Osnovni kurs VHDL

kôd prilagodjen sintezi

Osnovni kurs VHDL-a sastavni je deo predavanja iz predmeta «Projektovanje elektronskih kola» na smeru Elektronika i «Projektovanje integrisanih kola» na smeru Mikroelektronika koji se slušaju u VII i VIII semestru na Elektronskom fakultetu u Nišu.

Tekst pred vama predstavlja radnu verziju predavanja održanih u školskoj 2003/04 godini, a nastao je modifikacijom tekstova iz prethodnih godina. Prvi kurs u ovom obliku održan je školske 2001/02 godine.

Način izlaganja inspirisan je knjigom Sundar Rajan 'Essential VHDL: RTL Synthesis Done Right' , ISBN 0-9669590-0-0 iz 1998. godine. Naravno, u prikupljanju gradje korišćeni su i drugi bibliografski izvori, dati u poglavlju Literatura.

Autor podrazumeva da je polaznicima ovog kursa dostupan pomoćni udžbenik grupe autora 'Praktikum laboratorijskih vežbanja iz projektovanja i testiranja elektronskih kola i sistema', u izdanju Elektronskog fakulteta u Nišu, koji je uredila Prof. dr Milunka Damnjanović.

Sve primere iz ovog rukopis autor je proverio na ALDEC Active-HDL 5.1, Evaluation version, koja može da se preuzme sa internet adrese www.aldec.com/Registration/dwnl.htm.

Praktični deo kursa podrazumeva samostalni opis bar jednog projekta.

Autor moli čitaoce koji uoče greške u tekstu ili u primerima da ga o tome obaveste, kako bi ih otklonio u narednim verzijama.

Prof. dr Predrag Petković

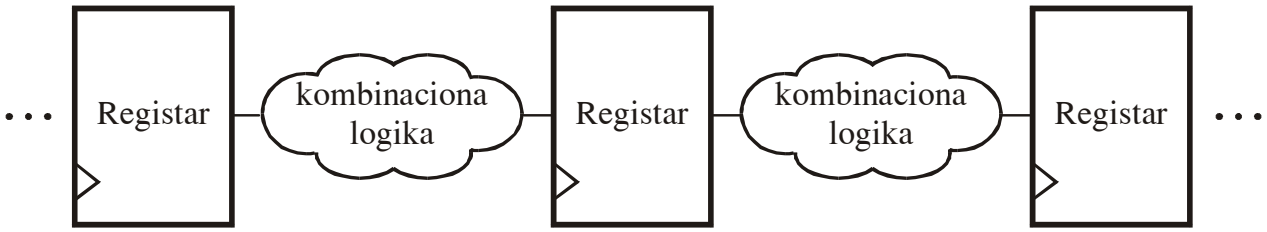
Uvod

Pokušaćemo ovim 'ubrzanim' kursom iz VHDL-a da za što kraće vreme dodjemo do prvih uspešnih opisa projekta. Ovo radimo sa namerom da demistifikujemo primenu VHDL-a kao složenog jezika. Pri tome ne treba izgubiti iz vida da VHDL poseduje mnoge opcije i da se jedna ista funkcija može opisati na više načina. Postoji više načina da se opiše ista funkcija, a da automatska sinteza rezultira istovetnim hardverom. Pri tome, neki od opisa su kraći a neki duži. Pored toga, zavisno od primenjenih konstrukcija i naredbi, istu funkciju mogu da realizuju različiti hardveri, pri čemu su jedni jednostavniji od drugih. Naravno, postoje i potpuno neželjene kategorije opisa, a to su one koje ne daju željenu funkciju kola. Da bi se one eliminisale, neophodno je simulacijom verifikovati opis pre njegovog prevodjenja u hardver.
S obzirom da se polaznici ovog kursa prvi put sreću sa VHDL-om, trudićemo se da što jednostavnije dodjemo do ispravnog opisa kola. Pri tome treba imati u vidu da mnoge jezičke konstrukcije koje pruža VHDL-93 standard neće biti pomenute u ovom kursu, već će se one obradjivati tokom nastave iz kurseva Projektovanje mikroračunara i Projektovanje VLSI.
Akronim VHDL označava Very High Speed Integrated Circuits Hardware Description Language. Nastao je iz potrebe da se popravi komunikacija medju projektantima koji rade na razvoju istog integrisanog kola. Standardizaciju je iniciralo i finansiralo Ministarstvo odbrane SAD-a (US DoD). VHDL je 1987. godine usvojen kao standard: IEEE Std. 1076-1987. Zato se ovaj HDL kraće zove VHDL-87. Priključivanje godine uz naziv ima smisla kada se uzme u obzir da razvoj tehnologije nameće nove zahteve jeziku, tako da IEEE komitet za standardizaciju svakih pet godina razmatra primedbe koje stižu od korisnika VHDL-a kako bi se povećala efikasnost jezika. Sledeća varijanta jezika definisana je standardom IEEE Std. 1076-1993 (VHDL-93), a razvijen je i VHDL AMS (što dolazi od Analogue and Mixed Signals) koja treba da omogući standardizaciju opisa analognih i hibridnih (analogno-digitalnih) kola. Ovaj standard usvojen je 1999. godine, te se naziva, mada redje, i VHDL-99.
VHDL može da se primenjuje u svim fazama projektovanja: opis, verifikaciju (simulaciju), sintezu i dokumentovanje. Osnovna prednost jeste mogućnost jednostavnog opisa projekta na višim nivoima apstrakcije.
Slično drugim programskim jezicima i u VHDL-u balansiranje izmedju fleksibilnosti i mogućnosti zahteva odredjenu disciplinu od strane korisnika time što uvodi striktna pravila. Ona se odnose na način deklarisanja tipova signala i na pravila koja definišu način opisa i mesto pojavljivanja odredjenih naredbi. Iako ova pravila na prvi pogled mogu da izgledaju zamršeno, njihova prava prednost dolazi do izražaja kod opisa složenih kola.

1. Osnove semantike i sintakse

1.1. Definicija logičkog kola na RTL nivou

Cilj ovog kursa jeste da osposobi inženjere da koriste VHDL za projektovanje, odnosno sintezu kola na RTL nivou (Register Transfer Level). Time se podrazumevaju kola koja se sastoje iz registara i kombinacione logike kao što pokazuje slika 1.1.

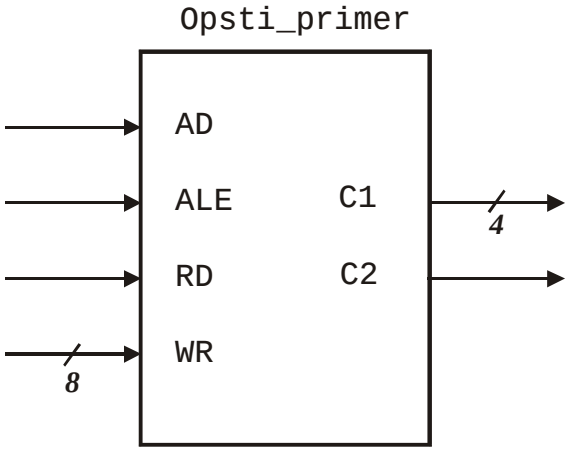


Slika 1.1: Definicija logičkog kola na RTL nivou

Sinteza na RTL nivou optimizuje logiku koja se nalazi izmedju registara.

1.2. Opis hardvera

Najbolji način da se grafički predstavi nepoznati hardver jeste u vidu bloka ili "crne kutije", odnosno pravougaonika sa izvodima, portovima **(*port*)**, preko kojih je vezan za svoje okruženje, kao što pokazuje slika 1.2.



Slika 1.2: Nepoznati hardver prikazan kao blok

Svaki ovakav blok VHDL prepoznaje kao **entitet** koji se definiše preko ključne reči ***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 reči ***architecture***.

Kao što se u elektronici jedana 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 zbog koje je projektovana. Na osnovu ovih prvih napomena vidi se da je struktura VHDL jezika prilagodjena osnovnim strukturama koje se javljaju tokom projektovanja elektronskih kola.

Osnovne konstrukcije u definisanju entiteta i arhitekture mogu da se iskažu za blok sa slike 1 na sledeći način.

Primer 1.1:

```
entity Opsti_primer is port(
 AD,
 ALE, RD ... \
 WR ...                  } deklaracija portova i konstanti
 C1 ...                  /
 C2 ...
 );
end entity Opsti_primer;

architecture proba of Opsti_primer is
 .                       \
 .                       } deklaracija parametara
 .                       /
begin
 .                       \
 .                       } telo arhitekture
 .                       /
end proba;
```

Važno je napomenuti da postoji još jedan, nešto sleoženiji oblik definisanja entiteta, kada se sem portova definišu i opšte konstante entiteta koji se zovu generici, ***generic***. O ovom tipu konstanti biće reči u poglavlju 3.2.

1.3. Šta su signali ?

Jedan entitet povezuje se sa drugim preko portova. Kroz portove putuju signali, a kako 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.

– 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, ... Da bi se usaglasio broj stanja sa kojima se manipuliše tokom verifikacije različitih projekata ili njihovih delova, najčešće se koristi skup logičkih stanja definisanih IEEE 1164 standardom. Ovaj skup vrednosti deklariše se kao ***std_logic*** tip signala, a prikazan je u Tabeli 1.1.

Tabela 1.1.

vrednost	značenje
U	neinicirani signal
X	jako nepoznato stanje
0	jaka 0
1	jaka 1
Z	visoka impedansa
W	slabo nepoznato stanje
L	slaba nula (Low)
H	slaba jedinica (High)
-	nebitno stanje (don't care)

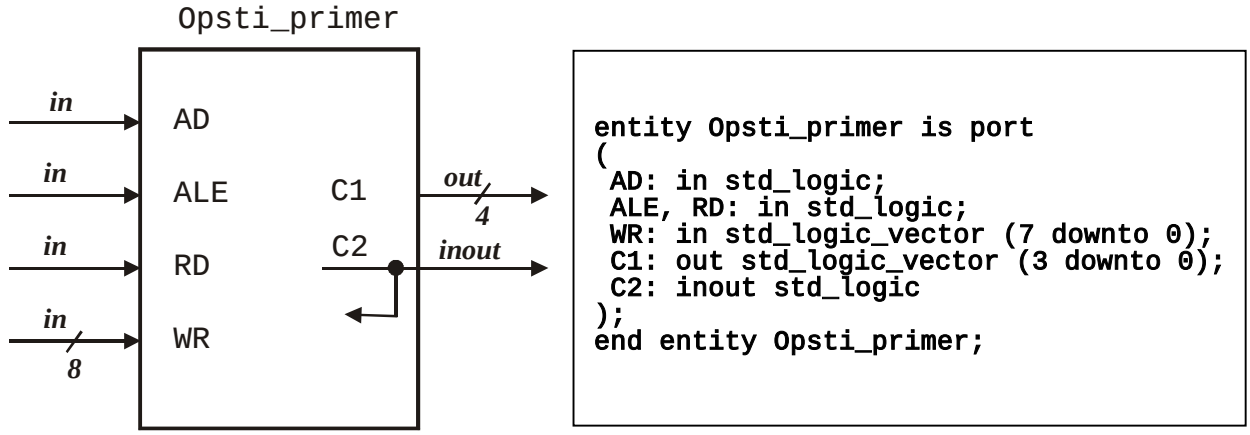
Prošireni skup vrednosti neophodan je zbog pravilnog modelovanja stanja u digitalnom kolu prilikom simulacije. Vrednosti "X" i "–" mogu se tretirati kao "don't care", a o razlikama medju njima biće reči kasnije. Da bi se izbegle razlike u tumačenju VHDL kôda od strane različitih programa za automatsku sintezu, preporučuje se dodeljivanje signalima samo X, 0, 1 i Z logičkih vrednosti.

– Druga važna karakteristika signala odnosi se na razliku izmedju signala koji putuje kroz jednu *žicu*, to je signal tipa **bit**, od signala koji se vodi kroz *magistralu*, odnosno **bus**. Signal

tipa bus predstavlja niz, odnosno vektor koji se sastoji od signala tipa bit (na slici 1.1 to su signali `WR` i `C1`). Zato se on deklariše kao signal tipa ***std_logic_vector***. Njemu je neophodno definisati dužinu izraženu u broju bitova, a mora se naznačiti i položaj LSB, odnosno MSB. U realnim sistemima, recimo PC računaru kod koga su magistrale u obliku sivog "kaiša", LSB /MSB je označen crvenom bojom.

– 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 signala navodi **mod** (***mode***) kojim se signali razvrstavaju na
  - ulazne, što se označava modom ***in***
  - izlazne, što se označava modom ***out***
  - bidirekcione, što se označava modom ***inout***.

Mod signala odredjuje se na osnovu toga gde se nalazi generator signala u odnosu na entitet.
Ukoliko je pobuda van entiteta, podaci ulaze u blok, i radi se o ***in*** signalu (portovi označeni sa `AD, ALE, RD` i `WR`) . Entitet može samo da čita sadržaj ovog signala, a ne može mu dodeljivati novu vrednost.
Ukoliko je pobuda u samom entitetu, radi se o ***out*** signalu (signal `C1`), vrednost mu se dodeljuje unutar entiteta, a njegov sadržaj ne može da se koristi za pobudu drugih delova istog entiteta. U slučaju da postoji potreba za takvim signalima, njima se definiše bidirekcioni mod ***inout*** (signal `C2`). Modovi svih signala eksplicitno su prikazani na slici 1.3.a. dok je na slici 1.3.b. prikazan potpuni VHDL opis entiteta `Opsti_primer`. Redosled navodjenja podataka je:
`naziv porta : mod tip`



a) **b)**

Slika 1.3: Značenje modova pojedinih portova

.

1.4. 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
– format navodjenja pojmova.

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

- **Ne pravi se razlika izmedju malih i velikih slova**

Primer 1.2:

Sve navedene oznake imaju isto značenje:
`Opsti_primer`

```
Opsti_Primer
OPSTI_PRIMER
```

- **Postoje rezervisani karakteri koji se NE SMEJU koristiti za označavanje pojmova: +, -, !, &, ... (umesto liste takvih karaktera, ovde dajemo skup opštih pravila:**
    a. Svi nazivi (identifikatori) moraju da počnu slovnim znakom
    b. Koristite samo slova (a-z i A-Z), brojeve (0-9) i podvučenu crtu (_)
    c. NE koristite znakove interpunkcije (!, ?, ., ,, i.t.d.)
    d. NE koristite sukcesivno dva simbola (_)
    e. NE mogu dva različita pojma u okviru istog entiteta ili arhitekture imati iste identifikatore.

Primer 1.3:

Ispravne oznake	Neispravne oznake	razlog
`Prvi_primer`	`1_Primer`	prekršeno pravilo a
`Opsti_Primer0`	`Opsti_#1_primer`	prekršeno pravilo b
`OPSTI_PRIMER_9`	`Opsti!_primer`	prekršeno pravilo c
`Opsti_Primer_0`	`Opsti__primer`	prekršeno pravilo d

- **Postoje rezervisane reči koje se NE SMEJU koristiti za označavanje pojmova. Ove reči date su u Tabeli 1.2. Samo neki programi za sintezu mogu da detektuju primenu ključnih reči u oznakama, ostali rezultiraju čudnim porukama o tipu greške ili pogrešnim rezultatom sinteze bez upozorenja!**

Tabela 1.2

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

* Označava rezervisane reči koje su uvedene VHDL-93 standardom

---

- **Linijski komentari počinju dvostrukim minus znakom "--".** 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 primenom komentara** kako bi ostali učesnici u projektu lakše tumačili značenje izvornog kôda. Primena linijskih komentara vidi se na primeru 1.4 kojim je opisan entitet sa slike 1.2.b:

Primer 1.4:

```
entity Opsti_primer is port

 ( AD: in std_logic;      -- moze svaki port da se deklarise u
                          -- posebnoj liniji
   ALE, RD: in std_logic; -- portovi istog tipa mogu da se
                          -- deklarisu u istoj liniji
   WR: in std_logic_vector (7 downto 0); -- tip bit-vektor mora
                                         -- da sadrzi duzinu i
                                         -- redosled, 7. je MSB

   C1: out std_logic_vector (0 to 3);    -- 0. je MSB, 3. - LSB
   C2: inout std_logic_vector --ovde nema ';'
 );  --jer se ';' stavlja iza zaokruzene
     --logicke celine koja sadrzi i zatvorenu zagradu
end entity Opsti_primer;
```

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

Primer 1.5:

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...elseif...else* strukturi:

Primer 1.6:

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

Jasno je da "**`if`** uslov1" ne predstavlja logičku celinu (pa ni naredbu) jer se očekuje neka aktivnost posle definisanja uslova. Takodje se, neposredno iza reči **`then`** očekuje 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 **`elseif`** naredbu u okviru *if...then...elseif...else* strukture.

- **Svaka logička celina završava se znakom** "**`;`**". Treba primetiti da ovo važi i za informaciju o završetku opisa cele strukture, **`end if`**.
- **Grupa naredbi obično počinje ključnom reči ili identifikatorom**, a može da sadrži
  -skup naredbi kojom se deklarišu promenljive koje se javljaju u toj grupi naredbi,
  -ključnu reč za početak opisa tela grupe naredbi **`begin`**,
  -telo grupe naredbi i
  -ključnu reč **`end`** kojom se opis grupe naredbi završava.

  Grupom naredbi definišu se entiteti, arhitekture, procesi i neke druge celine. Pojedinačno o svakoj od njih biće više reči kasnije. Za sada napominjemo da se iza **`begin`** očekuje nastavak opisa, tako da se iza nje ne očekuje znak **`;`** koji označava završetak logičke celine.

- **U okviru jedne logičke celine VHDL koristi "slobodni format" za pisanje koda.**

Primer 1.7:

```
if uslov1 then akcija1;

if     uslov1    then     akcija1;

if uslov1
    then akcija1;

if
uslov1
then akcija1;

if
  uslov1
then
  akcija1;
```

Svi prethodno navedeni opisi su korektni.

1.5. Opis osnovnih logičkih operacija

S obzirom da je VHDL namenjen opisu hardvera digitalnih kola, potpuno je prirodno da podržava korišćenje logičkih operatora nad signalima tipa *std_logic*. Spisak operatora koje VHDL podržava dat je u tabeli 1.3.

Tabela 1.3

operator	značenje

NOT	Logička operacija komplementa
AND	Logička I operacija
NAND	Logička NI operacija
OR	Logička ILI operacija
NOR	Logička NILI operacija
XOR	Logička Ekskluzivno ILI operacija
XNOR	Logička Ekskluzivno NILI operacija

Dodeljivanje logičke vrednosti nekom signalu označava se <= simbolom. Operacija nad dva signala, `a` i `b` može da se iskaže na sledeći način:

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

Operator NOT ima najveći prioritet, dok su svi ostali operatori iz Tabele 1.3 istog prioriteta. Redosled njihovog izvršavanja odredjuje se redosledom navodjenja 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). Medjutim u opisu:

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

redosled izvršavanja operacija jeste (((a AND (NOT b)) OR c) XOR d).
Redosled izvršavanja operatora kontroliše se upotrebom zagrada:

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

Najpre se obave AND i XOR operacija, a zatim OR. Da bi se povećala čitljivost koda, preporučuje se korišćenje zagrada.

1.6. Redosled navodjenja i izvršavanja naredbi

Ovo je veoma bitno.
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. Naime, arhitektura može da sadrži više logičkih celina a da signali iz jedne ne utiču na stanja u ostalim i obrnuto. Dakle, prirodno je da dogadjaji iz jedne celine ne utiču na dogadjaje u ostalima, odnosno da se prostiru nezavisno, konkurentno, t.j. paralelno u vremenu. VHDL podržava dva osnovna metoda kroz koje se signalima unutar arhitekture dodeljuju vrednosti. To su

- procesi (***process***) i
- konkurentna dodela vrednosti signalima.

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 obavlja se paralelno. Zato i redosled njihovog navodjenja u okviru opisa arhitekture nije važan.

Primer 1.8:

```
architecture proba of
 Opsti_primer is begin

  c2 <= ad OR ale;

  c1(0) <= rd AND wr(0);

  seq: process (wr(7))
   begin    .
            .
            .
  end process seq;

end architecture proba;
```

=

```
architecture proba of
 Opsti_primer is begin

  c2 <= ad OR ale;

  seq: process (wr(7))
   begin    .
            .
            .
  end process seq;

  c1(0) <= rd AND wr(0);

end architecture proba;
```

Z

Za razliku od konkurentnog dodeljivanja vrednosti signalima unutar arhitekture, unutar procesa dodeljivanje vrednosti obavlja se sekvencionalno po redosledu po kome su naredbe navedene. Zato je redosled navodjenja naredbi unutar procesa veoma bitan.

Primer 1.9:

```
architecture proba of
Opsti_primer is begin

  seq: process
   (ad, ale, rd, wr)
    begin

     c2 <= ad OR ale;

     c2 <= rd AND wr(0);
  end process;

end proba;
```

≠

```
architecture proba of
Opsti_primer is begin

  seq: process
   (ad, ale, rd, wr)
    begin

     c2 <= rd AND wr(0);

     c2 <= ad OR ale;
  end process;

end proba;
```

Generalno gledano, naredbe u procesu, pored toga što se izvršavaju sekvencijalno, karakterišu još dve važne osobine:

- proces traje u beskonačnoj petlji ukoliko ne dobije nalog da se zaustavi;
- proces se zaustavlja naredbom čekanja (***whait***) dok se vrednost nekog signala ne promeni.

Da bismo pojednostavili primenu procesa, posmatraćemo samo njegov jednostavni oblik koji je i prikazan na slici 1.4. Ovaj oblik podrazumeva da jedan proces može da počne samo kada nastane promena u nekom od signala koji se navode kao lista u zaglavlju opisa procesa. To su signali na koje je proces *osetljiv*. (Ovo odgovara slučaju da se kao poslednje naredbe u procesu nalaze naredbe koje nalažu čekanje dok ne dodje do promene stanja onih signala na koje je proces osetljiv). Detaljniji opis drugih oblika definisanja procesa može se naći u [1, 2, 3].

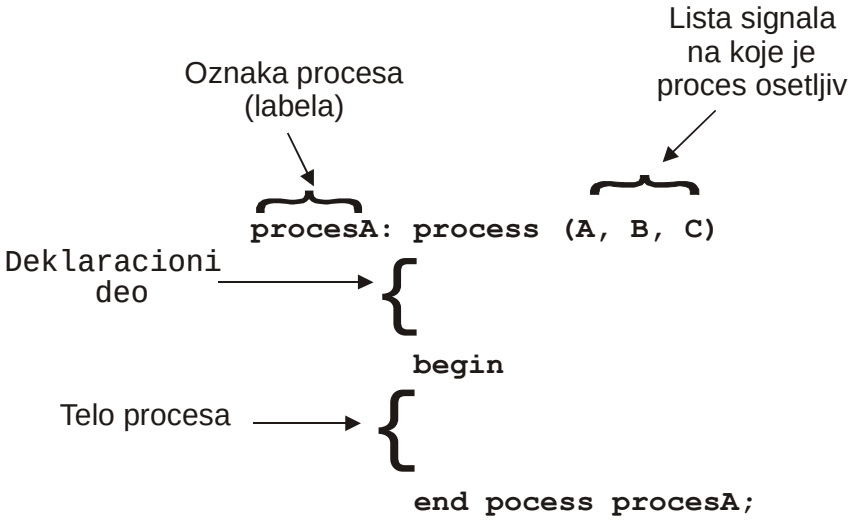
Tipična struktura opisa procesa prikazana je na slici 1.4.

Svaki proces trebao bi da nosi jedinstvenu oznaku. Ona je opciona ali se preporučuje naročito sa stanovišta sinteze. Signali koji se generišu automatskom sintezom sadržaće i ime procesa iz kojih su proistekli čime se olakšava tumačenje celog projekta, naročito u fazi otkrivanja potencijalnih grešaka.

Oznaka procesa završava se sa ‘`:`’ iza čega sledi ključna reč **`process`** i izmedju malih zagrada lista signala na koje je proces osetljiv, razdvojeni zapetama. Proces nazvan **`procesA`** sa slike 1.4 aktiviraće se kad god nastane promena u ma kome od signala **`A`**, **`B`** ili **`C`**. Navodjenje kompletne liste signala na koje je proces osetljiv od IZUZETNOG je značaja za pravilan opis projekta.

Proces, kao i arhitektura, sadrži polje za deklarisanje elemenata lokalnih za taj proces (ALI lokalni signali NE SMEJU da se deklarišu u okviru **`process`**-a).

- Centralno polje zauzima telo procesa u kome se navode naredbe u striktno sekvencijalnom redosledu.



Slika 1.4: Osnovna struktura opisa procesa

Opis procesa završava se sa **`end process`**, a opciono može da se doda ime procesa što se preporučuje. Poredjenjem poslednjeg primera sa prethodnim vidi se da nije neophodno uz naredbu **`end process`** naznačiti i oznaku procesa (u ovom slučaju **`seq`**).

Treba napomenuti da, iako se unutar procesa naredbe izračunavaju sekvencijalno, hardver dobijen posle sinteze, realizuje se na bazi konkurentne logike. Medjutim unutar nje su ugradjeni elementi koji obezbedjuju da se na izlazu pojavljuju podaci u željenom redosledu.

1.7. Stilovi opisa projekta

VHDL podržava opis projekata na algoritamskom nivou i nivou 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 projekta.

Postoje tri stila opisa projekta u VHDL-u. To su

- strukturni opis (struktural)
- opis toka podataka (dataflow)
- opis ponašanja (behavioral)

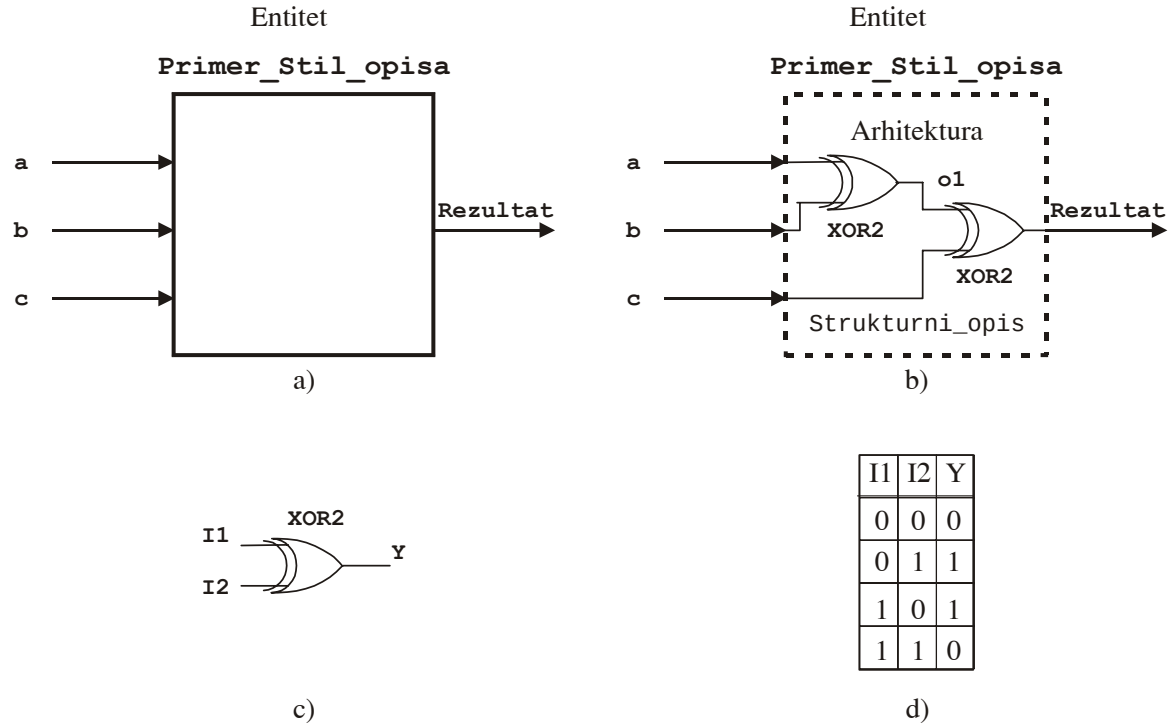
Da bismo ilustrovali razliku izmedju pojedinih stilova opisa razmotrićemo primer opisa kola sa slike 1.5.a.

Da bi se opisalo ovo kolo, treba najpre, definisati blok, odnosno entitet koji smo nazvali **`Primer_Stil_opisa`**.

Primer 1.10:

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

Sada se posvećujemo opisu arhitekture ovog entiteta.



Slika 1.5: a) Entitet Primer_stil _opisa, b) struktura njegove arhitekture
c) dvoulazno XOR kolo, XOR2, d) tablica istinitosti XOR2 kola

Najpre pretpostavimo da nam je struktura arhitekture na nivou logičkih blokova poznata i da izgleda upravo kao što prikazuje slika 1.5.b. Ovakav grafički opis bi se dobio primenom nekog od grafičkih editora električne šeme. Medjutim, on podrazumeva da nam na raspolaganju stoje logički elementi koji obavljaju funkciju XOR. Neka je njihov naziv XOR2, ulazni portovi nose nazive **`I1`** i **`I2`**, dok je izlazni port **`Y`**, kao što pokazuje slika 1.5.c. Kasnije tokom kursa videćemo kako se definišu i gde se smešta opis ovih elemenata. Za sada ćemo reći samo toliko da je njihova logička funkcija definisana tablicom istinitosti datom na slici 1.5.d.

Važno je napomenuti da sem portova, postoji unutar arhitekture jedan novi signal, označen na slici 1.5.b 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 1.1.2), a zatim sledi opis arhitekture koji mnogo podseća na net listu koja se sreće u mnogim programima za logičku simulaciju (PSpice).

Primer 1.11:

```
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,    -- ćtelo arhitekture
                         c => I2,
                         Rezultat => Y);
.
end architecture Strukturni_opis;     -- dovoljno je bilo reci
                                      -- end Strukturni_opis;
```

Opis iste funkcije entiteta **`Primer_Stil_opisa`** na nivou toka poidataka (dataflow) dat je sledećim primerom.

Primer 1.12:

```
architecture Dataflow 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 Dataflow;
```

Očigledno je korišćen pristup konkurentne dodele vrednosti signalima, tako da redosled navodjenja pojedinih naredbi nije bitan, s obzirom da se one izvršavaju paralelno (konkurentno), kao što je rečeno u odeljku 1.6.
Opis ponašanja ne razlikuje se mnogo u suštini od opisa toka podataka, naročito kada su u pitanju manja kola. Opis ponašanja zasniva se na algoritamskom opisu bloka pri čemu se koristi definisnisanje procesa (kao u odeljku 1.6) da bi se opisale sekvencijalne aktivnosti.

Primer 1.13:

```
     architecture Opis_Ponasanja of Primer_Stil_opisa is

     begin

      XOR_od_3: process (a, b, c)   -- imenovanje (labeliranje) procesa
                                    -- 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;
```

2. Naš prvi projekat

Definisaćemo naš prvi zadatak funkcionalnim opisom preko tablice istinitosti date u Tabeli 2.1. Dakle, zadatak je projektovati kolo koje, ako se na ulaze dovedu signali A, B i C, na izlazu daje signal Y specificiran tablicom istinitosti iz Tabele 2.1.

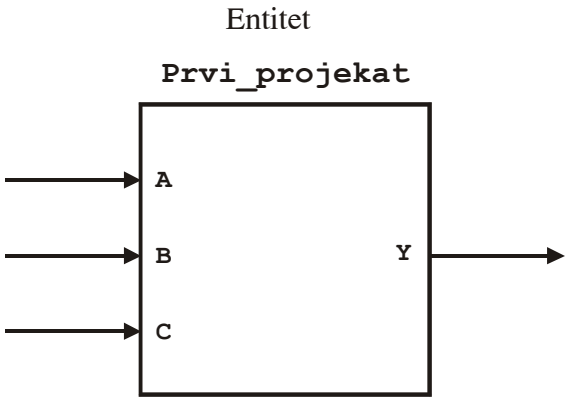
Tabela 2.1.

A	B	C	Y
0	0	X	1
X	X	1	1
ostale kombinacije			0

Oznake stanja u Tabeli 2.1 odgovaraju onima iz ***std_logic*** seta datog u Tabeli 1.1. Signali A, B i C su ulazni dok Y označava izlazni signal.

1.1. Definisanje entiteta

Na osnovu Tabele 2.1 dolazi se do zaključka da ceo logički blok može da se predstavi grafički kao i entitet koji smo koristili za ilustraciju entiteta **`Primer_Stil_opisa`** sa slike 1.5, a ponavljamo je na slici 2.1.



Slika 2.1: Prvi projekat prikazan u obliku bloka

Dakle, opis entiteta koji ćemo nazvati **`Prvi_projekat`** liči na onaj iz poglavlja 1.8.

Primer 2.1:

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

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

Novinu u opisu predstavljaju prva dva reda. Prvi se odnosi na deklarisanje biblioteke koja će biti korišćena tokom opisa. Naime, sve ono što nije definisano standardnim opisom nalazi se u biblioteci. Za sada podsećamo da smo definiciju tipa ***std_logic*** vezali za tipove signala date u Tabeli 1.1. Definisanje logičkih stanja u okviru opisa svakog projekta samo bi nepotrebno opterećivalo

kôd. Umesto toga, pozivamo se samo na biblioteku koja sadrži te podatke. U ovom slučaju radi se o biblioteci nazvanoj **IEEE**.
U okviru biblioteke koju prepoznaje VHDL nalazi se više celina koje se zovu paketi, dok su u paketima smeštene neophodne informacije o nestandardnom pojmu koji se traži (knjige). Njima se pristupa pozivanjem iskaza **`use`** koji znači iskoristi. Sintaksa iskaza **`use`** zahteva da se navede naziv biblioteke, zatim paketa, i na kraju navodi se željeni pojam. Svi oni razdvojeni su tačkom. Ukoliko umesto jednog pojma stoji reč **`all`** (sve), to znači da se koriste svi pojmovi iz specificiranog paketa što je slučaj u našem primeru.
Zapravo, kad god se koriste signali tipa ***std_logic*** čija se definicija (data Tabelom 1.1) nalazi u okviru standarda IEEE Std. 1164, koriste se biblioteka ***`IEEE`*** i paket ***`std_logic_1164`*** i to svi njeni elementi (***`all`***). O bibliotekama će više reči biti kasnije tokom ovog kursa.

2.1. Definisanje arhitekture

Kao što smo u prethodnoj glavi videli, arhitektura može da se opiše na bazi **protoka podataka**, strukturnom nivou i na nivou ponašanja. Za jednostavna kombinaciona kola najkompaktniji oblik definisanja arhitekture daje opis na bazi protoka podataka koji koristi konkurentne naredbe o kojima je bilo reči ranije u odeljku 1.6:
Primer 2.2:

```
architecture Protok_podataka of Prvi_projekat is

begin
 Y <= '1' when (A = '0' AND B = '0') OR
               (C = '1')
          else '0';
end architecture Protok_podataka;
```

Ovom prilikom ukazujemo na činjenicu da je redosled izvršavanja logičkih jednačina definisan zagradama.
U ovom primeru koristi se uslovna naredba u kojoj ključna reč ***when*** (treba je čitati "**kada je**") definiše uslove pod kojima Y uzima vrednost logičke jedinice, dok reč ***else*** ("u **ostalim slučajevima**") ukazuje na vrednost koju dobija Y u ostalim slučajevima. Dodeljivanje vrednosti signalima (levom strelicom <=) ukazuje na tok podataka, što odredjuje naziv ovom načinu opisa.

Opis ponašanja zasniva se na opisu pojedinih procesa (***process***). Kao što je rečeno u odeljku 1.6, u okviru grupe naredbi definisanih procesom, naredbe se izvršavaju sekvencijalno.

Postoji nekoliko sekvencijalnih naredbi u VHDL-u. Jedna od najčešće korišćenih je **`if`** naredba.
`if` naredba:

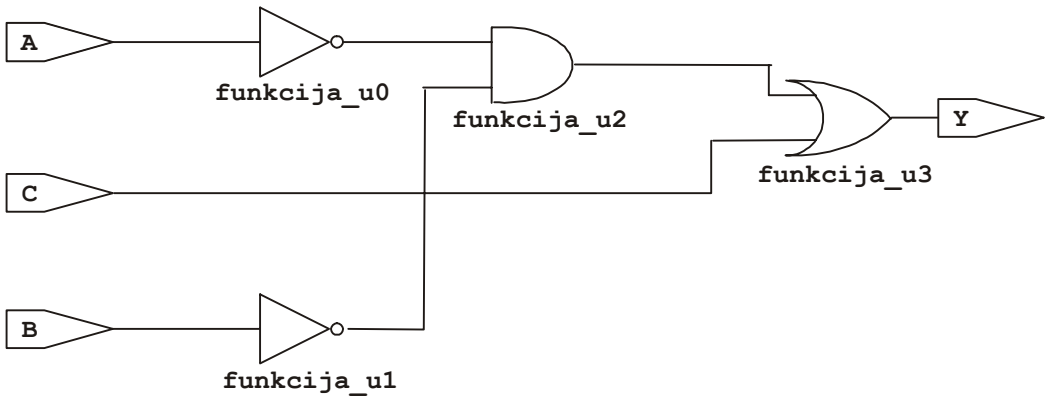
- koristi se da bi se ispitao odredjeni uslov pre izvršenja neke operacije;
- može da ispita višestruke uslove ako se koristi **`elsif`**
- završava se sa **`end if`**
- treba je kompletirati sa **`else`** kako bi se iscrple sve mogućnosti iz if lanca.

Imajući u vidu sve navedene osobine **`if`** naredbe zahtevana logička funkcija može da se iskaže na nivou opisa ponašanja na sledeći način:
Primer 2.3:

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

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

```
architecture Opis_ponasanja of Prvi_projekat is

begin

  funkcija: process (A, B, 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;

end architecture Opis_ponasanja;
```

Automatska sinteza koja proističe na osnovu oba načina opisa arhitekture našeg prvog projekta daje isti hardver prikazan na slici 2.2.

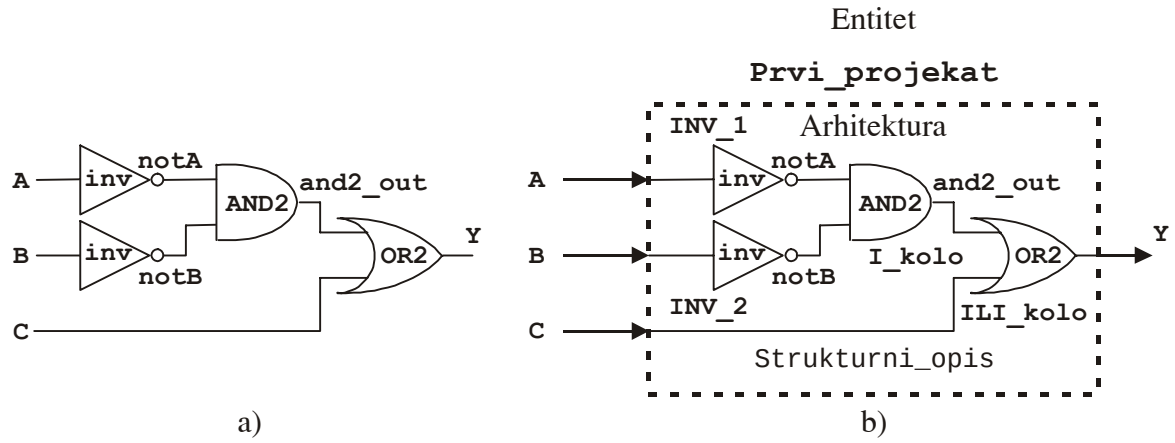


Slika 2.2: Rezultat sinteze prvog projekta

Projektovanje zasnovano na opisu ponašanja jednog entiteta, ne definiše način protoka podataka niti strukturu kola. Medjutim, programi za automatsku sintezu na osnovu opisa ponašanja generišu hardver. Zato je izuzetno važno da se stekne osećaj za to kakvu će implikaciju imati pojedina naredba i grupa naredbi u opisu projekta. Ovome ćemo posvetiti pažnju u nastavku kursa.

Strukturni opis odgovara pisanju net liste kola čija je logička šema poznata. Dakle, podrazumeva se da je već izvršena dekompozicija projekta na osnovne funkcionalne blokove, u našem slučaju to su logički elementi. Ovaj način opisa nije naročito efikasan, po broju linija, ali je koristan jer direktno preslikava nameru projektanta o izvršenoj dekompoziciji i vezi izmedju pojedinih celina.

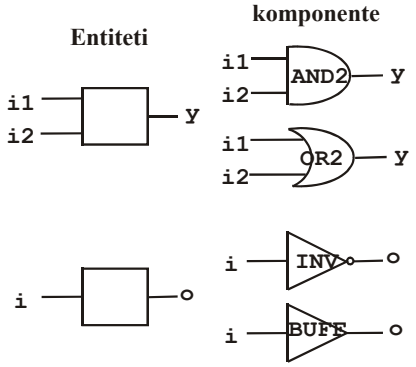
Očigledno je da pre nego se pristupi opisu treba skicirati logičku šemu odnosno izvršiti logičku sintezu željene funkcije. Za naš projekat odgovarajuća šema prikazana je na slici 2.3.



Slika 2.3: Struktura prvog projekta a) logička šema b) arhitektura

Kako slika 2.3 pokazuje, za realizaciju željene funkcije neophodna su nam dva invertora i po jedno dvoulazno I i ILI kolo. Sve ove komponente moraju prethodno da se definišu kao entiteti sa arhitekturom. Pri tome, nismo slučajno upoterbili reč komponenta. Naime, ***component*** je rezervisana reč koja označava entitet sa pridruženom arhitekturom.

Razlika izmedju entiteta i komponente očigledna je sa slike 2.4. 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 funkcije u kolu razlikuju jer su im arhitekture različite. Ista konstatacija važi i za invertor INV i bafer BUFF.



Slika 2.4 Razlika izmedju entiteta i komponente

Važno je napomenuti da:

- komponenta mora da ima isto ime kao i entitet
- imena signala u komponentama trebala bi da budu ista kao imena signala u entitetima.

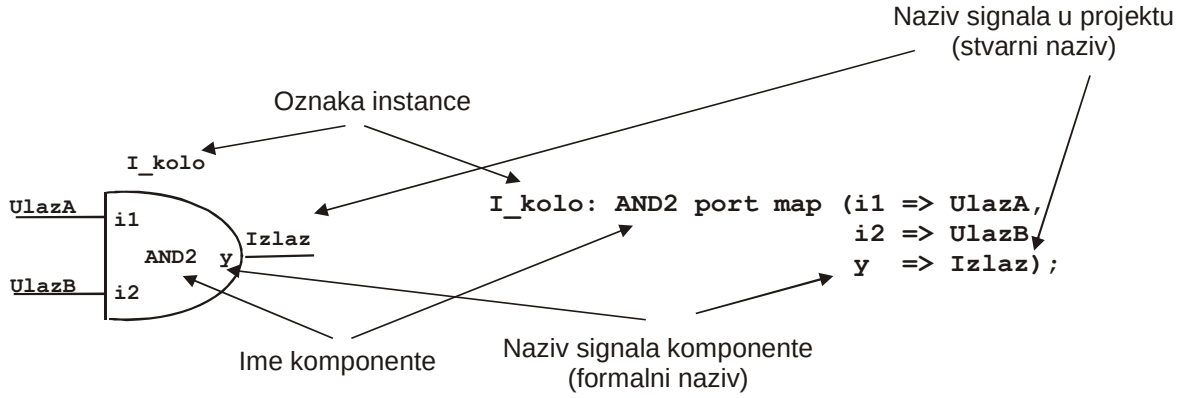
Uzorak tako definisane komponente može da se, zatim, koristi više puta u okviru projekta, time što se postavlja na odgovarajuće mesto u strukturnoj šemi. S obzirom da u našem jeziku ne postoji glagol koji opisuje takvu aktivnost, mi ćemo je, ubuduće, zvati *instanciranje* (od engleskog *instantiate*). Ovde se treba podsetiti da crtanje logičke ili električne šeme uz pomoć ma kog šematskog editora (recimo *Psche*d koji ide uz *Pspice*), svodi se na instanciranje komponenti iz biblioteke u šemu projekta.

Dakle, za opis glavnog projekta, potrebno je:

a. opisati par entitet-arhitektura za svaku buduću komponentu (pre definisanja entiteta glavnog projekta),
b. proglasiti (deklarisati) odgovarajuće parove entitet-arhitektura komponentama u okviru arhitekture projekta u kome će se koristiti (čime se omogućava njihova višestruka primena),

c. uneti ih (instancirati) u opis arhitekture projekta.

Sintaksa instanciranja ilustrovana je na primeru dvoulaznog I kola na slici 2.5 koje je definisano kao komponenta pod nazivom AND2, ima dva ulazna porta označena sa **i1** i **i2**, i jedan izlazni port, **y**. U arhitekturi u koju se smešta, ovoj komponenti je dodeljeno ime I_kolo, a portovi su vezani za stvarne signale sa imenima UlazA, UlazB i Izlaz.



Slika 2.5: Format instanciranja komponente

Pri svakom pozivu (instanciranju) jedne komponente istog tipa neophodno joj je dodeliti jedinstveni identifikator, iza koga sledi ‘:’, a zatim ime komponente. Zatim se navode ključne reči **port map** koje prati lista pridruživanja stvarnih imena signala u projektu, formalnim imenima signala na pristupima komponente.

Treba napomenuti da pored ovog ‘imenovanog’ dodeljivanja stvarnih signala formalnim portovima, postoji i takozvano ‘poziciono’ dodeljivanje. Ono zahteva da se redosled navodjenja stvarnih signala poklapa sa redosledom navodjenja portova u entitetu. U tom slučaju nije neophodno pisati imena formalnih portova i znak ‘=>’. Iako ovaj način deluje jednostavnije, preporučuje se primena imenovanog dodeljivanja jer je preglednije.

Pre početka opisa projekta ne treba zaboraviti, da u okviru strukture sa slike 2.3 postoje tri lokalna čvora koja nisu obuhvaćena portovima entiteta. Zato je neophodno da se oni navedu u polju za deklaraciju lokalnih signala zajedno sa definisanjem tipa signala u okviru arhitekture. Potpuni opis celog projekta na strukturnom nivou ilustruje sledeći primer.

Primer 2.4:

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

entity INV is
  port ( i: in std_logic;
         o: out std_logic);
 end entity INV;

architecture rtl of INV is
 begin
    o <= not i;
end architecture rtl;

-- AND2
library IEEE;
use IEEE.std_logic_1164.all;

entity AND2 is
  port ( i1: in std_logic;
```

```
        i2: in std_logic;
        y: out std_logic);
end entity AND2;

architecture rtl of AND2 is
 begin
    y <= '1' when i1 = '1' and i2 = '1' else '0';
end architecture rtl;

-- OR2
library IEEE;
use IEEE.std_logic_1164.all;

entity OR2 is
  port ( i1: in std_logic;
         i2: in std_logic;
         y: out std_logic);
end entity OR2;

architecture rtl of OR2 is
 begin
    y <= '1' when i1 = '1' or i2 = '1' else '0';
end architecture rtl;

-- Definisanje entiteta za Prvi_projekat
library IEEE;
use IEEE.std_logic_1164.all;

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

-- Definisanje strukturnog opisa arhitekture
architecture Strukturni_opis of Prvi_projekat is

-- deklaracija komponenti
      component INV
      port ( i: in std_logic;
             o: out std_logic);
      end component INV;

      component AND2
      port ( i1: in std_logic;
             i1: in std_logic;
             y: out std_logic);
      end component AND2;

      component OR2
      port ( i1: in std_logic;
             i1: in std_logic;
             y: out std_logic);
      end component OR2;

-- deklaracija internih signala
     signal notA, notB, and2_out: std_logic;

     begin

      INV_1: INV port map (i => A,  -- instanciranje prvog invertora
                           o => notA);

      INV_2: INV port map (i => B,  -- instanciranje drugog invertora
                           o => notB);

      I_kolo: AND2 port map (i1 => notA,  -- instanciranje I kola
                             i2 => notB,
```

```
                        y  => and2_out);

    ILI_kolo: OR2 port map (i1 => and2_out,  -- instanciranje ILI kola
                            i2 => C,
                            y  => Y);

  end architecture Strukturni_opis;
```

Napomene:
- Par entitet-arhitektura za svaku komponentu navodi se pre definisanja komponente u kojoj se pozivaju (Prvi_projekat).
- Svakom paru entitet-arhitektura prethode posebne **`lib ieee`** i **`use`** naredbe.
- Komponente se instanciraju konkurentno u okviru arhitekture a nikada se ne smeštaju unutar procesa.

Opis složenih projekata koji sadrže veći broj komponenti postaje nepregledan ako se navode potpuni opisi svih komponenata. S obzirom da je VHDL pisan sa namerom da olakša opis složenih projekata, uveden je pojam *biblioteka* sa ciljem da se deo opisa nekih kommponenta smesti u biblioteku i kasnije iz nje poziva, gde god zatreba u projektu. Pored toga, u okviru biblioteka definišu se *paketi*, a sve komponente koje su definisane u nekom paketu postaju dostupne za ponovnu upotrebu i u okviru drugih projekata.
Značaj biblioteka i paketa zahteva da im se posvet više pažnje i da se opšu u više detalja.

Biblioteke
Fizička implementacija biblioteka nije obuhvaćena standardom, tako da ih različiti alati drugačije realizuju, ne narušavajući njihovu osnovnu namenu. Najčešće se, ipak, one realizuju kao direktorijumi. Kao što smo do sada videli na primeru biblioteke ***IEEE***, one moraju da imaju jedinstveno ime. Biblioteke postaju dostupne naredbom.

```
library ieee;
```

Pored ovih postoje i dve biblioteke koje su implicitno dostupne korisnicima bez navodjenja. To su **`std`** i **`work`**. U biblioteci **`std`** smešteni su podaci o tipovima promenljivih i operatorima (*and, or, not,* i.t.d.) za promenljive tipa bit ili operatori relacija (=, <, >, i.t.d.). Biblioteka **`work`** je podrazumevana (default) radna korisnička biblioteka u koju se smaštaju podaci o projektu.

Dodavanje komponenti u biblioteku
Često je jako korisno da se logičke celine iz jednog projekta čuvaju u posebnim fajlovima. Time se pojednostavljuje praćenje složenih projekata. Većina alata za sintezu podržava podelu projekta u više fajlova. Prethodni primer bi mogao da se pojednostavi ukoliko se svaki od parova entitet-arhitektura za INV, AND2 i OR2 smesti u poseban fajl.
Izgled fajla u kome je definisan invertor bio bi:

Primer 2.5:

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

      entity INV is
      port ( i: in std_logic;
             o: out std_logic);
     end entity INV;

      architecture rtl of INV is
      begin
           o <= not i;
      end architecture rtl;
```

Kompilovanjem ovih fajlova, automatski se smeštaju podaci o komponentama u radnu biblioteku **work**. Oni postaju upotrebljivi za projekat ako se pozovu naredbom **use**. Za konkretan primer, iza opisa entiteta Prvi_projekat treba navesti

```
use work.inv;
use work.and2;
use work.or2;
```

Opis našeg projekta izmeniće se utoliko što su sve komponente definisane u posebnim fajlovima, dok se u glavnom fajlu za `Prvi_projekat` pojavljuju naredbe **use** pre definisanja arhitekture, kao što pokazuje Primer 2.6.

Primer 2.6:

```
-- Definisanje entiteta za Prvi_projekat
      library IEEE;
      use IEEE.std_logic_1164.all;

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

-- Naredbe use cine vidljivim komponente iz biblioteke work
use work.inv;
use work.and2;
use work.or2;

-- Definisanje strukturnog opisa arhitekture
     architecture Strukturni_opis of Prvi_projekat is

-- deklaracija komponenti
      component INV
      port ( i: in std_logic;
             o: out std_logic);
      end component INV;

      component AND2
      port ( i1: in std_logic;
             i1: in std_logic;
             y: out std_logic);
      end component AND2;

      component OR2
      port ( i1: in std_logic;
             i1: in std_logic;
             y: out std_logic);
      end component OR2;

-- deklaracija internih signala
     signal notA, notB, and2_out: std_logic;

     begin

      INV_1: INV port map (i => A,  -- instanciranje prvog invertora
                           o => notA);

      INV_2: INV port map (i => B,  -- instanciranje drugog invertora
                           o => notB);

      I_kolo: AND2 port map (i1 => notA,  -- instanciranje I kola
                             i2 => notB,
                             y  => and2_out);

      ILI_kolo: OR2 port map (i1 => and2_out,  -- instanciranje ILI kola
                              i2 => C,
                              y  => Y);
```

```
end architecture Strukturni_opis;
```

Ni ovaj način opisa ne deluje naročito efikasno ako se uzme u obzir da se u velikim projektima javlja daleko veći broj komponenta na niskom nivou, gejtovskom. Šta više, najverovatnije će mnogi projekti koristiti ove komponente. Da bi se pojednostavio opis projekta preporučuje se korišćenje *paketa* u okviru biblioteke.

Paketi

Paket predstavlja skup deklaracija kojima može da se pristupi korišćenjem iskaza **`use`**. Pretpostavimo da želimo deklaraciju svih komponenta najnižeg nivoa da smestimo u paket koji hoćemo da nazovemo **primitive** a nalazi se u **`work`** biblioteci.

Pretpostavljajući da se parovi entitet-arhitektura već nalaze u posebnim fajlovima za sve komponente, tada se u posebnom fajlu, u jednom paketu, deklarišu sve primitive (za sada ih imamo tri) na način prikazan u Primeru 2.7:

Primer 2.7:

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

package primitive is

component INV
port ( i: in std_logic;
 o: out std_logic);
 end component INV;

component AND2
port ( i1: in std_logic;
        i2: in std_logic;
 y: out std_logic);
end component AND2;

component OR2
port ( i1: in std_logic;
        i2: in std_logic;
 y: out std_logic);
end component OR2;

end package primitive;
```

Ovde napominjemo da su **`package`** i **`is`** rezervisane reči, a **primitive** je naziv koji smo mi definisali.

Da bi sadržaj paketa mogao da se koristi, on mora da se iskompajlira pre glavnog projekta. Najčešći oblik iskaza **`use`** pri sintezi je:

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

Tako da bismo mogli da koristimo naredbe

```
use work.primitive.inv;
use work.primitive.and2;
use work.primitive.or2;
```

Medjutim, da bi se pojednostavio poziv na komponente, može da se koristi i kraći oblik:

```
use work.primitive.all;
```

koji podrazumeva da su dostupne sve komponente iz paketa **primitive** u biblioteci **work**.

Konačno, posle smeštanja svih komponenta u paketu primitive, strukurni opis našeg projekta biće:

Primer 2.8:

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

entity Prvi_projekat is
```

```
  port( A: in std_logic;
        B: in std_logic;
        C: in std_logic;
        Y: out std_logic);
end entity Prvi_projekat;
                                                          -- koristi sve
     komponente iz
use work.primitive.all;            -- paketa "primitive"

-- Definisanje strukturnog opisa arhitekture
     architecture Strukturni_opis of Prvi_projekat is

                                   -- nema deklaracije  "component"
signal notA, notB, and2_out: std_logic;

begin

  INV_1: inv port map ( i => A,
                        o => notA);

  INV_2: inv port map ( i => B,
                        o => notB);

  I_kolo: AND2 port map (i1 => notA,   -- nije zavrsena logicka celina
                         i2 => notB,   -- mape portova instance i
                                       -- nema ';', vec se koristi ','
                         y  => and2_out);

  ILI_kolo: OR2 port map (i1 => and2_out,
                          i2 => C,
                          y  => Y);

end architecture Strukturni_opis;
```

Napomena:

Treba zapamtiti:

a) biblioteka primitiva postaje dostupna posle iskaza **use** na početku opisa;
b) sadržaj biblioteke koja je 'otvorena' sa **library** i **use** važi samo za projektnu celinu koja se nalazi neposredno iza ovih iskaza (kada se u istom fajlu nalazi opis više parova entitet-arhitektura, svaki mora da ima svoje referenciranje na biblioteku i pakete, *Primer 2.4*; isto bi važilo i za slučaj da se iza deklaracije paketa u *Primeru 2.7* nalazila nova deklaracija entiteta ili paketa) ;
c) za razliku od portova koji su sastavni deo opisa entiteta, lokalni signali se deklarišu u okviru arhitekture;
d) prilikom instanciranja svakoj komponenti dodeljuje se jedinstvena oznaka;

Konfigurisanje komponenata

Pored toga što postoji mogućnost da se jedna komponenta instancira više puta, VHDL dopušta projektantu da *konfiguriše* (***configuration***) svaku od njih na drugačiji način. Pod konfiguracijom se podrazumeva dodela konkretne arhitekture istom entitetu. Da bismo ovo pojasnili podsećamo na činjenicu da jedan entitet, pa i komponenta, može da ima više arhitektura. Pri tome, treba imati u vidu da konfiguracija ima smisla kada se radi o simulaciji, jer ovu opciju ne podržavaju svi alati za automatsku sintezu. Umesto toga, oni koriste jednu podrazumevanu (default) arhitekturu.
Različitim instanciranim komponentama istog tipa mogu, dakle, da se dodele različite arhitekture. Ovo se omogućava blokom ***configuration*** čija je sintaksa:

```
configuration <identifikator> of <ime entiteta> is
  for <ime arhitekture>
   telo konfiguracionog bloka
end configuration identifikator;
```

Za naš primer definisaćemo konfiguraciju pod imenom **NasePrimitive**:

Primer 2.9:

```
configuration NasePrimitive of Prvi_projekat is
  for structural

    for INV_1: INV
      use entity primitive.INV(rtl);
    end for;

    for INV_2: INV
      use entity primitive.INV(rtl);
    end for;

    for I_kolo: AND2
      use entity primitive.AND2(rtl);
    end for;

    for ILI_kolo: OR2
      use entity primitive.OR2(rtl);
    end for;

  end for;

end configuration NasePrimitive;
```

Ovde smo koristili arhitekture **rtl** koje su definisane na način prikazan u *Primeru 2.4*. Da je za definiciju invertora postojala i arhitektura sa nazivom **rtlA**, mogao je jedan od invertora da se konfiguriše sa tom arhitekturom, tako što bi se, recimo, navelo:

```
for INV_1: INV
 use entity primitive.INV(rtlA);
end for;

for INV_2: INV
 use entity primitive.INV(rtl);
end for;
```

S druge strane, mnogo je češći slučaj da sve komponente istog tipa imaju i istu arhitekturu. U tom slučaju se ključnom rečju ***all*** svim komponentama definiše ista konfiguracija:

Primer 2.10:

```
configuration NasePrimitive of Prvi_projekat is
  for structural

    for all: INV
      use entity primitive.INV(rtl);
    end for;

    for I_kolo: AND2
      use entity primitive.AND2(rtl);
    end for;

    for ILI_kolo: OR2
      use entity primitive.OR2(rtl);
    end for;

  end for;

end configuration NasePrimitive;
```

Treba napomenuti da sve **for** naredbe mogu da se hijerarhijski navode, tako da je moguće tačno definisati odgovarajuće arhitekture za svaki specifični poziv na neku komponentu u hijerarhijski podeljenom projektu.

Na kraju valja skrenuti pažnju da osim ovog načina konfigurisanja koji se naziva konfiguracija deklaracijom (configuration declaration), postoji i konfiguracija specifikacijom (configuration specification). Medjutim, s obzirom da za sada, ne podržavaju svi softveri za automatsku sintezu

konfiguraciju komponenata dobro je da se kunfigurisanje (**`configuration`** blok) zapamti kao poseban fajl koji će se pridružiti tokom simulacije, tako da se za sintezu i simulaciju koriste isti opisi projekta. O simulaciji na bazi VHDL-a biće reči u narednom poglavlju.
Često se prilikom opisa složenijih projekata kombinuju sva tri stila opisa arhitekture. Izbor pravog stila zavisi od nekoliko faktora, o čemu će biti više reči tokom kursa.
Ovde je dobro uspostaviti paralelu sa opisom nekog zadatka iz programiranja. Naime, moguće je rešiti isti problem na različite načine, ali zavisno od kodiranja programa neki od načina su više ili manje efikasni. Isto se odnosi i na opis projekta u VHDL-u. Jedan opis može da rezultira više ili manje složenim hardverom.