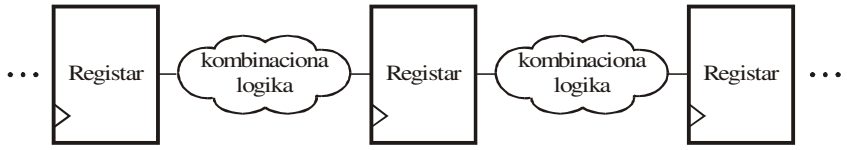
Iako se iz naziva jezika može zaključiti da mu je osnovna namena **opis hardvera**, važno je istaći da se isti VHDL kôd može primeniti u svim fazama projektovanja. To znači da isti opis prihvataju različiti alati koji se koriste tokom projektovanja digitalnih integrisanih kola. Prepoznaju ga kako simulatori, tako i programi za automatsku sintezu i to na različitim hijerarhijskim nivoima. Na ovaj način projektanti se maksimalno koncentrišu na kreativni deo u opisu željenog zadatka, a ostale faze projektovanja digitalnih IC, do fizičkog projektovanja, obavljaju se automatizovano. Posle

verifikacije simulacijom, isti VHDL opis, se prosleđuje do alata za automatsku sintezu koji „dešifruju“ kôd i preslikavaju ga u digitalne strukture vezane za određenu tehnologiju. Ovo omogućava da se iz istog VHDL opisa projektuje digitalno kolo u različitim tehnologijama (različita dužina kanala) ili različitim stilovima (ASIC ili FPGA).

Važnu osobinu VHDL-a predstavlja mogućnost jednostavnog opisa projekta na višim nivoima apstrakcije. Ova osobina proistekla je iz osnovnog koncepta jezika, a to je da se svaka logička struktura opiše na intuitivan način koji je blizak inženjerskom načinu razmišljanja. Praktično značenje ove tvrdnje biće ilustrovano na nizu primera opisa osnovnih kombinacionih i sekvencijalnih modula. U ovom poglavlju posebna pažnja biće posvećena primeni VHDL-a u projektovanju na takozvanom **RTL** (***R**egister **T**ransfer **L**evel*) nivou. Kola na ovom nivou opisa sastoje se iz registara i kombinacione logike, što je prikazano na sl. 3.1. Sintezom na RTL nivou optimizuje se logika koja je smeštena između registara.

Akcenat stavljamo na primenu VHDL-a u sintezi i verifikaciji. Zato ćemo se fokusirati na opis funkcije hardvera, ostavljajući alatima za sintezu da generišu njegovu strukturu na nivou blokova raspoloživih u određenoj tehnologiji. Strukturni opis biće pomenut samo u delu koji se tiče verifikacije primenom *Test-bench* opcije koja je sastavni deo VHDL editora u svim profesionalnim alatima za projektovanje digitalnih kola.Pojedine naredbe biće opisane samo u onoj meri koliko je neophodno da se prati izlaganje o osnovnim digitalnim kolima u narednim poglavljima. U daljem tekstu, gde god je to moguće, izlaganja vezana za logičku funkciju digitalnog kola biće ilustrovana VHDL opisom na funkcionalnom nivou.

U ovom poglavlju biće predstavljene osnove VHDL jezika za opis hardvera sa ciljem da se čitalac osposobi da razume značenje pojedinih linija kôda i poveže ga sa funkcijom koju kolo obavlja. Ne treba izgubiti iz vida da VHDL pruža mnogo više mogućnosti od onih koje su opisane u ovom poglavlju. Zato se napredni čitaoci upućuju na priručnike koji se detaljno bave primenom VHDL-a u projektovanju digitalnih kola. U realizaciji svih primera korišćene



Slika 3.1. Definicija logičkog kola na RTL nivou

su opcije koje prepoznaje većina programa za automatsku sintezu i verifikaciju. Konkretno, primeri su ilustrovani primenom *Active-HDL*™ alata koji distribuira firma *Aldec*.

Slično drugim programskim jezicima i u VHDL-u postoje neka striktna pravila kojih se korisnici moraju pridržavati. Ona se odnose na način deklarisanja tipova signala i na pravila koja definišu način opisa i mesto pojavljivanja određenih naredbi. Zato počinjemo poglavlje sa kratkim pregledom semantike i sintakse VHDL jezika.

3.1. OSNOVE SEMANTIKE I SINTAKSE

VHDL se sastoji iz tri dela:

- deklaracije biblioteka i paketa,
- deklaracije entiteta i
- opisa arhitekture.

Deklaracija biblioteka sastoji se od liste svih biblioteka koje se koriste u projektu. Biblioteka se obično deklariše sa dve linije koda. Prva je ime biblioteke, a druga je njena upotreba koja počinje ključnom rečju **use**, tj.:

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

Definisanje logičkih stanja u okviru opisa svakog projekta samo bi nepotrebno opterećivalo kôd. Umesto toga, pozivamo se samo na biblioteku (*library*) koja sadrži te podatke. U ovom slučaju radi se o biblioteci nazvanoj `IEEE`. Dakle, opis projekta može da se rastereti time što se strukture (deklaracije, elementi, itd.) koje se često javljaju, i obično su zajedničke za više projekata, definišu posebno i smeste u biblioteku. Nadalje, kad god je to moguće, koristiće se prednosti koje pruža organizacija podataka u okviru biblioteka. Cilj je da jezgro opisa bude maksimalno posvećeno konkretnom primeru, a da se sve ostalo nalazi u biblioteci.

U okviru biblioteke, koju prepoznaje VHDL, nalazi se više celina koje se zovu **paketi** (`package`). Paketi, između ostalog, sadrže podatke o logičkim komponentama, familijama, serijama, konstantama, signalima, funkcijama i procedurama. Njima se pristupa pozivanjem iskaza **use** koji znači „iskoristi“. Sintaksa iskaza **use** zahteva da se navede naziv biblioteke, zatim paketa, i na kraju se navodi željeni pojam. Svi oni razdvojeni su tačkom:

```
use <ime biblioteke>.<ime paketa>.<ime komponente>;
```

Ukoliko umesto jednog pojma stoji ključna reč **all** (sve), to znači da se koriste svi pojmovi iz specificiranog paketa. Zapravo, kad god se koriste signali tipa *std_logic* čija se definicija (data tabelom 3.1) nalazi u okviru standarda *IEEE Std. 1164*, koriste se biblioteka *IEEE* i paket *std_logic_1164* i to svi njeni elementi (**all**):

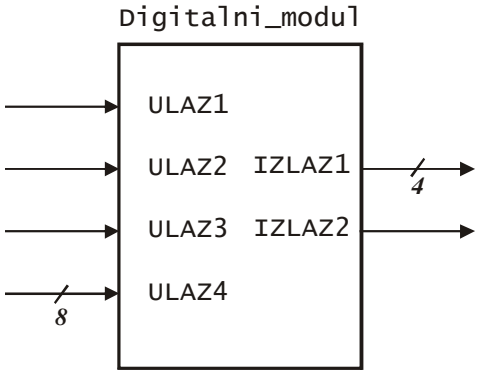
```
use IEEE. std_logic_1164.all;
```

Treba reći da korisnicima stoji na raspolaganju radna ili korisnička biblioteka sa nazivom ***work*** koju mogu proizvoljno da organizuju smeštajući sopstvene sadržaje tekućeg projekta u odgovarajuće pakete. S obzirom da je „radna", ona je uvek otvorena (dostupna), tako da nije neophodna naredba `library`.

Korisnicima je implicitno dostupna i biblioteka `std` u kojoj su smešteni podaci o tipovima promenljivih i odgovarajućim operatorima.

3.1.1. Opis hardvera

Najbolji način da se nepoznati hardver predstavi grafički jeste u vidu „crne kutije", odnosno pravougaonika, sa izvodima, portovima **(*port*)** (sl. 3.2), preko kojih je povezan sa okruženjem. Hardveru, sa četiri ulazna i dva izlazna porta, dodeljeno je ime Digitalni_modul. Tri ulaza (ULAZ1, ULAZ2 i ULAZ3) predstavljaju jednobitni signali, a četvrti ulaz (ULAZ4) čini 8-bitna reč. Izlazni port IZLAZ1 prosleđuje 4-bitni, a IZLAZ2 jednobitni



Slika 3.2. Nepoznati hardver prikazan kao blok

signal. Svaki ovakav blok prepoznaje se u VHDL-u kao *entitet* koji se definiše ključnom rečju **entity**. Da bi se entitetu dodelila funkcija, koja povezuje stanja na ulaznim i izlaznim priključcima, odnosno portovima, potrebno je definisati *arhitekturu* entiteta. Opis arhitekture počinje ključnom rečju **architecture**.

Kao što se u elektronici jedna ista funkcija može realizovati na više načina, tako i ponašanje jednog entiteta može da se opiše sa više arhitektura. Naravno, obrnuto ne važi jer jedna arhitektura može da obavlja samo funkciju za koju je projektovana. Na osnovu ovih uvodnih napomena može se zaključiti da je struktura VHDL jezika prilagođena osnovnim celinama koje se uočavaju tokom projektovanja elektronskog kola.

Osnovne konstrukcije u definisanju entiteta i arhitekture koji odgovaraju bloku sa sl. 3.2 mogu da se iskažu delimičnim opisom datim u primeru 3.1. Masnim slovima (engl. *bold*) označene su rezervisane ključne reči o kojima će biti reči u odeljku 3.1.3. Slična nomenklatura koristiće se i u ostalom delu teksta u ovom poglavlju. Važno je napomenuti da postoji još jedan, nešto složeniji oblik definisanja entiteta, kada se sem portova definišu i opšte konstante entiteta koji se zovu generici (eng. *generic*). O ovom tipu konstanti biće reči u odeljku 3.3.2.

Primer 3.1

```
entity Digitalni_modul is port(
ULAZ1, ULAZ2,     -- deklaracija portova i
ULAZ3, ULAZ4,     --  konstanti
IZLAZ1, IZLAZ2);
end entity Digitalni_modul;

architecture proba of Digitalni_modul is
   .
   .    -- deklaracija signala,
   .    -- konstanti, ...
begin
  .
  .     -- telo arhitekture
  .
end proba;
```

3.1.2. Signali

Jedan entitet povezuje se sa drugim preko portova. Kroz portove teku signali, a s obzirom da se radi o opisu digitalnih kola, reč je o digitalnim signalima. Zato i definicija signala u VHDL-u ima sve atribute koji se javljaju u realnim digitalnim kolima. Svaki ulazno/izlazni (U/I) signal u opisu entiteta naziva se **port** i predstavlja objekat podataka. Niz portova jednog bloka naziva se deklaracija porta. Pri tome, svaki port mora da ima: ime, tip podataka i smer.

Najpre, svaki signal nosi *informaciju o logičkom stanju* ili *logičkoj vrednosti*. Najjednostavnije predstavljanje podrazumeva dodeljivanje vrednosti logičke 0 ili logičke 1. Naravno, poznato je da je skup logičkih stanja u realnom kolu mnogo veći i da može da sadrži vrednost visoke impedanse Z, slabe 0, slabe 1, nepoznato stanje, itd. Da bi se uspostavio broj stanja kojima se manipuliše tokom verifikacije različitih projekata i njihovih delova, najčešće se koristi skup logičkih stanja koji je definisan standardom *IEEE 1164*. Ovaj skup vrednosti deklariše se kao *std_logic* tip signala, a prikazan je u tabeli 3.1.

Tabela 3.1. Skup vrednosti signala tipa `std_logic`

Vrednost	Značenje
U	Neinicijalizovan signal
X	Jako (forsirano) nepoznato stanje
0	Jaka (forsirana) nula
1	Jaka (forsirana)jedinica
Z	Visoka impedansa
W	Slabo nepoznato stanje
L	Slaba nula (*Low*)
H	Slaba jedinica (*High*)
-	Nebitno (nedefinisano) stanje (*don't care*)

Proširenи skup vrednosti neophodan je zbog pravilnog modelovanja stanja u digitalnom kolu prilikom simulacije. Signali tipa '0', '1', 'L' i 'H' su logička stanja podržana sintezom, dok su 'Z' i '-', takođe, podržani sintezom, ali samo za izvore sa tri stanja i nebitna stanja. Stanja 'U', 'X' i 'W' nisu podržana sintezom. Da bi se izbegle razlike u tumačenju VHDL kôda od strane različitih programa za automatsku sintezu, preporučuje se da se signalima dodeljuju samo logičke vrednosti 0, 1 i Z.

Jednobitne vrednosti u VHDL-u pišu se unutar apostrofa ('0' i '1'), a višebitne vrednosti i binarni brojevi unutar navodnika ("1101"). Dekadni brojevi pišu se bez posebnih oznaka.

Druga važna karakteristika signala odnosi se na razliku između signala koji teku samo kroz jednu *žicu* (signal tipa **bit**), od signala koji se vodi kroz *magistralu*, odnosno **bus**. Signal tipa bus predstavlja niz, odnosno vektor, koji se sastoji od više signala tipa bit (na sl. 3.1 to su signali ULAZ4 i IZLAZ1). Zato se on deklariše kao signal tipa *std_logic_vector*. Njemu je neophodno definisati dužinu izraženu brojem bitova, a mora se naznačiti i položaj cifara najmanje (LSB), odnosno najveće težine (MSB). U realnim sistemima, recimo PC računaru, kod koga su magistrale u obliku sivog „kaiša“, LSB /MSB je označen crvenom bojom.

Tabela 3.2. VHDL tipovi signala

Tip signala	Sintaksa	Semantika
bit	**signal** A:bit	0 ili 1
bit_vector	**signal** A: bit_vector(0 **to** 7); **signal** B: bit_vector(7 **downto** 0);	Vektor bitova u rastućem ili opadajućem poretku
std_logic; std_logic_vector	library IEEE; **use** IEEE.std_logic_1164.all; **signal** D0,D1,S :std_logic; **signal** A:std_logic_vector(0 **to** 7); **signal** B:std_logic_vector(7 **downto** 0);	Prethodno definisani tipovi signala koji su uključeni u IEEE biblioteku kao paket std_logic_1164
integer	**signal** A:integer **range** – 128 **to** 127:	8–bitni celi broj u 2-komplementu
boolean	**signal** status: boolean;	false(0) ili true(1)
nabrajanje	**type** stanje **is** (stanje A, stanje B, stanje C); **signal** s: stanje;	Stanje automata sa konačnim brojem stanja

Karakteristični tipovi signala, definisani standardnima *IEEE 1076* i *1164* dati su u tabeli 3.2.

Sintaksa signala bit_vector ima sledeće značenje:

A je 8-bitni signal, 0 je LSB, a 7 MSB,

B je 8-bitni signal, 7 je MSB, a 0 LSB.

Treća karakteristika signala koja proističe iz karakteristike digitalnih kola jeste jasno *usmeren put signala* od izlaza jednog kola ka ulazu narednog. Zato se prilikom deklarisanja portova nekog entiteta navodi smer ili `mod` (**mode**) kojim se oni razvrstavaju na:

- ulazne, označavaju se sa ***in***
- izlazne, označavaju se sa ***out***
- dvosmerne, označavaju se sa ***inout*** *i*
- bafer, označavaju se sa ***buffer***.

Mod porta određuje se na osnovu toga gde se nalazi generator signala u odnosu na entitet.

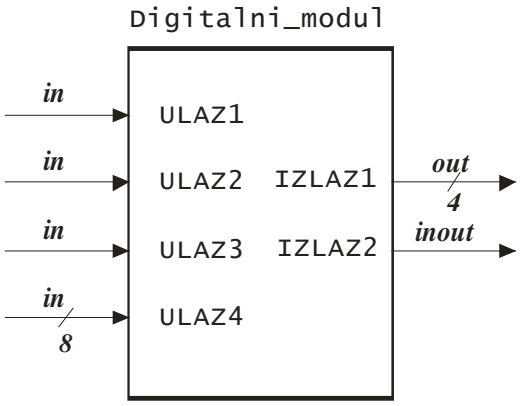
In: Ukoliko je pobuda van entiteta (podaci ulaze u blok) radi se o ***in*** signalu (portovi označeni sa `ULAZ1, ULAZ2, ULAZ3` i `ULAZ4` na sl. 3.2). Entitet može samo da čita sadržaj ovog signala, a ne može mu dodeljivati novu vrednost. Dakle, podaci teku samo u entitet. Koristi se za neusmerene ulaze podataka, za ulaze takta i kontrolne ulaze kao što su: *enable, load, read/write* i sl.

Out: Ukoliko je pobuda u samom entitetu, radi se o ***out*** signalu (signal `IZLAZ1` na sl. 3.2). Vrednost mu se dodeljuje unutar entiteta, a njegov sadržaj ne može da se koristi za pobudu drugih delova istog entiteta (nije čitljiv u istom entitetu).

Inout: Dvosmerni signali deklarišu se portom ***inout***. Preko ovog porta dozvoljen je protok podataka unutar ili izvan entiteta (`IZLAZ2` sl. 3.2). Dvosmernost se obično ostvaruje trostatičkim kolima (sl. 3.3).

Buffer: Izvor signala ovog porta je, takođe, unutar arhitekture entiteta (interni signal). Dakle, signal na portu buffer je izlazni za entitet, ali se može ponovo upotrebiti unutar entiteta. Dozvoljeno je da se pojavi s obe strane operatora pridruživanja <=.

Na sl. 3.3 prikazani su modovi (režimi) portova jednog entiteta. Data je i logička šema entiteta koja dodatno objašnjava značenje pojednih portova.



Slika 3.3. Modovi portova entiteta sa slike 3.2

U primeru 3.2 dat je VHDL opis portova entiteta `Digitalni_modul` (sl. 3.2). Redosled navođenja podataka je:

naziv porta: mod tip;

Primer 3.2

```
entity Digitalni_modul is port(
ULAZ1, ULAZ2, ULAZ3: in std_logic;
ULAZ4: in std_logic_vector (7 downto 0);
IZLAZ1: out std_logic_vector (3 downto 0);
IZLAZ2: inout std_logic
);
end entity Digitalni_modul;
```

3.1.3. Osnove VHDL pravopisa

Svaki jezik karakteriše skup sintaksnih pravila za pisanje kojih se korisnici moraju pridržavati. Ta pravila mogu da se podele u dve osnovne kategorije:

- način označavanja pojmova i
- format navođenja pojmova.

Osnovna pravila u VHDL-u koja se tiču načina *označavanja pojmova* jesu:

- Ne pravi se razlika između malih i velikih slova.

  Sve sledeće oznake imaju isto značenje:

```
Digitalni_modul
digitalni_modul
DIGITALNI_MODUL
```

- ***Postoje rezervisani karakteri koji se NE SMEJU koristiti za označavanje pojmova: +, -, !, ?, itd.***

  Umesto liste takvih karaktera, ovde dajemo skup opštih pravila:

  a. Svi nazivi (identifikatori) moraju da počnu slovnim znakom,
  b. Koristiti samo slova (a-z i A-Z), brojeve (0-9) i podvučenu crtu (_),
  c. NE koristiti znakove interpunkcije (!, ?, ., ,, itd.),
  d. NE koristiti donje crtice u nizu (_ _),
  e. NE može donja crtica da bude posljednji znak i
  f. NE mogu dva različita pojma u okviru istog entiteta ili arhitekture imati iste identifikatore.

  U primeru 3.3 dato je nekoliko ispravnih i neispravnih oznaka.

Primer 3.3

Ispravne oznake	Neispravne oznake	Razlog
`Digitalni_modul`	`1_Digitalni_modul`	Prekršeno pravilo a
`Digitalni_modul_0`	`Digitalni_#0_modul`	Prekršeno pravilo b
`DIGITALNI_MODUL_9`	`Digitalni!_modul`	Prekršeno pravilo c
`Digitalni_modul_0`	`Digitalni_ _modul`	Prekršeno pravilo d

- Postoje rezervisane reči koje se NE SMEJU koristiti za označavanje pojmova.

Ove reči date su u tabeli 3.3.

Samo neki programi za sintezu mogu da detektuju primenu ključnih reči u oznakama, ostali rezultuju čudnim porukama o tipu greške ili pogrešnim rezultatom sinteze bez upozorenja!

Tabela 3.3. Rezervisane (ključne) reči

abs	**downto**	**library**	**procedure**	**subtype**
access	**else**	**linkage**	**process**	**then**
after	**elsif**	**literal**	**pure**	**to**
alias	**end**	**loop**	**range**	**transport**
all	**entity**	**map**	**record**	**type**
and	**exit**	**mod**	**register**	**units**
architecture	**file**	**nand**	**reject**	**unaffected**
array	**for**	**new**	**rem**	**until**
assert	**function**	**next**	**report**	**use**
attribute	**generate**	**nor**	**return**	**variable**
begin	**generic**	**not**	**rol**	**wait**
block	**group**	**null**	**ror**	**when**
body	**guarded**	**of**	**select**	**while**
buffer	**if**	**on**	**severity**	**with**
bus	**impure**	**open**	**shared**	**xor**
case	**in**	**others**	**signal**	**xnor**
component	**inertial**	**out**	**sla**	
configuration	**inout**	**package**	**sll**	
constant	**is**	**port**	**sra**	
disconnect	**label**	**postponed**	**srl**	

- Linijski komentari počinju dvostrukim znakom minus „--“.

Značenje teksta koji se nalazi iza „**--**“, ignoriše se do kraja tekuće linije. On nije sastavni deo opisa kola, već se tretira kao komentar. Veoma je preporučljivo da se svaki deo kôda dodatno opiše komentarom kako bi ostali učesnici u projektu lakše tumačili značenje izvornog kôda. Primena linijskih komentara vidi se na primeru 3.4 kojim je opisan entitet sa sl. 3.2:

Primer 3.4

```
entity Digitalni_modul is port

 ( ULAZ1: in std_logic; -- svaki port moze da se
            -- deklarise u posebnoj
                       -- liniji
    ULAZ2, ULAZ3: in std_logic; -- portovi istog
tipa
          -- mogu da se deklarisu u istoj liniji
-- tip bit-vektor mora da sadrzi
-- duzinu i redosled:
   ULAZ4: in std_logic_vector (7 downto 0);
-- 7 je MSB
   IZLAZ1: out std_logic_vector (0 to 3);
-- 0 je LSB
   IZLAZ2: inout std_logic   -- ovde nema ';'
       );
-- jer se ';' stavlja iza
-- zaokruzene logicke celine
-- koja sadrzi i zatvorenu zagradu
end entity Digitalni_modul;
```

- Ne postoji ograničenje u broju karaktera kojim se može označiti neki pojam.

Međutim neki programi za sintezu prepoznaju najviše 32 karaktera. Zato se preporučuje da broj karaktera u jednoj oznaci ne prelazi 32. *Dobro je da oznake budu dovoljno duge da ukažu na pravo značenje, ali da ne budu i preduge. Dobra je praksa da se signali nazovu* **clock, data** ili **global_input,** *a ne* **c, d,** ili **g**.

Format pisanja VHDL-a treba sagledati sa stanovišta opisa pojedinih celina. Pri tome celinu čini jedna naredba, struktura naredbi ili grupa naredbi. Pod jednom ***naredbom*** podrazumevamo:

- opis koji počinje nekom od rezervisanih reči ili
- opis aktivnosti koja označava dodeljivanje vrednosti nekom signalu.

Pod ***strukturom naredbi*** podrazumevaćemo više povezanih naredbi sa jasnim semantičkim značenjem. Ovo će biti jasnije kada budemo opisivali **if**, **case** i slične naredbe. Za sada ćemo navesti samo primer logičkih celina u **if**...**then**...**elsif**...**else** strukturi u primeru 3.5.

Primer 3.5

```
if uslov1 then akcija1;
elsif uslov2 then akcija2;
else akcija3;
end if;
```

- Svaka logička celina završava se simbolom „;“.

Treba primetiti da ovo važi i za informaciju o završetku opisa cele strukture, **end if**. Jasno je da „**if** uslov1“ ne predstavlja logičku celinu (pa ni naredbu), jer se posle definisanja uslova očekuje još neka aktivnost. Takođe, neposredno iza reči **then** očekuje se izvršenje neke akcije i bez nje ne može da se govori o završetku jedne logičke celine. Tek kada se kompletira logička celina, onda se stavlja „**;**“. Slično važi i za **elsif** naredbu u okviru ***if...then...elsif...else*** strukture.

- Grupa naredbi obično počinje ključnom reči ili identifikatorom, a može da sadrži:
  - skup naredbi kojim se deklarišu promenljive koje se javljaju u toj grupi naredbi,
  - ključnu reč **begin** za označavanje početka opisa tela grupe naredbi,
  - telo grupe naredbi i

- ključnu reč **end** kojom se opis grupe naredbi završava.

Grupom naredbi definišu se entiteti, arhitekture, procesi i neke druge celine. O svakoj od njih pojedinačno, biće više reči kasnije. Za sada napominjemo da se iza **begin** očekuje nastavak opisa, što znači da se iza ove reči ne očekuje znak „**;**“ koji označava završetak logičke celine.

- U okviru jedne logičke celine VHDL koristi „slobodni format“ za pisanje kôda, kao što je pokazano u primeru 3.6.

Ignorišu se prekidi linije i prazni karakteri. Svi navedeni opisi u Primeru 3.6 su korektni.

Primer 3.6

```
if uslov1 then akcija1;     -- prva varijanta

if    uslov1  then   akcija1; -- druga varijanta

if uslov1
        then akcija1;     -- treca varijanta


if
uslov1
then akcija1;        -- cetvrta varijanta


if
uslov1
then            -- peta varijanta
```

- VHDL prepoznaje predefinisane tipove podataka i korisnički definisane tipove podataka.

Oni moraju da se deklarišu pre upotrebe. Treba reći da, osim podataka tipa `std_logic`, odnosno `std_logic_vector`, VHDL prepoznaje i druge tipove podataka od kojih su osnovni: `BOOLEAN` (sa elementima `True, False`),

BIT (sa elementima 0, 1), `INTEGER` (celi brojevi u rasponu -2^{32-1} do $+2^{32-1}-1$). Da bi se olakšao opis na funkcionalnom nivou, zgodno je da se koriste aritmetički operatori (+, -, *, /), a da se tipovima `std_logic_vector` dodeljuje značenje cifre a ne niza nula i jedinica. U binarnoj interpretaciji dekadnog broja, MSB može, a i ne mora, da označava znak broja. Da bi se nedvosmisleno tumačila vrednost podatka tipa `std_logic_vector`, VHDL prepoznaje i tipove `signed` i `unsigned` koji se koriste za interpretaciju broja sa znakom i bez znaka, redom.

3.1.3. VHDL operatori

S obzirom da je VHDL namenjen opisu hardvera digitalnih kola, potpuno je prirodno da podržava korišćenje različitih operatora nad signalima tipa `std_logic`. Spisak operatora koje VHDL podržava dat je u tabeli 3.4.

Tabela 3.4. VHDL operatori

Operator	Klasa operatora
****, abs, not**	Mešoviti operatori
***, |, mod, rem**	Operatori množenja/deljenja
+, -	Operatori predznaka
+, -, &	Operatori zbrajanja
=, /=, <, <=, >, >=	Relacioni operatori
SLL, SLR, SLA, SRA, ROL, ROR	Operatori pomeranja
AND, NAND, OR, NOR, XOR, XNOR	Logički operatori

Dodeljivanje logičke vrednosti nekom signalu ili portu označava se simbolima „manje“ „jednako“ <=, koji čine “strelicu ulevo”. Rezultat operacije nad većim brojem signala, npr. **a** i **b** može da se iskaže na sledeći način:

```
rezultat <= a AND b;
```

Od svih logičkih operatora NOT ima najveći prioritet, dok su svi ostali logički operatori istog prioriteta. Redosled njihovog izvršavanja određuje se redosledom navođenja u naredbi. Tako u naredbi:

```
rezultat <= a AND b OR c XOR d;
```

izvrši se najpre AND operacija, zatim OR i na kraju XOR, što može da se iskaže kao

```
(((a AND b) OR c) XOR d).
```

Međutim u opisu:

```
rezultat <= a AND NOT b OR c XOR d;
```

redosled izvršavanja operacija jeste NOT, AND, OR, XOR:

```
(((a AND (NOT b)) OR c) XOR d).
```

Redosled izvršavanja operatora može da se kontroliše upotrebom zagrada tako da se u primeru:

```
rezultat <= (a AND b) OR (c XOR d);
```

najpre obave AND i XOR operacija, a zatim OR. Da bi se povećala čitljivost koda, preporučuje se korišćenje zagrada. Generalno, najviši prioritet imaju mešani operatori (****** - stepenovanje, **abs** - apsolutna vrednost i **NOT** - logički operator negacije).

Za opis ponašanja na funkcionalnom nivou mnogo je pogodnije da se podaci tipa `std_logic_vector` interpretiraju kao brojevi u dekadnom sistemu. To podrazumeva mogućnost konverzije podataka iz tipa `std_logic_vector` u numeričke tipove `signed` ili `unsigned` ili `integer`, a i u obrnutom smeru. Zatim je moguće koristiti operatore za aritmetičke operacije: sabiranja "+", oduzimanja "-", množenja "*", deljenja "/", kao i poređenja jednako "=", manje "<", veće ">", manje ili jednako "<=", veće ili jednako ">=", nije jednako, različito,"/=". Značenja ostalih operatora su sledeća: `mod` - modul, `rem`-ostatak, `SLL` i `SLR` - logičko pomeranje ulevo i udesno, redom, `SLA` i `SRA` - aritmetičko pomeranje u levo i u desno, redom, `RQL` i `ROR` - rotiranje ulevo i udesno, redom. Svi ovi tipovi, kao i operatori definisani su u biblioteci pod nazivom `ieee` u paketu `numeric_std`.

3.1.4. Konkurentne i sekvencijalne naredbe

VHDL je namenjen za opis ponašanja u digitalnim sistemima. S obzirom da se aktivnosti u hardveru dešavaju uglavnom paralelno, konkurentno, i u opisu arhitekture nekog entiteta primenjuje se ista logika. Arhitektura može da sadrži više logičkih celina. Događaji iz jedne celine najčešće ne utiču na događaje u ostalim, odnosno odvijaju se nezavisno, konkurentno, tj. paralelno

u vremenu. Međutim, postoje slučajevi kada su događaji međusobno uslovljeni. Tada jedan događaj pokreće naredni, odnosno postoji sekvenca događaja koju hardver izvršava u striktno definisanom redosledu. Zato VHDL podržava dva osnovna metoda kroz koje se signalima unutar arhitekture dodeljuju vrednosti. To su:

- konkurentna i
- sekvencijalna dodela vrednosti signalima.

Operator dodeljivanja <= još se naziva naredba jednostavnog dodeljivanja (engl. *simple signal assignment*), jer se odnosi na dodeljivanje pojedinom signalu ili pojedinoj grupi signala. Osim ovog, VHDL podržava izborne (selekcione) i uslovne naredbe dodeljivanja. To su naredbe istovremenog ili konkuretnog dodeljivanja (engl. *concurrent assignment*). Njima se modeluju istovremene (paralelne) aktivnosti hardvera. Događaji iz jedne celine ne utiču na događaje iz ostalih celina. Drugim rečima, događaji se odvijaju nezavisno, tj. paralelno u vremenu. Zato i redosled njihovog navođenja, pri opisu arhitekture, nije važan. Ove naredbe, koriste ključne reči **with** i **when**. Naredbom izbornog dodeljivanja, signalu se dodeljuje vrednost uslova, a uslovi nakon ključne reči **when** moraju da budu međusobno isključivi. Sintaksa ove naredbe je sledeća:

```
with izraz select
  ime signala <= izraz when konstanta_vrednost {, izraz when
konstanta_vrednost};
```

Semantika uslovnog dodeljivanja (engl. *conditional signal assignment*) proizilazi iz sekvencijalnog (slednog) karaktera izračunavanja, jer se signalu dodeljuje vrednost utvrđena izrazom uz istinit logički izraz uslova. Sintaksa ove naredbe je:

```
ime_signala <= izraz when logički_izraz else {, izraz when
logički_izraz else} izraz;
```

Upotreba naredbi konkurentnog dodeljivanja pokazana je u primeru 3.7.

Primer 3.7

Opisati BCD/dekadni decoder sa aktivnim nulama na izlazu upotrebom konstruktora:

a) **with** – **select** i

b) **when** – **else**.

U oba slučaja opis entiteta je isti:

```
library ieee;
 use ieee.std_logic_vector_1164.all;

    entity dek4na10 is
      port (A : in std_logic_vector(3 downto 0);
            Y : out std_logic_vector(0 to 9);

end dek4na10;
```

Pri opisu arhitekture dekodera naredbom izbornog dodeljivanja, upotrebom konstruktora **with** – **select**, vrednost ulaznog vektora A određuje uzorak bitova koji se pridružuju vektoru izlaza Y. Definišu se sve alternative izlaznog signala. Nekorišćene ulazne kombinacije 1010 do 1111 obuhvaćene su ključnom (rezervisanom) reči **others**:

a)

```
architecture ponasanje of dek4na1 is
begin
         with A select
Y<=”0111111111” when ”0000”,
Y<=”1011111111” when ”0001”,
.
.
.
 Y<=”1111111110” when ”1001”,
Y<=”1111111111” when others;
end ponasanje;
```

b)

Pri opisu arhitekture naredbom uslovnog pridruživanja, logički izrazi nakon ključne reči **when** ispituju se po redu, pa pridruživanje nije potrebno eksplicitno navoditi.

```
architecture ponasanje of dek4na1 is
begin
Y<="0111111111" when A="0000" else
   "1011111111" when A="0001" else
.
.
.
    "1111111110" when A="1001" else
   "1111111111" when others;
 end ponasanje;
```

Opis jedne arhitekture obično sadrži više signala kojima treba paralelno dodeliti vrednosti. Zato se u okviru tela arhitekture može definisati više procesa i više konkurentih dodela vrednosti signalima. U jednom istom trenutku procesi i konkurentna dodela vrednosti signalima obavljaju se paralelno. Zato i redosled njihovog navođenja u okviru opisa arhitekture nije važan. Ovo je ilustrovano u primeru 3.8.

Da bi se omogućio opis sekvencijalnih događaja, koji suštinski definišu određeni proces, uvedena je, kao posebna kategorija, grupa naredbi nazvana **process**. Njena sintaksa prikazana je u primeru 3.8, a detaljno će biti objašnjena nešto kasnije.

Za razliku od konkurentnog dodeljivanja vrednosti signalima unutar arhitekture, dodeljivanje vrednosti unutar procesa obavlja se sekvencijalno po redosledu po kome su naredbe navedene. Zato je redosled navođenja naredbi unutar procesa kada se koriste promenljive, veoma bitan. Ovo je prikazano u primeru 3.9.

Primer 3.8

```
architecture proba of
    Digitalni_modul is
 begin
  IZLAZ2 <= ULAZ1 OR ULAZ2;
  IZLAZ1(0) <= ULAZ3 AND
            ULAZ4(0);
  seq: process (ULAZ4(7))
    begin  .
      .
      .
    end process seq;
end architecture proba;
```

=

```
architecture proba of
    Digitalni_modul is
 begin
  IZLAZ2 <= ULAZ1 OR ULAZ2;
  seq: process (ULAZ4(7))
    begin  .
      .
      .
    end process seq;
  IZLAZ1(0) <= ULAZ3 AND
            ULAZ4(0);
   end architecture proba;
```

Primer 3.9

```
architecture proba of
       Digitalni_modul is
 begin
  seq: process
   (ULAZ1, ULAZ2, ULAZ3,
    ULAZ4)
    begin
     IZLAZ2 <= ULAZ1 OR
               ULAZ2;
     IZLAZ2 <= ULAZ3 AND
               ULAZ4(0);
    end process;
 end proba;
```

≠

```
architecture proba of
       Digitalni_modul is
 begin
  seq: process
   (ULAZ1, ULAZ2, ULAZ3,
    ULAZ4)
    begin
     IZLAZ2 <= ULAZ3 AND
               ULAZ4(0);
     IZLAZ2 <= ULAZ1 OR
               ULAZ2;
    end process;
 end proba;
```

Generalno gledano, naredbe u procesu, osim sekvencijalnog izvršavanja, karakterišu još dve važne osobine:

- proces traje u beskonačnoj petlji ukoliko ne dobije zahtev da se zaustavi i
- proces se zaustavlja naredbom čekanja (***wait***), dok se vrednost nekog od signala, iz liste osetljivosti, ne promeni.

Naredbom čekanja (**wait**), proces se bezuslovno zaustavlja. Postoje i uslovne naredbe čekanja. Uslov se navodi iza ključnih reči **for**, **on** i **until.** Uslovna **wait** naredba podrazumeva da se čeka:

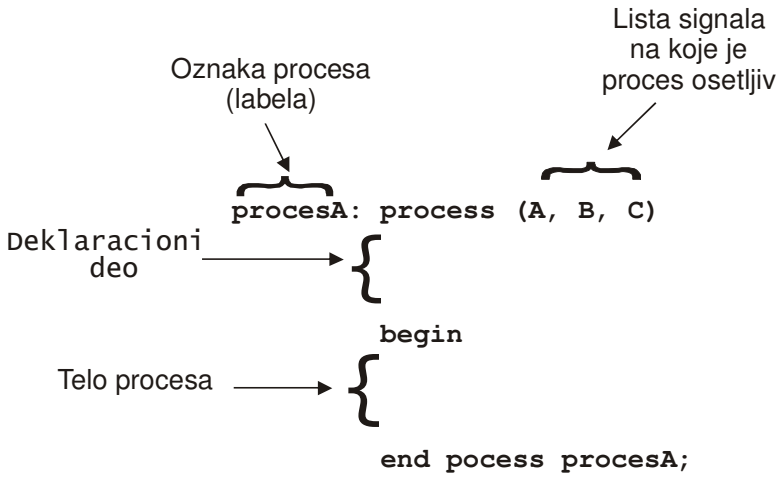
- za neki vremenski interval (**for**),
  for <vreme> -- vremenski interval,
- na promenu nekog signala (**on**)
  on <ime signala> -- promena vrednosti nekog signala ili
- dok ne bude ispunjen neki logički uslov (**until**).
  until <bulov iskaz> **--** ispunjenost bulovog (true/false) uslova.

Naglasimo još jednom da je proces, bez naredbe čekanja, stalno aktivan (traje "beskonačno") ukoliko nije navedena lista signala na koje je osetljiv.

Da bismo objasnili primenu procesa, posmatraćemo samo njegov jednostavni oblik koji je prikazan na sl. 3.4. Ovaj oblik podrazumeva da jedan proces može da počne samo kada nastane promena u nekom od signala koji se u vidu liste navode na početku deklaracije procesa. To su signali na koje je proces *osetljiv*. Ovakav opis ekvivalentan je slučaju da se kao poslednje naredbe u procesu nalaze one koje nalažu čekanje sve dok ne dođe do promene stanja bilo kog od signala na koje je proces osetljiv. Detaljniji opis procesa može se naći u [Ash96, Zwo00].

Svaki proces može da nosi jedinstvenu oznaku - labelu. Ona je opciona, ali se preporučuje naročito sa stanovišta sinteze. Nazivima signala koji se generišu automatskom sintezom pridodaje se, ukoliko je deklarisano, ime procesa iz koga su proistekli, čime se olakšava tumačenje celog projekta, naročito u fazi otkrivanja potencijalnih grešaka.

Oznaka (labela) procesa završava se sa simbolom „:" iza koga sledi ključna reč **process**, nakon čega se, u malim zagradama, navodi lista signala na koje je proces osetljiv. Oznake pojedinih signala u listi razdvojene su zapetama. Proces nazvan `procesA` sa sl. 3.4 aktiviraće se kad god nastane promena u ma kome od signala `A, B` ili `C`. Navođenje kompletne liste signala na koje je



Slika 3.4. Osnovna struktura procesa

proces osetljiv od *izuzetnog* je značaja za pravilan opis projekta. Zato mu se mora posvetiti posebna pažnja, kako tokom opisa tako i tokom verifikacije kôda.

Proces, kao i arhitektura, sadrži polje za deklarisanje promenljivih, lokalnih za taj proces. Lokalni signali *ne smeju* da se deklarišu u okviru `procesa`, tj. nakon ključne reči **`begin`**.

U okviru procesa navode se naredbe koje se izvršavaju sekvencijalno. Sekvencijalne naredbe obuhvataju naredbe čekanja, grananja, prekida petlje i druge, slične, naredbe date u tabeli 3.5. One sadrže i naredbu jednostavnog dodeljivanja.

Naredba grananja **if-elsif-else** usmerava izvršavanje VHDL koda u zavisnosti od vrednosti logičkih izraza nakon klauzula **`if`**, odnosno **`elsif`**. Naredba višestrukog grananja usmerava program na odgovarajuće alternative, u zavisnosti od vrednosti izraza u klauzuli **`case`**. Treba je tumačiti kao: *u slučaju da važi definisani izraz, izlaz dobija odgovarajuću vrednost.*

Naredbe **for-loop** i **while-loop** su naredbe upravljanja petljom. Petlja **for-loop** će se izvršavati sve dok je promenljiva petlje u definisanom opsegu. S druge strane, **while-loop** petlja će se izvršavati dokle god se ne ispuni zadati uslov.

Tabela 3.5. Sintaksa sekvencijalnih naredbi procesa sa oznakom procesa

Naredba	Sintaksa
Naredba čekanja **wait**	procesA **wait** [**on** ime_signala{, ime_signala}] [**until** logički_izraz] [**for** vremenski_izraz];
Naredba petlje s brojačem **for-loop**	procesA **for** ime_promenljive **in** opseg **loop** naredba; {naredba;} **end loop** proces;
Naredba uslovne petlje **while-loop**	procesA: **while** logički_izraz **loop** naredba; {naredba;} **end loop** procesA;
Naredba prekida petlje **exit**	**exit** [**when** logički_izraz];
Naredba grananja **if-elsif-else**	**if** logički_izraz **then** izraz; {izraz;} **elsif** logički_izraz **then** izraz; {izraz;} **else** izraz; {izraz;} end if;
Naredba višestrukog grananja **case**	**case** izraz **is** **when** konstanta_vrednost=> izraz; {izraz;} end case;

Postoje i naredbe prekida petlji. To su **exit** i **next**. Naredba **exit** uslovljava izlazak iz petlje. Naredbom **next** preskakače se ostatak petlje.

Opis procesa završava se sa **end process**, a opciono može da se doda ime procesa što se uglavnom i preporučuje. Poređenjem opisane strukture sa prethodnim primerom, vidi se da nije neophodno uz naredbu **end process** naznačiti i oznaku procesa (u slučaju primera 3.9, **seq**).

Treba napomenuti da, iako se unutar procesa naredbe izvršavaju sekvencijalno, hardver dobijen posle sinteze, realizuje se na bazi konkurentne logike. Međutim, unutar nje su ugrađeni elementi koji obezbeđuju da se na izlazu pojavljuju podaci u željenom redosledu.

3.1.4. Implicitna memorija

Sekvencijalnim naredbama mogu se opisati sekvencijalni sklopovi. Nekompletiranom naredbom grananja **if-then** modeluje se pamćenje prethodne vrednosti. Razmotrimo naredni kôd:

```
process (lista_osjetljivosti)
begin
   if uslov then A <= izraz;
end if;
end process;
```

Ako je uslov naredbe **if-then** ispunjen, signal A dobija vrednost određenu izrazom. Ako, međutim, uslov nije zadovoljen, signal A se ne menja, već zadržava prethodnu vrednost. Ovakav način modelovanja pamćenja naziva se **implicitna memorija** (engl. *Implicit memory*).

Primer 3.10

Primenom implicitne memorije modelovati sinhroni D leč sa aktivnim CP=1.

Takt impuls *CP* je signal liste osetljivosti. Ako je *CP=0*, nema promene stanja na izlazu Q, a ako je *CP=1*, onda je *Q=D*.

```
library ieee;
use ieee.std_logic_vector_1164.all;
entity Dlec is
          port (D, CP: in std_logic;
                   Q : out std_logic);
end Dlec;

architecture ponasanje of Dlec is
begin
    process (D, CP)
    begin
       if CP='1'
       then Q<=D;
       end if;
   end process;
         end ponasanje;
```

Promena stanja flipflopa sinhronizovana je sa prednjom ili zadnjom ivicom takt impulsa CP. Za modelovanje sinhronizacije sa ivicom takt impulsa koristi se atribut `event`. Sintaksa primene ovog atributa za opis flipflopa osetljivog na prednju ivicu je:

- **if** `CP'event` **and** `CP='1'` **then**, kada se koristi naredba **if**…**then**, odnosno
- **wait until** `CP'event and CP='1';` kada se koristi naredba čekanja **wait-until**.

Naredba čekanja može se upotrebiti i bez atributa `event`, na sledeći način:

```
wait until CP='1';
```

što se tumači kao:

```
wait on CP until CP='1';.
```

Ovo znači da je proces neaktivan sve do promene *CP* (`wait on` *CP*) signala, a ne izvršava se ukoliko (**until**) nije *CP*=1, što odgovara prednjoj ivici impulsa takta.

Primer 3.11

Opisati D flipflop osetljiv na prednju ivicu takt signala korišćenjem naredbe čekanja.

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

entity ivicni_DFF is
     port (D, CP : in std_logic;
             Q : out std_logic);
end ivicni_DFF;
architecture ponasanje of ivicni_DFF is
begin
         process
          begin
     wait until CP'event and CP='1'; Q<=D;
          end process;
   end ponasanje;
```

Ako flipflop poseduje asinhrone ulaze za direktno brisanje (R_d) i direktan upis (S_d), treba prvo ispitati vrednost asinhronih ulaza, pa onda pojavu aktivne ivice takt impulsa CP. Ukoliko je signal R_d dominantan nad S_d, što je najčešće, on se modeluje sa:

```
if Rd='1' then Q<='0';

elsif Sd='1' then Q<='1';

elsif CP'event and CP='1' then Q<=D;
```

U slučaju zahteva da postavljanje S_d ima prednost nad brisanjem R_d, prvo bi bio naveden uslov za S_d, pa onda za R_d.

3.2. STILOVI OPISA PROJEKTA

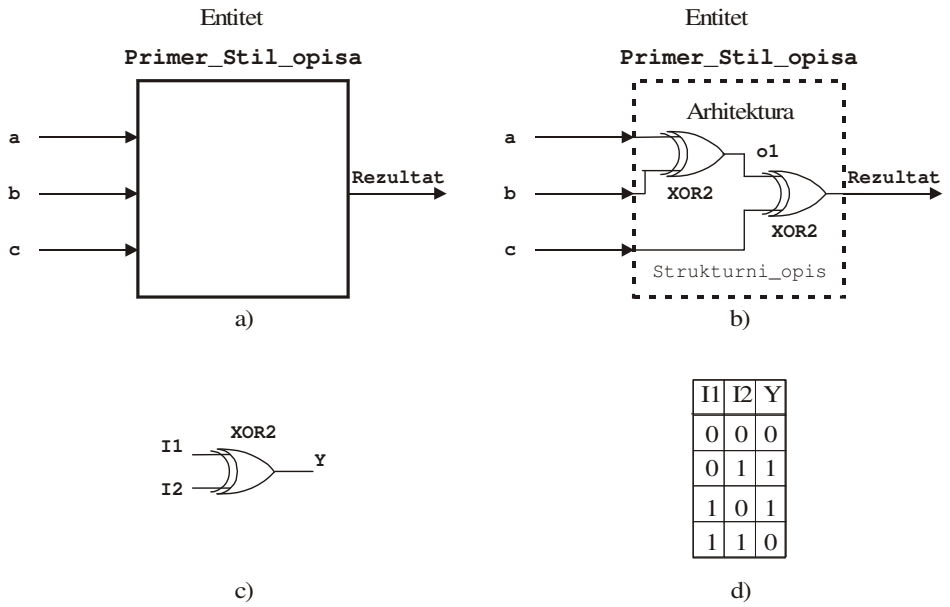
VHDL podržava opis projekata na više hijerarhijskih nivoa; od algoritamskog, do nivoa logičkih jednačina. Naravno, prvi nivo pogodan je za opis složenijih kola i sistema. S obzirom da je VHDL i razvijan sa ciljem da se efikasno opišu složena kola, rezultujući kôd karakteriše jezgrovitost koja proističe iz hijerarhijskog pristupa dekompoziciji (razlaganje na manje celine) projekta.

U VHDL-u mogu da se koriste tri stila opisa kola. To su:

- strukturni opis (engl. *structural*)
- opis toka podataka (engl. *dataflow*) i
- opis ponašanja (engl. *behavioral*).

Da bismo ilustrovali razliku između pojedinih stilova opisa razmotrićemo primer kola sa sl. 3.5. Svaki opis počinje deklarisanjem entiteta. U ovom slučaju entitet sa sl. 3.5.a nazvali smo `Primer_Stil_opisa`. Njegova deklaracija ne zavisi od stila opisa arhitekture, pa je data kao poseban primer 3.12.

Sada se posvećujemo opisu arhitekture ovog entiteta. Najpre pretpostavimo da je arhitektura na nivou logičkih blokova poznata i da izgleda upravo kao što prikazuje sl. 3.5b. Takav grafički opis se može dobiti primenom nekog od grafičkih editora logičke šeme. To podrazumeva da već postoji definisana komponenta koja obavlja funkciju XOR (prema tablici sa



Slika 3.5. a) Entitet `Primer_Stil_opisa`, b) struktura njegove arhitekture c) XOR2, d) tablica istinitosti XOR2 kola

sl. 3.5d). Pretpostavimo da je deklarisana sa nazivom XOR2, da ulazni portovi nose formalne nazive `I1` i `I2`, dok je izlazni port `Y`, kao što pokazuje sl. 3.5c.

Primer 3.12

```
entity Primer_Stil_opisa is
 port
( a, b, c: in std_logic;
  Rezultat: out std_logic
);
end entity Primer_Stil_opisa;
```

Važno je napomenuti da sem portova, unutar arhitekture postoji jedan novi signal, označen na sl. 3.5b kao `o1`. S obzirom da tip ovog signala do sada

nije bio deklarisan, on mora da se definiše u okviru arhitekture (videti primer opšteg oblika definisanja arhitekture iz odeljka 3.1.1), a zatim sledi opis arhitekture koji mnogo podseća na net listu koja se sreće u mnogim programima za logičku simulaciju (npr. *SPICE*).

Primer 3.13

```
architecture Strukturni_opis of
             Primer_Stil_opisa is
  signal o1: std_logic; --deklaracija internih
         --signala

begin
  u1:  xor2 port map (  a => I1,   -- )
       b => I2,     -- )
       o1 => Y);       -- )
  u2:  xor2 port map ( o1 => I1,   -- } telo
       c => I2,     -- )
       Rezultat => Y);    -- )
end architecture Strukturni_opis;
```

U prethodnom primeru prvi put je upotrebljena "strelica udesno" „=>“ kojom se stvarne vrednosti signala (`a,b,c` i `o1`) dodeljuju formalnim portovima `XOR2` kola (`I1`, `I2` i `Y`).

Opis iste funkcije entiteta `Primer_Stil_opisa` na nivou toka podataka (engl. *data-flow*) dat je u primeru 3.14.

Primer 3.14

```
architecture Tok_podataka of Primer_Stil_opisa is
signal o1: std_logic;   -- deklaracija
         -- internog signala
```

```
begin
o1 <= I1 XOR I2;
I1 <= a;
I2 <= b;
Rezultat <= o1 XOR c;

end architecture Tok_podataka;
```

Očigledno je primenjen pristup konkurentne dodele vrednosti signalima, tako da redosled navođenja pojedinih naredbi nije bitan, s obzirom da se one izvršavaju paralelno (konkurentno), kao što je rečeno u odeljku 3.1.5.

Opis ponašanja ne razlikuje se mnogo od opisa toka podataka, naročito kada su u pitanju manja kola, i zasniva se na algoritamskom opisu bloka. Pri tome se za opisivanje sekvencijalnih aktivnosti koristi `process` (kao u odeljku 3.1.5). Takav opis iste arhitekture je dat u primeru 3.15.

Primer 3.15

```
architecture Opis_Ponasanja of Primer_Stil_opisa
is

begin

XOR_od_3: process (a, b, c)  -- davanje imena
             -- procesu nije neophodno,
            -- ali je korisno
begin
if((a XOR b XOR c) = '1') then
  Rezultat <= '1';
  else
  Rezultat <= '0';
end if;
end process XOR_od_3;

end architecture Opis_Ponasanja;
```

Podsećamo da se vrednosti tipa *std_logic* odnose na bit, a ne na vektor. Njihova eksplicitna dodela signalima zadaje se između jednostrukih navoda kao '1', '0' u prethodnom primeru, ali mogu biti i 'X', 'Z' ili neka druga vrednost iz tabele 3.1.

Često se prilikom opisa složenijih projekata kombinuju sva tri stila opisa arhitekture. Za VHDL opis osnovnih digitalnih struktura u ovoj knjizi koristiće se stilovi opisa ponašanja ili toka podataka. Zato će se, u daljem tekstu, strukturni opis koristiti samo za testiranje u okviru opcije koja se na engleskom zove „*Test-Bench*", što bi u prevodu značilo „sto za testiranje". Ona će detaljnije biti opisana u okviru odeljka 3.3.

3.2.1. Prvi projekat

Definisaćemo naš prvi zadatak funkcionalnim opisom stanja datim u tabeli 3.6. Oznake stanja odgovaraju onima iz std_logic seta datog u tabeli 3.1. Signali A, B i C su ulazni, dok Y označava izlazni signal.

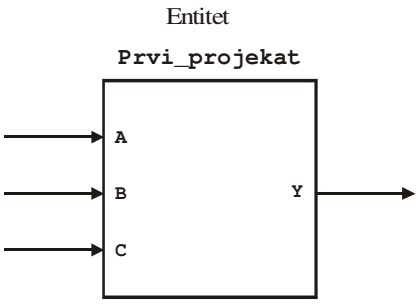
Dakle, zadatak je projektovati kolo na čijem će se izlazu javiti logička jedinica samo ukoliko su signali na ulazima A i B istovremeno na niskom logičkom nivou ili ukoliko je signal na ulazu C u stanju logičke jedinice. Ovako definisan zadatak implicira da kolo ima tri ulazna i jedan izlazni signal tipa std_logic.

Tabela 3.6. Tablica istinitosti opisa stanja u prvom projektu

A	B	C	Y
0	0	-	1
-	-	1	1
Ostale kombinacije			0

Definisanje entiteta

Ceo logički blok može da se predstavi grafički (sl. 3.6) na isti način kao i entitet iz primera Primer_Stil_opisa sa sl. 3.5. Savremeni editori VHDL kôda omogućavaju da se grafički predstavi entitet kome se definišu portovi. Tada se automatski generiše kôd definicije entiteta. Dakle, opis entiteta (primer 3.16) koji ćemo nazvati Prvi_projekat liči na onaj iz poglavlja 3.2.



Slika 3.6. Prvi projekat prikazan u obliku bloka

Primer 3.16

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

entity Prvi_projekat is port
( A, B, C : in std_logic;
      Y: out std_logic );
end entity Prvi_projekat;
```

Definisanje arhitekture

Kao što smo videli u poglavlju 3.2, opis arhitekture može da se bazira na **toku podataka**, na opisu **ponašanja** i opisu **strukture**. Za jednostavna kombinaciona kola, najkompaktniji oblik definisanja arhitekture daje opis na bazi toka podataka koji koristi konkurentne naredbe o kojima je bilo reči ranije u odeljku 3.1.6. Primena ovog tipa opisa ilustrovana je u Primeru 3.17.

Primećujemo da se koristi uslovna naredba u kojoj ključna reč ***when*** (treba je čitati „kada je“) definiše uslove pod kojima Y uzima vrednost logičke jedinice, dok reč ***else*** („drugo/inače“) ukazuje na vrednost koju Y dobija u ostalim slučajevima. Dodeljivanje vrednosti signalima (“strelicom ulevo“ <=) ukazuje na tok dodeljivanja podataka, odakle potiče i naziv ovakvom načinu opisa. Dodeljivanje vrednosti navodi se između jednostrukih navodnika, s obzirom da se radi o tipu *bit*.

Primer 3.17

```
architecture Tok_podataka of Prvi_projekat is

begin
   Y <= '1' when (A = '0' and B = '0')
                    or (C = '1')
        else '0';
end architecture Tok_podataka;
```

Opis ponašanja zasniva se na opisu pojedinih procesa. U okviru grupe naredbi definisanih procesom, naredbe se izvršavaju sekvencijalno. Postoji nekoliko sekvencijalnih naredbi u VHDL-u. Jedna od najčešće korišćenih je ***if*** naredba. Ova naredba:

- koristi se da bi se ispitao određeni uslov pre izvršenja neke operacije (npr. dodeljivanja),
- može da ispita višestruke uslove ako se koristi **elsif,**
- završava se sa **end if** i
- treba je kompletirati sa **else** kako bi se iscrple sve mogućnosti iz **if** lanca.

Imajući u vidu sve navedene osobine **if** naredbe, zahtevana logička funkcija može da se iskaže na nivou opisa ponašanja na način prikazan u primeru 3.18.

Treba napomenuti da je korisno da početak opisa svakog projekta u VHDL-u, ili njegovih delova, ima odgovarajuće zaglavlje. U njemu treba, u vidu komentara, navesti sve relevantne podatke koji će omogućiti kasniju modifikaciju. To su, osim informacija o nameni i funkciji kôda, i podaci o nazivu projekta, autorima, vremenu kada je projekat rađen, kada je i zašto modifikovan. Ovo je ilustrovano u primeru 3.18. Savremeni VHDL editori obično automatski generišu zaglavlje sa datumom u koje korisnik unosi samo podatke o nazivu projekta i imenima autora kôda. (Naravno, tada je osnovni

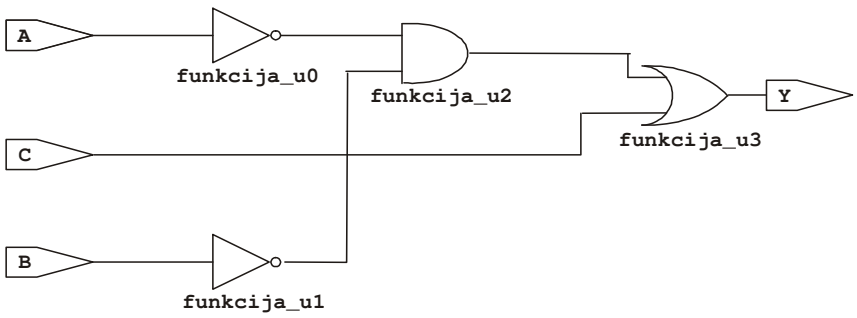
tekst zaglavlja na engleskom jeziku.) Zbog kompaktnosti opisa, ovakva zaglavlja neće biti ispisivana u narednim primerima VHDL opisa, ali će se podrazumevati da ona u opisima postoje.

Primer 3.18

```
-----------------------------------------------
-- Korisno je da VHDL opis ima zaglavlje
-- iz koga ce se videti:
-- Naziv         : Prvi_projekat
-- Projekat      : Prvi_projekat
-- Autor         : Predrag Petkovic
-- Kompanija     : LEDA laboratorija, EF - Nis
-----------------------------------------------
-- File          : Prvi_projekat.vhd
-- Datum       : Thu Feb 24 08:48:42 2015
-- kod pisao     : Bojan Andjelkovic
-----------------------------------------------
-- Kratak opis   :  Primena opisa ponasanja u
--        Prvom projektu
-----------------------------------------------
-- Modifikovao   : Miljana Sokolovic
-- Datum        : Thu Mar 24 09:40:41 2015
-- Opis         : Uneti komentari
-----------------------------------------------
library IEEE;  -- otvaranje biblioteke
use IEEE.STD_LOGIC_1164.all; -- poziv svih
          -- elemenata paketa
          -- STD_LOGIC_1164
entity Prvi_projekat is    -- Opis entiteta
 port(                     -- ulazni portovi:
                           -- a, b, c
   a,b,c : in STD_LOGIC;
   y : out STD_LOGIC    -- izlazni port: y
     );
end Prvi_projekat;       -- Kraj opisa entiteta
                    -- rec 'entity' iza 'end'
                    -- nije obavezna
```

```
--Pocetak opisa arhitekture sa nazivom
Opis_Ponasanja

architecture Opis_Ponasanja of Prvi_projekat is
begin
-- definisanje procesa sa imenom 'funkcija'
funkcija: process (a,b,c)
-- proces osetljiv na promenu signala a, b i c
begin
  if (a='0' and b='0') then
    y <= '1';
  elsif c = '1' then
    y <= '1';
  else
    y <= '0';
  end if;
end process funkcija; -- zavrsetak opisa procesa
end Opis_Ponasanja;
       -- zavrsetak opisa arhitekture
       -- rec 'architecture' iza 'end'
       -- je korisna, nije obavezna,
       -- olakšava tumacenje koda
```

Automatska sinteza koja proističe iz oba navedena načina opisa arhitekture (primer 3.17 i primer 3.18) našeg prvog projekta daje isti hardver, i on je prikazan na sl. 3.7.



Slika 3.7. Rezultat sinteze prvog projekta

Programi za automatsku sintezu prepoznaju VHDL instrukcije i dodeljuju im odgovarajući hardver. Na primer, ukoliko se prepozna deo kola u kome se javlja instrukcija NOT u okviru arhitekture, pridružuje se invertor. Ovo važi i za kompleksnije opise. Opis funkcije nekog kola u VHDL-u sličan je opisunekog zadatka iz programiranja. Naime, moguće je rešiti isti problem na različite načine, ali zavisno od kôdovanja programa neki od načina su više ili manje efikasni. Jedan opis može da rezultuje sa više ili manje složenim hardverom. Zato je izuzetno značajno da projektanti steknu osećaj za implikacije koje pojedina naredba ili grupa naredbi ima na automatski sintetizovani hardver. Ta veština zahteva detaljnije poznavanje primene VHDL-a za sintezu kola [Pet09].

3.3. VERIFIKACIJA PROJEKTA - TB

Najlakši način da se uverimo da li kolo korektno obavlja željenu funkciju jeste simulacija. VHDL, kao univerzalni jezik za opis projekta, podržava i verifikaciju. Pravilnije rečeno, programi za simulaciju prepoznaju VHDL, ne samo kao jezik za opis hardvera, već i za definisanje parametara neophodnih za simulaciju.

Uobičajeni način provere ispravnosti elektronskog kola jeste posmatranje odziva za datu pobudu na maketi za testiranje. Ovu aktivnost moguće je simulirati, i to uz pomoć para entitet-arhitektura koji se naziva ***T**est-**B**ench* (**TB**). TB mora da sadrži tri celine:

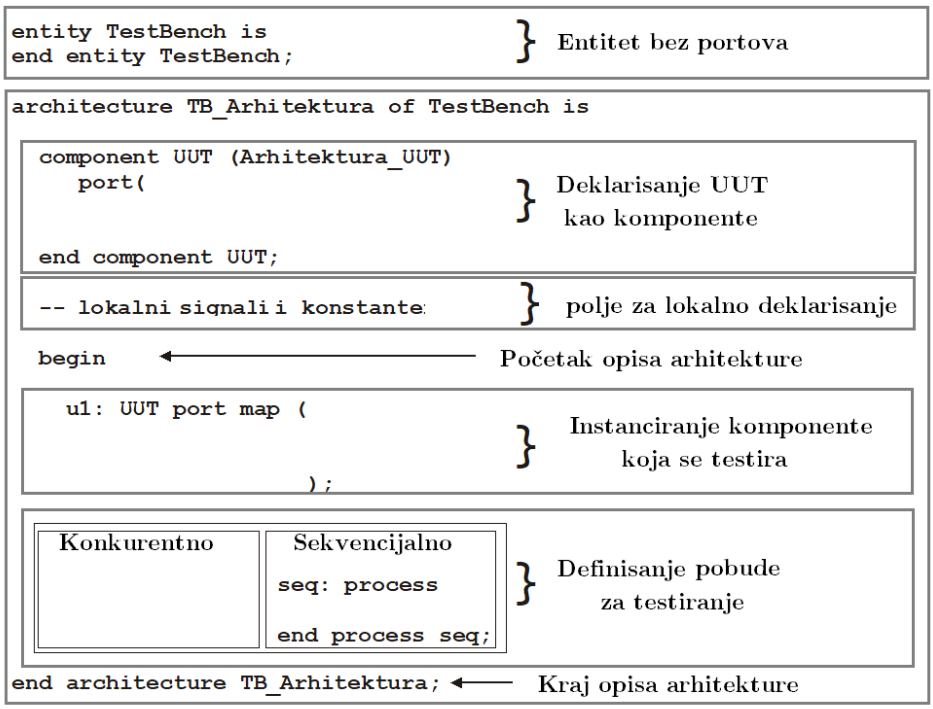
- komponentu koja se ispituje **UUT** (***U**nit **U**nder **T**est*),
- pobudu (generatori test sekvence) i
- monitor („instrument“ na kome se prati odziv).

S obzirom da su i generator pobude i monitor sastavni deo TB, očigledno je da se radi o jedinstvenom entitetu koji ne komunicira sa okolinom preko portova. Prema tome, saglasno definiciji entiteta, opis TB entiteta ne sadrži portove. S druge strane, UUT je posebna celina koja je prethodno već opisana entitetom i arhitekturom i kao takva se stavlja u probno kolo za verifikaciju. Sa pobudnim signalima i monitorom ona je vezana preko sopstvenih portova koji u odnosu na TB predstavljaju interne signale. Zato se arhitektura TB opisuje na strukturnom nivou. Podrazumeva se da je UUT već opisana kao par entitet-arhitektura, tako da je potrebno da se ona u TB arhitekturu unese i deklariše kao **komponenta**. Pri tome, nismo slučajno upotrebili reč komponenta. Naime, ***component*** je rezervisana reč koja označava entitet sa

pridruženom arhitekturom. Način deklarisanja UUT kao komponente opisan je u narednom odeljku.

Struktura opisa TB prikazana je na sl. 3.8. Očigledno je, da osim deklarisanja komponente, postoji potreba za deklarisanjem internih signala i konstanti. Logično je da blok u kome se obavljaju ove aktivnosti mora da prethodi opisu ostatka arhitekture, tako da je pozicioniran ispred ključne reči **`begin`**.

Ključnu komponentu u opisu arhitekture TB predstavlja predmet testiranja, odnosno UUT. Da bi se ona dodala arhitekturi TB, neophodno je da se stvarni signali „prikače" za nju preko portova UUT. Ovo se obavlja mapiranjem, odnosno dodelom stvarnih signala iz TB, signalima na portovima komponente. Zato se za ovu svrhu koristi **`port map`** instrukcija. Postupak priključivanja komponente u projektantskom žargonu naziva se **instanciranje** (engl. *instantiate*).



Slika 3.8. Struktura Test-Bench-a

Osim priključivanja komponente koja se testira, neophodno je definisati proces testiranja. On se sastoji od zadavanja pobude i tumačenja odziva i može da bude podeljen u više procesa ili čak komponenti. To se naročito preporučuje pri testiranju složenih kola.

Pobuda može da se zada na tri načina:

- posebnim editorima koji predstavljaju deo alata za projektovanje u okviru paketa za simulaciju koji podržavaju VHDL opis,
- kao tabela u okviru samog procesa i
- čitanjem posebnog tekstualnog fajla.

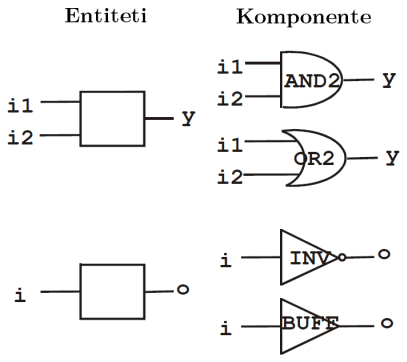
Odziv može da se prati na dva načina:

- posmatranjem talasnih oblika i
- zapisivanjem odziva (izlaznih signala) u fajl i poređenjem sa očekivanim rezultatom.

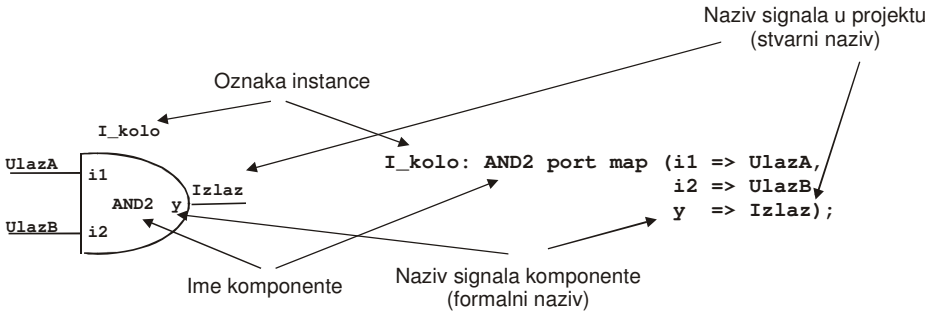
Kada se govori o verifikaciji projekta važno je napomenuti da se, za razliku od opisa namenjenog sintezi, u modelima za simulaciju javljaju i vremenski parametri. Oni, osim definisanja talasnih oblika pobude, obuhvataju modele kašnjenja signala na putu od ulaza do izlaza.

3.3.1. Definisanje UUT kao komponente

Razlika između entiteta i komponente uočava se sa sl. 3.9. Vidi se da entitet dobija značenje komponente tek kada mu je pridružena arhitektura. Tako AND2 i OR2 komponente imaju identičnu topologiju entiteta, ali im se



Slika 3.9. Razlika između entiteta i komponente



Slika 3.10. Povezivanja komponente za signale UlazA, UlazB i Izlaz

funkcije u kolu razlikuju, jer su im arhitekture različite. Ista konstatacija važi i za invertor INV i bafer BUFF. Komponenta mora da ima isto ime kao i entitet, a nazivi, modovi i tipovi portova u komponentama treba da budu isti kao u entitetima.

Korišćenje komponenata karakteristično je za VHDL opis na strukturnom nivou koji nije glavna tema ovog teksta. Zato ćemo samo kratko reći da jednom deklarisane komponente mogu da se koriste više puta u okviru projekta, vezivanjem za odgovarajuće signale (čvorove) u strukturnoj šemi. Povezivanje se obavlja primenom instrukcije **`port map`**. U opštem slučaju sintaksa povezivanja neke komponente ilustrovana je na primeru dvoulaznog I kola na Sl. 3.10.

Dvoulazno I kolo sa sl.3.10, deklarisano je kao komponenta pod nazivom `AND2`, sa ulaznim portovima `i1` i `i2` i izlaznim portom, `y`.

U arhitekturi u koju se smešta, ovoj komponenti je dodeljeno stvarno ime „`I_kolo`". Za njene portove vezani su stvarni signali sa imenima `UlazA`, `UlazB i Izlaz`.

U našem slučaju komponentu predstavlja entitet `Prvi_projekat` sa odgovarajućom arhitekturom. Zato se za njeno deklarisanje na početku opisa arhitekture TB koristi opis:

```
component Prvi_projekat
  port (A, B, C: in std_logic;
        Y: out std_logic);
end component;
```

U odeljku 3.2.2 definisali smo dve arhitekture uz entitet `Prvi_projekat`. Zato se odmah nameće pitanje, na koju se arhitekturu

odnosi ova komponenta. Da bi se jednoznačno odredio par entitet/arhitektura koji se koristi u opisu TB arhitekture, posle naredbe **end architecture** komponenta se konfiguriše korišćenjem bloka instrukcija **configuration**. Opšti oblik ovog bloka definisan je sa:

```
configuration <identifikator> of <ime_entiteta > is
  for <ime_arhitekture>
-- telo konfiguracionog bloka
    for Comp_1: entitetA
  use entity work.entitetA(arhitekturaA1);
    end for;
    for Comp_2: entitetA
  use entity work.entitetA(arhitekturaA2);
    end for;
    for Comp_3: entitetB
  use entity work.entitetB(arhitekturaB1);
    end for;
  end for;
end configuration <identifikator>;
```

Konfigurisanje koristi rezervisanu reč **for**, koja označava „za“. Svaki iskaz **for** treba završiti sa **end for;** Očigledno je da se **configuration** blok odnosi na entitet: „**of** <ime_entiteta>” za koji se definiše arhitektura: „**for** <ime_arhitekture>”. U okviru ove arhitekture konfigurišu se odgovarajući parovi entitet/arhitektura koji su prethodno smešteni u radnoj biblioteci. Pri tome se entitetu dodeljuje arhitektura tako što se ime arhitekture navodi između malih zagrada kao: **work.** <naziv_entiteta> (<naziv_arhitekture>). U navedenom primeru konfiguracije vidi se da se koriste tri komponente. Dve (Comp_1 i Comp_2) imaju isti entitet entitetA, ali Comp_1 koristi arhitekturu arhitekturaA1: entitetA(arhitekturaA1); a Comp_2 arhitekturu arhitekturaA2: entitetA(arhitekturaA2);. Komponenta Comp_3 konfigurisana je kao entitet entitetB sa arhitekturom arhitekturaB1, odnosno: entitetB(arhitekturaB1);.

Primer konfiguracionog bloka za TB kojim se verifikuje Prvi_projekat sa arhitekturom Protok_podataka:

```
configuration TB_ZA_Prvi_projekat of TestBench is
  for TB_Arhitektura
    for UUT : Prvi_projekat
     use entity work.Prvi_projekat(Protok_podataka);
```

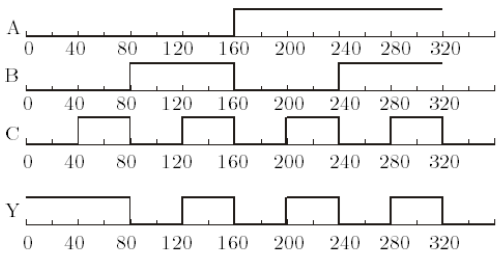
```
    end for;
  end for;
end TB_ZA_Prvi_projekat;
```

3.3.2. Definisanje pobude i praćenje odziva

Za manje projekte najčešće se pobuda definiše primenom ugrađenih editora signala. Realni digitalni sistemi pobuđuju se periodičnim i aperiodičnim signalima. Periodični su taktni signali koji imaju jasno definisanu frekvenciju (periodu) i simetrično trajanje visokog i niskog logičkog nivoa. Ostali pobudni signali mogu da budu periodični sa asimetričnim trajanjem visokog i niskog logičkog nivoa ili potpuno aperiodični. Editori signala podržavaju zadavanje parametara svih tipova pobude. Kada su u pitanju periodični signali, moguće je zadati periodu, dužinu trajanja visokog i niskog nivoa i fazu (kašnjenje). Kod aperiodičnih, zadaju se podaci o trenutku promene i logičkom stanju signala u tom trenutku. Način zadavanja pobude u različitim alatima razlikuje se samo u grafičkom interfejsu prema korisniku.

Definisanje pobude primenom VHDL-a u obliku procesa ili čitanjem sadržaja testne sekvence iz fajla prevazilazi ambiciju ovog poglavlja. Zato ćemo samo zagolicati znatiželju zainteresovanim čitaocima kroz primer 3.19 na kraju odeljka, a za detalje upućujemo na VHDL korisnička uputstva i literaturu koja se ozbiljnije bavi ovom temom [Pet05].

Ukoliko povorka izlaznih signala nije velika, najbrži način provere tačnosti odziva jeste posmatranje talasnih oblika signala. Naime, svi alati za simulaciju omogućavaju vizuelizaciju rezultata simulacije kroz prikazivanje talasnih oblika signala. Međutim, u slučajevima testiranja složenijih kola, kada su povorke signala duge i obimne, mnogo je povoljnije da se informacija o odzivu upisuje u izlazni fajl. VHDL omogućava da se očitava sadržaj fajlova sa podacima zadatim u vidu alfanumeričkih podataka. Poređenjem podataka smeštenih u datoteci sa očekivanim rezultatom i podataka dobijenih simulacijom, utvrđuje se ispravnost rada kola. Postoje posebne konstrukcije namenjene da generišu poruke upozorenja kada se otkrije odstupanje od očekivanog rezultata. Kod složenih kola jako je korisno da se takve poruke upisuju u fajl tako da njihov sastavni deo bude i tačan trenutak na vremenskoj osi u kome je zapaženo odstupanje [Pet09].



Slika 3.11. Vremenski dijagram pobudnih signala (A, B, C) i očekivanog odziva (Y) kola opisanog kao `Prvi_projekat`

Talasni oblici pobude na ulazima A, B i C koji nose stvarna imena signala `A`, `B` i `C`, i očekivanog odziva u na izlazu `Y`, u slučaju provere UUT sa nazivom `Prvi_projekat` definisani su na sl. 3.11. Vremena na sl. 3.11 data su u nanosekundama.

Da bi se ispitao odziv kola `Prvi_projekat` za sve moguće kombinacije ulaznih signala, potrebno je da se kolo pobudi sa tačno 2^3, odnosno osam kombinacija koje su prikazane u tabeli 3.7.

Tabela 3.7. Potpuni skup kombinacija kojim se testira funkcionalnost za `Prvi_projekat`.

A	B	C	Y
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

S obzirom da je testni vektor relativno kratak (samo 8 kombinacija), nije teško da se sve kombinacije i navedu. Zato ćemo u narednom primeru da ilustrujemo zadavanje pobude upotrebom naredbe **`process`**. Za opis pobude koristićemo proces sa oznakom `pobuda` kao u primeru 3.19. Istovremeno,

demonstriraćemo još nekoliko praktičnih opcija koje nude VHDL i odgovarajući alati za kreiranje i verifikaciju kôda. To su:

a) upotreba `Active-HDL TBgen` alata za automatsko generisanje TB,
b) pristup pojedinim bitovima podatka tipa `std_logic_vector`,
c) deklarisanje i tipovi konstanti i
d) upotreba naredbe **`wait`**.

Svaka od njih biće objašnjena u tekstu koji sledi posle primera.

Primer 3.19

```
--************************************************
--* This file is automatically generated test bench
template*
--* By ACTIVE-VHDL <TBgen v1.10>. Copyright (C)
ALDEC Inc.  *
--*                                                *
--* This file was generated on: 7:05 PM, 11/30/01
*
--* Tested entity name:             Prvi_projekat
*
--* File name contains tested entity:       *
-- $DSN\src\prviprojekatstruct.vhd            *
--
************************************************

library ieee;
use ieee.std_logic_1164.all;
-- Add your code here ...
-- ovde se otvaraju potrebne biblioteke sa use
```

```
entity tb_wizard is
end tb_wizard;

architecture TB_WIZARD_ARCH of tb_wizard is
-- Component declaration of the tested unit
component Prvi_projekat
port( A, B, C : in std_logic;
        Y : out std_logic );
```

```
end component;

-- Stimulus signals - signals mapped to the input
and inout
-- ports of tested entity
signal A : std_logic;
signal B : std_logic;
signal C : std_logic;
-- Observed signals - signals mapped to the output
ports of
-- tested entity
signal Y : std_logic;

-- Add your code here ...
-- definišu se interni signali i konstante
   signal ulaz: std_logic_vector (2 downto 0);
   signal izlaz: std_logic;

   constant Kasnjenje: time := 40 ns;

begin

-- Unit Under Test port map
UUT : Prvi_projekat
  port map
     (A => ulaz(2),
      B => ulaz(1),
      C => ulaz(0),
      Y => Y );

-- Add your stimulus here ...
pobuda: process

begin
ulaz <= "000"; wait for Kasnjenje;
ulaz <= "001"; wait for Kasnjenje;
ulaz <= "010"; wait for Kasnjenje;
ulaz <= "011"; wait for Kasnjenje;
ulaz <= "100"; wait for Kasnjenje;
ulaz <= "101"; wait for Kasnjenje;
ulaz <= "110"; wait for Kasnjenje;
ulaz <= "111"; wait for Kasnjenje;

end process pobuda;

end TB_WIZARD_ARCH;
```

```
configuration    TESTBENCH_FOR_Prvi_projekat    of
tb_wizard is
for TB_WIZARD_ARCH
   for UUT : Prvi_projekat
    use entity
work.Prvi_projekat(Opis_ponašanja);
   end for;
end for;
end TESTBENCH_FOR_Prvi_projekat;
```

---

a) Namerno je zasenčen deo kôda kako bi se naglasilo da je ubačen primenom alata za automatsko generisanje TB koji imaju gotovo svi VHDL editori. Tamo gde postoji potreba da korisnik doda svoj kôd, nalazi se odgovarajući komentar:

`-- Add your code here` ili `-- Add your stimulus here.`

Ponovo skrećemo pažnju na početni deo kôda – zaglavlje. Kao što je već naglašeno, veoma je korisno da se u zaglavlju nađu sve relevantne činjenice o kôdu: ko je autor, kada je kôd napisan, na šta se odnosi, kako se zove fajl i gde se nalazi na disku. Ukoliko se naknadno unose izmene u kôd, treba napisati zašto je, kada i ko modifikovao fajl. U narednim primerima neće se prikazivati zaglavlje (kako bi se uštedelo na prostoru) ali to ne znači da zaglavlje treba izostavljati.

Očigledno je da alat za automatsko generisanje TB u velikoj meri rasterećuje projektanta od unosa podataka koji su već opisani, kao što su pristup standardnim bibliotekama i paketima, definisanje entiteta, deklarisanje UUT kao komponente, priključivanje komponente arhitekturi TB i konfigurisanje komponente koja se testira. Naravno, korisnik može da izmeni, a mora i da dopuni, ponuđeni kôd.

b) Drugu novinu u ovom primeru predstavlja opis načina na koji se pristupa pojedinim bitovima signala tipa `std_logic_vector` (a pravilo važi i za ostale vektore). Naime, za definisanje pobude uveden je trobitni signal `ulaz` tipa `std_logic_vector(2 downto 0)`. Ulaznom signalu `A` dodeljen je MSB signala `ulaz`, signalu `B` dodeljen je srednji, a signalu `C` dodeljen je LSB: (`A => ulaz(2); B=>ulaz(1); C=>ulaz(0)`). Uvođenjem vektora `ulaz` olakšan je opis pobudnih signala time što se vrednosti za

sva tri signala dodeljuju jednom naredbom: `ulaz <= "000"`. Iz ovog primera treba uočiti dve stvari koje se tiču upotrebe podataka deklarisanih kao vektori:

- k-ti član vektora `ulaz` izdvaja se jednostavno pisanjem `ulaz(k);`
- prethodno važi i za izdvajanje dela vektora od *k*-tog do *m*-tog bita, gde je *k-m=n>0*, kao:
  `vek1(n-1 downto 0) => vek(k downto m)`
- dodeljivanje vrednosti vektoru zahteva da se umesto jednostrukih (koji se koriste za dodeljivanje vrednosti bitu), koriste dvostruki navodnici: `ulaz <= "000";` .

Naravno, podrazumeva se, kao i kod drugih programskih jezika, da se mora voditi računa o poklapanju dužine vektora sa deklarisanom dužinom, kao i o poklapanju tipova promenljivih (ne mešati promenljive različitih tipova).

c) Treća novina uvedena kroz ovaj primer jeste definisanje pojma **`constant`**. Radi se, zapravo o konstantnoj vrednosti nekog parametra koji se prosleđuje celoj arhitekturi (kada se definiše u deklaracionom delu arhitekture). Pogodnost primene konstanti ogleda se u olakšanom korigovanju vrednosti parametara koji se višestruko pojavljuju u opisu. Ukoliko je vrednost parametra definisana kao konstanta, korekcija se radi ispravkom u samo jednom redu. Sintaksa deklarisanja konstante je, kao što se iz primera može zaključiti, sledeća:

```
constant <ime>: <tip> := <vrednost>;
```

Tipovi konstanti definisani su u svim detaljnim VHDL priručnicima. Ovde ćemo pomenuti samo neke koji se najčešće sreću: `integer`, `time` (vreme), `std_logic`, `std_logic_vector` itd... Treba napomenuti da korisnici mogu deklarisati sopstvene tipove, kao što su matrice sa različitim tipovima podataka, ali se to preporučuje naprednijim korisnicima. Konstante se najčešće koriste za definisanje dimenzija vektora (nizova), u petljama za kontrolisanje broja ponavljanja i za definisanje vremenskih parametara (kao što je slučaj u ovom primeru). Naglasimo da se one *ne koriste za definisanje dimenzija portova*, jer za to služe opšte generičke konstante koje se deklarišu u okviru entiteta, a ne arhitekture. Kao što je to već istaknuto, *opšte konstante karakteriše ključna reč* **`generic`**. Koriste se za specifikaciju parametara koji definišu opšte osobine i entiteta i pripadajućih arhitektura.

Dobar primer predstavlja opis portova sa n bitova. Da bi se jednim opisom obuhvatile komponente sa istom funkcijom ali proizvoljnim brojem bitova, onda se n zadaje kao generik sa nekom dodeljenom vrednošću koja se može menjati prilikom upotrebe (instanciranja) te komponente u opisu konkretnog projekta. S obzirom da je n ceo broj, on se mora, prethodno, deklarisati kao tip `integer`.

Način korišćenja opšte konstante ilustrovan je u primeru 3.20.

Primer 3.20

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

entity Primer_generic is
generic (
  na: integer := 8;  -- definisanje
                     -- broja bitova porta a
   nb: integer := 4); -- definisanje broja
                      -- bitova porta b
port ( A: in std_logic_vector ((na-1) downto 0);
B: out std_logic_vector ((nb-1) downto 0);
 end entity Primer_generic;
```

Vidimo da se opšte konstante (`na` i `nb`) mogu koristiti neposredno posle deklarisanja (već u opisu portova). Njihova vrednost prenosi se i ka arhitekturi, tako da ih ne treba tamo deklarisati ponovo.

Ukoliko je komponenta opisana entitetom u kome su definisane opšte konstante (**`generic`**), prilikom priključivanja komponente projektu, osim instrukcije **`port map`**, kojom se dodeljuju stvarne oznake formalnim portovima, koristi se i **`generic map`** instrukcija. Njena primena ilustrovana je na primeru 3.21 koji prikazuje deo koda u kome se komponenta `Primer_generic` (primer 3.17) priključuje nekom projektu pod nazivom `U1`.

Primer 3.21

```
U1 : Primer_generic
generic map(na => 16, nb => 8)
port map (A => ulaz,
          B => izlaz );
```

Skrećemo pažnju da se prilikom priključivanja komponente projektu (instanciranje), konstantama **generic** mogu dodeliti nove vrednosti (različite od onih u deklaraciji entiteta). Dodela se obavlja simbolom **=>**.

d) U primeru 3.19 koristi se naredba **wait for** kojom se proces zaustavlja za neki vremenski interval. Generalno, naredbom **wait** proces može biti zaustavljen bezuslovno, ili uz određeni uslov koji se navodi iza ključnih reči **for**, **on**, **until** koje označavaju uslov.

Konfiguracijom u poslednjem delu opisa primera 3.19 definiše se arhitektura koju hoćemo da proverimo:

```
for UUT : Prvi_projekat use entity

work.Prvi_projekat(Opis_ponašanja);
```

Za proveru sintetizovane arhitekture `Tok_podataka` bilo bi dovoljno navesti:

```
for UUT : Prvi_projekat use entity

    work.Prvi_projekat(Tok_podataka);
```

Na ovaj način mogu se testirati arhitekture, za sintezu (bez vremenskih parametara), ili arhitekture dobijene posle fizičkog projektovanja, pri čemu se koristi isti TB opis u oba slučaja.

3.4. OSNOVNA ARITMETIČKA KOLA

3.4.1. Sabirači

Logička funkcija jednobitnog potpunog sabirača definisana je tablicom istinitosti (Tabela 3.8.), a njegov VHDL opis dat je u primeru 3.22.

Tabela 3.8. Tablica istinitosti potpunog sabirača

cin	a	b	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

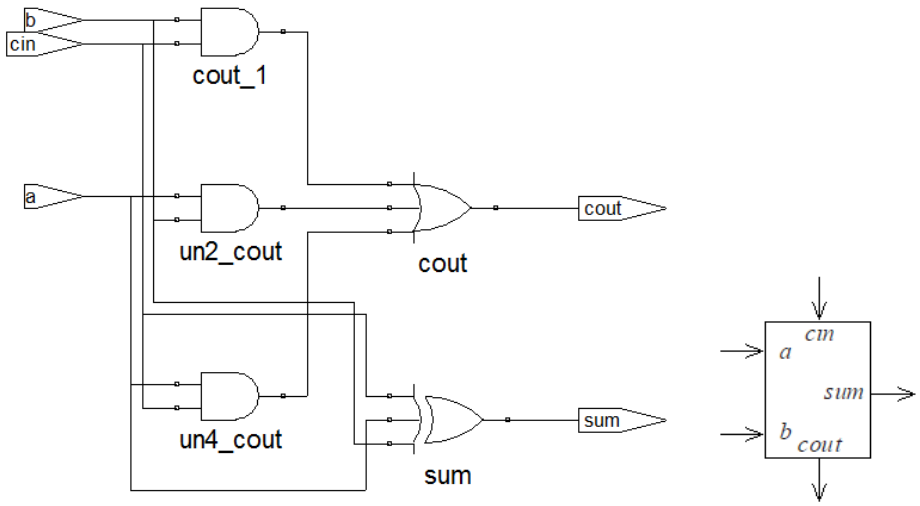
Primer 3.22

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


entity PotpuniSabirac1 is
port (
   a: in STD_LOGIC;
   b: in STD_LOGIC;
   cin: in STD_LOGIC;
   sum: out STD_LOGIC;
   cout: out STD_LOGIC
);
end PotpuniSabirac1;
```

```
architecture PotpuniSabirac1 of PotpuniSabirac1 is
begin
  -- <<enter your statements here>>
  sum <= ((a XOR b) XOR cin);
  cout <= (a AND b) OR (a AND cin)
                    OR (b AND cin);

end PotpuniSabirac1;
```

Šema hardverske realizacije koja proističe iz opisa datog u primeru 3.22, prikazana na sl. 3.12.a. Blok šema potpunog jednobitnog sabirača prikazana je na sl. 3.12.b.

Alternativno, višebitni sabirač može da se opiše kao n-bitni, na način koji pokazuje primer 3.23.



a) b)

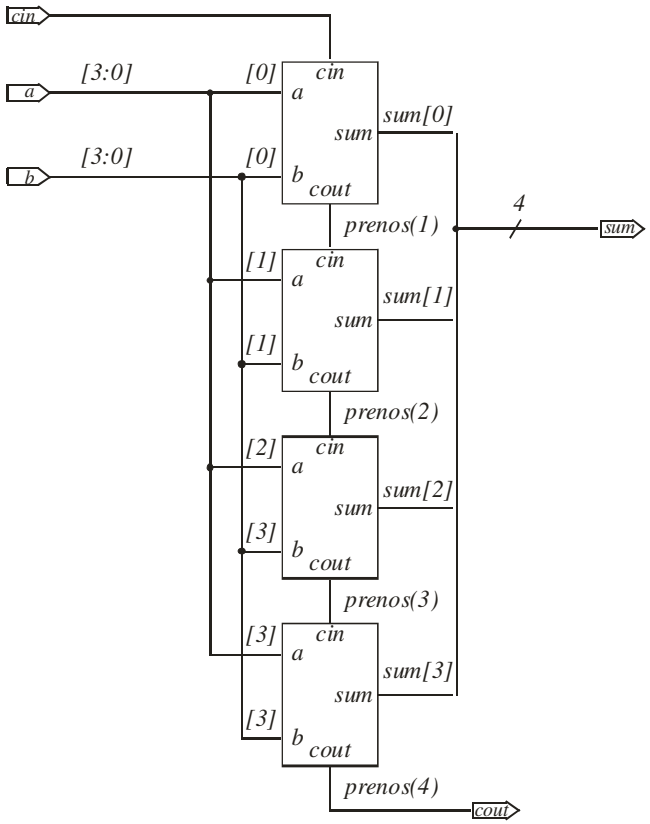
Slika 3.12. Jednobitni potpuni sabirač: a) primer 3.22 i b) blok šema

Primer 3.23

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

entity PotpuniSabiracN is
  generic ( n: integer := 4);
  port (
   a: in STD_LOGIC_VECTOR (n-1 downto 0);
                              -- sabirak1
   b: in STD_LOGIC_VECTOR (n-1 downto 0);
                              -- sabirak2
   cin: in STD_LOGIC;         -- bit prenosa
   sum: out STD_LOGIC_VECTOR (n-1 downto 0);
                              -- zbir
   cout: out STD_LOGIC );     -- prenos na izlazu
);
end PotpuniSabiracN;

architecture PotpuniSabiracN of PotpuniSabiracN is
begin
 -- <<enter your statements here>>
Zbir: process (a, b, cin)
 variable
  Prenos: STD_LOGIC_VECTOR (n downto 0);
 variable
  LokalniZbir: STD_LOGIC_VECTOR (n-1 downto 0);
  begin
   Prenos (0) := cin;
   for i in 0 to n - 1 loop
    LokalniZbir(i) := (a(i) XOR b(i))
                        XOR Prenos(i);
    Prenos (i+1) := (a(i) AND b(i))
                 OR (Prenos(i)
                           AND (a(i) OR b(i))
                     );
end loop;

sum <= LokalniZbir;
cout <= Prenos(n);
  end process Zbir;

end PotpuniSabiracN;
```

Konkretno, prikazan je slučaj za n=4, odnosno, ilustrovan je opis četvorobitnog sabirača. Kao u prethodnom primeru, broj bitova deklarisan je, prethodno, kao celobrojna opšta konstanta `n` kojoj je dodeljena vrednost 4. Hardverska realizacija koja proističe iz ovog opisa prikazana je na sl. 3.13.

Opis iz primera 3.23 zasnovan je na činjenici da se ponašanje višebitnog sabirača može opisati formulama koje su pogodne za primenu **`for`** petlje. Bilo je potrebno na početku definisati `Prenos(0)` i izjednačiti ga sa signalom prenosa `cin`. Takodje, znajući da se vrednosti signala dodeljuju na kraju procesa, jasno je da će globalni prenos iz celog sabirača `cout` biti jednak prenosu najvišeg bita (MSB) na izlasku iz petlje. Zato je potrebno da variabla `Prenos` bude za jedan bit veća od "širine" sabirača.



Slika 3.13. Realizacija n-bitnog potpunog sabirača (primer 3.23, n=4)

Osim toga, primena opšte konstante `generic` dopušta da se isti opis koristi za sabirače proizvoljne dimenzije. Još jednom treba napomenuti da primena opštih konstanti olakšava višestruko korišćenje jednom opisanih komponenata kod kojih je osnovna funkcija ista, a menja se samo broj bitova nad kojim se primenjuju.

Primer 3.23 pokazuje kako može da se opiše n-bitni sabirač na relativno niskom nivou apstrakcije. Zapravo, sve vreme koristimo samo logičke operatore AND, OR, XOR, da bismo opisali hardver koji obavlja aritmetičku operaciju sabiranja. Pri tome, ne treba izgubiti iz vida da se logički operatori primenjuju nad promenljivama koje su tipa `std_logic`, odnosno mogu imati jednu od 9 vrednosti navedenih u tabeli 3.1. U složenim projektima mnogo je korisnije i preglednije da se takav hardver opiše na višem nivou. Konkretno, to bi značilo da se koristi simbol + kao operator za oznaku sabiranja. Medjutim, u skupu `std_logic` ne postoje elementi kojima bi mogao da se dodeli rezultat operacije +. Tek za vektore tipa `std_logic_vector` mogli bismo da govorimo o značenju, odnosno vrednosti, koju ima signal " 1010 ".

Realno, tip `std_logic_vector` predstavlja tek niz elemenata tipa `std_logic` i nema numeričku vrednost. Zato su, kao što je rečeno u odeljku 3.1.3, uvedeni drugi tipovi vektora sastavljeni od elemenata `std_logic`, kojima se može dodeliti vrednost. To su tipovi ***signed*** i ***unsigned***. Ovi tipovi mogu da budu interpretirani kao prirodni brojevi, pozitivni, bez znaka ***unsigned*** ili kao ***signed***, odnosno sa znakom, tako da mogu da budu i negativni. Oni mogu da se prevode u intedžere nad kojima se obavljaju aritmetičke operacije. Na primer, vektor dužine četiri, tipa `std_logic`**,** koji ima vrednost 1010 biće protumačen kao dekadna cifra 10 ukoliko je deklarisan kao ***unsigned***. Međutim, istom signalu dodeljuje se značenje negativnog broja (-6) ukoliko je daklarisan kao ***signed***. U projektima na RTL nivou pojavljuju se samo 'pozitivne' vrednosti iz `std_logic` seta, tako da za njihov opis treba koristiti ***unsigned*** tipove.

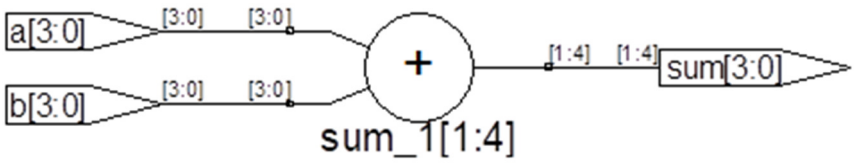
VHDL ima ugrađene aritmetičke funkcije koje podržavaju promenljive tipa intedžer. Medjutim, u najvećem broju projekata i modelima namenjenim sintezi, najčešće se koriste promenljive tipa `std_logic`. Operatori koji se odnose na tip `std_logic`, odnosno na vektore ***unsigned*** i ***signed*** sačinjene od tipova `std_logic`, smešteni su u paketima:

- `numeric_std ,`
- `std_logic_signed ,`
- `std_logic_unsigned i`
- `std_logic_arith .`

Ovi paketi najčešće se nalaze u IEEE biblioteci, ali oni NISU standardni deo ovo biblioteke. Da bi se portabilnost projekta povećala, **preporučuje** se korišćenje `numeric_std` paketa. Zato nećemo ni da ulazimo u detalje ostalih paketa, ali se naprednim korisnicima VHDL-a savetuje da informacije o njima potraže u literaturi. Paket `numeric_std` obuhvata i ***signed*** i ***unsigned*** `std_logic` tipove. Dakle, upotreba ovih vektora neophodna je kad god treba da se definiše neka operacija nad vektorima tipa `std_logic`, a samo na kraju, ako je to potrebno, na odgovarajućim portovima treba izvršiti konverziju u `std_logic_vector`. Uvodjenje operatora nad vektorima `std_logic_vector` značajno pojednostavljuje opis hardvera što je očigledno iz primera 3.24.

Primer 3.24

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;
entity Sabirac4 is
 generic ( n: integer :=4);
 port(
    a : in unsigned(3 downto 0);
    b : in unsigned(3 downto 0);
    sum : out unsigned(3 downto 0)
      );
end Sabirac4;


architecture Sabirac4 of Sabirac4 is
begin


 -- enter your statements here --

  sum <= a + b;

end Sabirac4;
```



Slika 3.14. Grafička interpretacija sabirača iz primera 3.24

U ovom primeru definisani su zbir (`sum`) i operandi `a` i `b` kao `unsigned`, čime se omogućava njihovo direktno sabiranje. Funkcija + vraća operand kod koga krajnji levi bit ima najveću težinu - MSB. Oduzimanje se obavlja na sličan način. Grafička interpretacija ovog sabirača prikazana je na sl. 3.14.

Primena unsigned tipa i + operatora u opisu potpunog n-bitnog sabirača koji bi odgovarao onom iz primera 3.23 data je u primeru 3.25.

Primer 3.25

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity PotpuniSabiracUN is
   generic (n: integer :=4);
   port (
         a: in STD_LOGIC_VECTOR (n-1 downto 0);
         b: in STD_LOGIC_VECTOR (n-1 downto 0);
         cin: in STD_LOGIC_VECTOR (0 downto 0);
         sum: out STD_LOGIC_VECTOR
                    (n-1 downto 0);
         cout: out STD_LOGIC
);
end PotpuniSabiracUN;

architecture PotpuniSabiracUN of
             PotpuniSabiracUN is
begin
  -- <<enter your statements here>>
  Zbir: process (a, b, cin)
    variable LokalSum: unsigned (n downto 0);
    variable LokalCin : unsigned (0 downto 0);
```

```
  begin
    LokalCin := unsigned(cin);
    LokalSum := unsigned('0' & a) +
                unsigned('0' & b);
    LokalSum := LokalSum + LokalCin;
    sum <=
    std_logic_vector( LokalSum(n-1 downto 0));
    cout <= std_logic(LokalSum(n));
  end process Zbir;
end PotpuniSabiracUN;
```

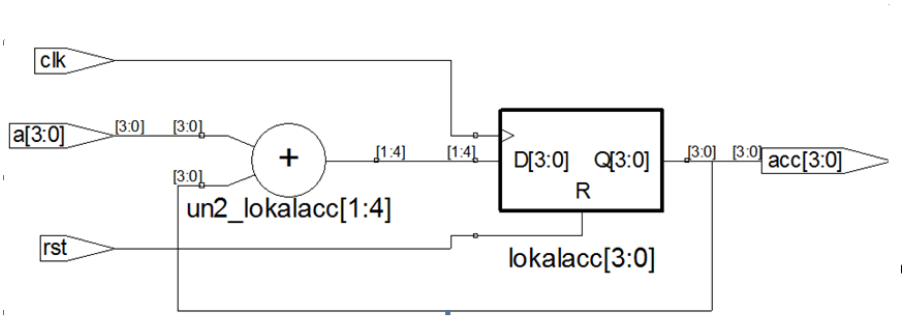
Iz ovog primera može da se nauči sledeće:

Svi portovi u entitetu definisani su kao `std_logic_vector`, odnosno `std_logic`. Da bi operacija sabiranja + mogla da se primeni, u procesu `zbir`, koriste se promenljive tipa **`unsigned`**. Konverzija iz tipa `std_logic_vector` u **`unsigned`** obavlja se funkcijom `unsigned()` pri čemu je argument tipa `std_logic_vector`. Podsećamo da je tip ***unsigned*** po prirodi vektor, zato direktna konverzija zahteva da se i `cin` predstavi kao `std_logic_vector` dužine 1 (`0 downto 0`). (Ovo će implicirati da se i pri zadavanju vrednosti ovom signalu tokom verifikacije koriste “ umesto ‘). Obrnuta konverzija neophodna je na kraju procesa, kako bi se izlazna reč prevela u željeni `std_logic` tip. Ona se obavlja funkcijom `std_logic_vector()`. Naravno, u ovom slučaju argument je tipa ***unsigned***, a funkcija vraća vrednost tipa `std_logic_vector`.

Da bi se izdvojio prenos `cout` iz promenljive `suma`, izdvaja se samo njen MSB. Očigledno, zbog prenosa, promenljiva `suma` mora da bude za jedan bit veća od operanada `a` i `b`. Zato je u operaciji **+**, u kojoj se istovremeno obavlja konverzija tipova `std_logic_vector` operanada `a` i `b` u ***unsigned***, bilo neophodno da se vektori `a` i `b` prošire sa leve strane dodavanjem 0, što je učinjeno operatorom za konkatenaciju (spajanje): `'0' & a`. (Dodavanje nule sa desne strane, što odgovara množenju sa 2, bilo bi `a & '0'`).

3.4.2. Akumulatori

Akumulatori su sabirači kod kojih je jedan od operanada jednak trenutnoj sumi na izlazu. Da bi se memorisalo prethodno stanje na izlazu, koriste se registri. Opis akumulatora n-bitnog signala (sa n=4) koji uvećava vrednost



Slika 3.15. Grafička interpretacija akumulatora iz primera 3.26

pri svakoj prednjoj ivici signala takta i asinhronim resetom dat je u primeru 3.26, a odgovarajuća hardverska realizacija prikazana je na slici 3.15. U ovom primeru obnovljena su iskustva koja smo stekli tokom tumačenja nekompletirane **if...then** strukture. Naime, ona obavlja funkciju implicitnog leča, odnosno memorisanje prethodnog stanja akumulatora.

Primer 3.26

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity AkumulatorN is
   generic (n: integer :=4);
   port (
      a: in STD_LOGIC_VECTOR (n-1 downto 0);
      clk: in STD_LOGIC;
      rst: in STD_LOGIC;
      acc: out STD_LOGIC_VECTOR (n-1 downto 0)
         );
end AkumulatorN;

architecture AkumulatorN of AkumulatorN is
signal LokalAcc: unsigned (n-1 downto 0);
begin
  -- <<enter your statements here>>
  Akumuliraj: process (clk, rst)
```

```
  begin
    if (rst = '1') then
      LokalAcc <= (others =>'0');
     elsif (clk'event and clk = '1') then
      LokalAcc <= LokalAcc + unsigned(a);
    end if;
  end process Akumuliraj;

  acc <= std_logic_vector(LokalAcc);
end AkumulatorN;
```

Skrećemo pažnju na način na koji se resetuje vektor čija je dimenzija definisana opštom konstantom generic:

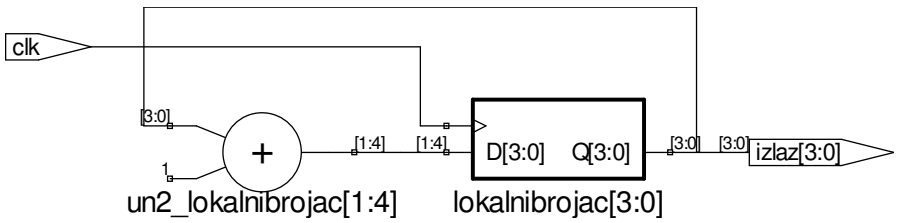
```
LokalAcc <= (others =>'0');
```

Naravno, ključna reč **others** može da se koristi i u ostalim slučajevima gde za to postoji potreba (popunjavanje vektora 1, Z i slično).

3.4.3. Brojači

Najjednostavnije i najčešće korišćene aritmetičke funkcije u složenim projektima jesu brojači. Savremeni alati za sintezu najčešće prepoznaju brojače na osnovu načina na koji su opisani i unose ih u projekat kao posebne celine. Zato je važno da se zna kako opisati brojač na način koji će mašina uspeti da razume.

Najjednostavniji tip brojača jeste asinhroni brojač unapred, koji uvećava stanje za jedan na svaku prednju ivicu takta. Odgovarajući opis dat je u primeru 3.27, a automatski sintetizovana hardverska pealizacija prikazana je na sl. 3.16.



Slika 3.16. Realizacija jednostavnog brojača unapred (primer 3.27)

Primer 3.27

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

entity Up_counter is
  port (
     clk: in STD_LOGIC;
     izlaz: out STD_LOGIC_VECTOR (3 downto 0)
        );
end Up_counter;

architecture Up_counter of Up_counter is
  signal LokalniBrojac:
                unsigned (3 downto 0):="0000";

 begin
    Dodaj_1: process (clk)
     begin
       if (clk'event and clk ='1') then
           LokalniBrojac <= LokalniBrojac + 1;
       end if;
     end process Dodaj_1;

   izlaz <= std_logic_vector(LokalniBrojac);
end Up_counter;
```

Važno je napomenuti da je korišćen lokalni signal `LokalniBrojac` iz dva razloga.

- Operacija `+` odnosi se na tip `std_logic`, a ne na `std_logic_vector`; `+` postoji samo za tipove **`unsigned`** i **`signed`** koji su, takodje, vektori čiji su elementi tipa `std_logic`;
- izlaz je moda `out`, tako da ne može da se nadje sa desne strane iskaza (ne može da se definiše `izlaz = izlaz + 1`).

U ovom primeru srećemo se i sa funkciom `std_logic_vector()` kojom se signal tipa `unsigned` pretvara u signal tipa `std_logic_vector`. Osim toga,

treba imati u vidu da brojač MORA da startuje iz poznatog stanja. Zato se lokalnom signalu u startu dodeljuje neka vrednost (ovde smo dodali "0000"). Ovo je važno sa stanovišta simulacije a ne sinteze.

Treba uočiti da **`if...then`** struktura nije kompletirana. Podsećamo da to u realizaciji izaziva strukturu implicitnog leča, odnosno povratnu vezu sa izalaza na ulaz, kao što pokazuje sl. 3.16.

Inkrementiranje vrednosti lokalnog signala `LokalniBrojac` u primeru 3.27 obavljeno je dodavanjem celobrojne vrednosti (intedžera) `'1'`, `unsigned` signalu:

```
LokalniBrojac <= LokalniBrojac + 1;
```

Mnogo je prirodnije da se sabiranje obavi nad sabircima istog tipa, što bi značilo da treba dodati:

```
LokalniBrojac <= LokalniBrojac + "0001";
```

Prethodna osobina naziva se preopterećenje operatora '+', jer jedan simbol ima isto značenje za različite tipove signala/promenljivih. Ona je ugradjena u VHDL čime se pojednostavljuje pisanje koda. Tako bi u prethodnom primeru, ako bi brojač trebao da uveća vrednost za 5, bilo mnogo jednostavnije i preglednije da se napiše

```
LokalniBrojac <= LokalniBrojac + 5;
```

nego

```
LokalniBrojac <= LokalniBrojac + "0101";
```

U poslednjem primeru ilustrujemo i činjenicu da vektor `unsigned` tipa podrazumeva da je MSB krajnje levo, a LSB krajnje desno. Preopterećenost operatora dopušta da operandi budu različitih dužina, pri čemu se kraći operand, implicitno, izjednačava sa dužim, ubacivanjem potrebnog broja nula sa leve strane. Tako bi isti efekat imao i opis

```
LokalniBrojac <= LokalniBrojac + "101";
```

Do sada smo se sretali i sa drugim preopterećenim operatorima, ali kod kojih isti simbol ima drugačije značenje, zavisno od mesta u kôdu na kome se nalazi (<=, = u dodeljivanju vrednosti i ispitivanju uslova).

Ilustracije radi, razmotrimo slučaj brojača unazad, kod koga se posle svake ivice takta, stanje brojača smanjuje za 3. VHDL opis ovog brojača dat je u primeru 3.28. Usvojićemo da brojač ima asinhroni reset signal kojim se izlaz postavlja u stanje koje je definisano četvorobitnim signalom sa nazivom `start` čija je vrednost, u našem primeru, stanje "1111"). Ovaj primer ilustruje primenu asinhronog reseta, konverziju std_logic_vector u unsigned vektor funkcijom unsigned() i definisanje dekrementa kao unsigned vektora dužine kraće od LokalniBrojac'range.

Primer 3.28

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;

entity Down_Counter3 is
  port(
    clk : in STD_LOGIC;
    start : in STD_LOGIC_VECTOR(3 downto 0);
    rst : in STD_LOGIC;
    izlaz : out STD_LOGIC_VECTOR(3 downto 0)
       );
end Down_Counter3;

architecture Down_Counter3 of Down_Counter3 is
signal LokalniBrojac:
   unsigned (3 downto 0) :=unsigned(start);
begin
   -- enter your statements here --

  Oduzmi_3: process (clk, rst, start)
  begin
   if rst ='1'
       then LokalniBrojac <= unsigned(start);
   elsif (clk'event and clk ='1')
       then
        LokalniBrojac <= LokalniBrojac - "11";
   end if;
```

```
    end process Oduzmi_3;

    izlaz <= std_logic_vector(LokalniBrojac);

end Down_Counter3;
```

---

*
* *

Kratak prikaz mogućnosti koje primena VHDL-a pruža u projektovanju digitalnih kola i sistema opisanih u ovoj glavi, samo treba da zaintrigira čitaoca da se ovoj temi ozbiljnije posveti ukoliko želi da se profesionalno bavi ovom oblašću. Osim VHDL-a, kao jezik za opis hardvera veoma se često korisi i Verilog. Čitaoce podsećamo na detalje koji se odnose na ovu temu, opisane u glavi 1.

Literatura

[Ash96] Ashenden P.J., The Designer`s Guide to VHDL, Morgan Kaufmann Publishers, Elsevier, 2008., ISBN: 978-0-12-088785-9

[Pet09] Petković Predrag, Milić Miljana, Mirković Dejan, VHDL i VHDL-AMS podrška projektovanju elektronskih kola i sistema, Univerzitet u Nišu, Elektronski fakultet 2009, ISBN 978-86-85195-85-3.

[Zwo00] Zwolinski M., Digital System Design with VHDL, Prentice Hall, 2000. ISBN: 0-201-36063-2.