SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

PROJEKTIRANJE DIGITALNIH SUSTAVA

UVOD U PROGRAMSKI JEZIK VHDL

SINIŠA ŠEGVIĆ

Sadržaj

Popis slika

Popis tablica

Popis ispisa

Poglavlje 1

Uvod

Formalno modeliranje digitalnog sustava je korisno jer omogućava razmatranje sustava prije nego što je on fizički izgrađen. Model je po definiciji skup svojstava sustava koja su u danom kontekstu važna odnosno relevantna. Kako se sustavi u pravilu mogu razmatrati u većem broju konteksta, za isti sustav ima smisla graditi više modela. Tipični konteksti u kojima se koriste modeli digitalnih sustava su specifikacija zahtjeva, dokumentacija mogućnosti, testiranje i konačno formalna verifikacija. Pored toga, u posljednje se vrijeme sve više radi na automatskoj sintezi sklopova iz zadanog ponašajnog modela.

Tradicionalno, digitalni sustavi su se opisivali Booleovim jednadžbama i, u novije doba, shematskim dijagramima koji se unose putem računala. Pokazalo se da ti pristupi imaju slijedeće nedostatke:

- nemogućnost rada sa ponašajnim (nestrukturnim) modelima;
- slabiji rezultati sa velikim sustavima;

Pokazalo se da se drugi nedostatak počinje javljati kad broj jednadžbi odnosno logičkih vrata dosegne nekoliko tisuća. Međutim, najnoviji sklopovi se sastoje od milijuna vrata, dok njihove gustoće još uvijek rastu.

Glavni nedostatak se međutim očituje u tome što se strukturni model teško dobiva u ranoj fazi projekta, kad je poznata samo specifikacija sustava, a ne i izvedbeni detalji. Specifikacija se uvijek zadaje u obliku očekivanog ponašanja sustava za zadani skup okolnosti pa je tako potrebno specifikaciju prevesti u logičke jednadžbe manualnim metodama. To prevođenje se može u potpunosti izbjeći upotrebom *formalnih jezika za opis sklopovlja* (HDL - Hardware Description Language). Većina HDL alata omogućavaju opise sekvencijalnih kombinatornih sustava ponašajnim modelima — konačnim automatima odnosno tablicama istine. Takve specifikacije se mogu automatski prevesti u HDL kôd koji se opet može automatski prevesti u sklopovlje alatima za sintezu.

Iako se formalni jezici mogu koristiti i pri projektiranju integriranih sklopova, njihovo glavno područje primjene još uvijek je oblikovanje sustava temeljenih na programirljivim sklopovima (PLD, FPGA). Trenutno postoji više takvih jezika u širokoj upotrebi, ali najpopularniji su VHDL, Verilog i Abel.

Moderni digitalni sustav se može opisati na više razina opisa, od razine tehnološkog procesa do konačnog složenog sustava. Na svakoj od tih razina, sustav se može analizirati u terminima strukture i željenog ponašanja, kao što je prikazano tablicom 1.1. Stupci tablice označeni sa *sh*, *bj* i *HDL* označavaju područja upotrebe shematskih dijagrama, Booleovih jednadžbi odnosno formalnih jezika.

Razina opisa	strukturni opis	ponašajni opis	sh	bj	HDL
sustav		specifikacija performanse	*		
integrirani sklop	složene komponente (RAM, procesor, ...)	algoritmi, mikrooperacije	*		*
registar	registri, multiplekseri, ALU, ...	tablice istine, konačni automati	*		*
vrata	logička vrata, bistabili	Booleove jednadžbe	*	*	*
sklopovi	tranzistori, R, L, C	diferencijalne jednadžbe			
tehnološki proces	geometrijski objekti				

Tablica 1.1: Formalne metode za opis digitalnih sustava i njihova područja upotrebe.

Pored problema sa složenim sustavima, shematski modeli pate od međusobno nekompatibilnih datotečnih formata koje koriste različiti alati. Tako je tokom projektiranja sustava shematskim dijagramima jako teško promijeniti alat što otežava ionako složen posao. Idealan način projektiranja bi se stoga trebao temeljiti na otvorenom standardu koji pokriva čim veći broj razina opisa sustava, a VHDL predstavlja jedan od značajnijih rezultata napora u tom smjeru.

VHDL je skraćenica od "Very High Speed Integrated Circuit HDL", a razvijen je pod sponzorstvom ministarstva obrane SAD. Prvi simulatori su se pojavili početkom 90-ih godina, dok je u širu uportrebu došao oko 1994. Iako VHDL može pokriti i područje automatske sinteze, glavni razlozi kreiranja jezika su modeliranje, simulacija i dokumentacija složenih digitalnih sustava. Sinteza je dodana u kasnijoj fazi razvoja jezika, a njen rayvoj nažalost još nije gotov jer različiti alati mogu sintetizirati različite podskupove jezika.

Unatoč tome što je početni razvoj VHDL-a financiran od strane vojske sjedinjenih država, jezik je relativno brzo standardiziran od strane društva IEEE kao najutjecajnije profesionalne organizacije na tom području. Prvi standard se pojavio 1987, dok je 1993. godine objavljena njegova revizija, tako da danas postoji velik broj alata koji gotovo u potpunosti podržavaju standard iz 1993. Napori na standardizaciji se nastavljaju tako da su pored temeljnog standarda koji definira jezik (1076, VHDL), objavljeni i standardni format za razmjenu podataka za testiranje (1029, WAVES) te standardni tip podataka za viševrijednosnu logiku (1164, STD_LOGIC). U planu su standardi za matematičke operacije, biblioteke temeljnih programirljivih sklopova i sintezu.

opis na različitim razinama	sustavi mogu biti specificirani strukturnim i ponašajnim modelima (ili oboje)
pogodan za velike sustave	omogućava hijerarhijske pristupe projektiranju i ispitivanju složenih sustava
simulacija	postoje dobri simulacijski alati po pristupačnim cijenama
sinteza	postoje uglavnom funkcionalni alati za sintezu
široko područje primjene	VHDL nije ograničen na modeliranje elektronike

Tablica 1.2: Prednosti VHDL-a u odnosu na ostale formalne metode projektiranja.

Poglavlje 2

Temeljni pojmovi

2.1 Elementi modela složenog sustava

Kakvu god funkciju obavljao neki tehnički sustav, on na neki način mora pribaviti ulazne podatke iz svoje okoline, te isto tako dojaviti korisniku rezultate obrade. Drugim riječima, sustav mora komunicirati sa okolinom. Taj komunikacijski dio sustava se naziva sučeljem. Kvaliteta sučelja je vrlo bitno svojstvo sustava jer omogućava jednostavno korištenje i apstrakciju implementacijskih detalja. Sučelje se u VHDL-u definira deklaracijom entiteta (`entity`), koji je temeljna jedinica oblikovanja sustava. Kao što je besmisleno imati fizički sustav bez sučelja, tako se ne može graditi VHDL model bez opisa entiteta.

Zbog svoje široke prisutnosti i modularne arhitekture, sklopovlje osobnog računala je dobar odabir za metaforu složenog sustava. Na najvišoj razini, sučelje računala prema korisniku se sastoji od tipkovnice i prikazne jedinice.

U cilju postizanja željenog cilja, ulazni podaci moraju unutar sustava proći nekakvu transformaciju. Ta transformacija se zbiva u unutrašnjosti (tijelu, implementaciji) sustava, a u VHDL-u se opisuje arhitekturom (`architecture`) entiteta. Funkcionalnost sustava zadana implementacijom može biti jednostavna ili kompliciranа (npr. autopilot na putničkom avionu), ali se u svakom slučaju sustav može opisati pomoću sučelja i implementacije.

Sustavi mogu temeljiti neka svojstva na vanjskim standardnim komponentama. Tako se na sabirnicu osobnog računala može priključiti pločica sa specifičnim sklopovljem za obradu signala i tako poboljšati svojstva računala. Iako se takva dodatna pločica može tretirati kao dio implementacije računala, ima je smisla razmatrati odvojeno jer postupak njenog oblikovanja nije vezan samo za jedan konkretan tip računala.

Tri osnovna elementa modela sustava (sučelje, implementacija i dodatne komponente) se u VHDL-u tretiraju kao odvojene jedinice oblikovanja, a biti će detaljnije razmatrane u nastavku.

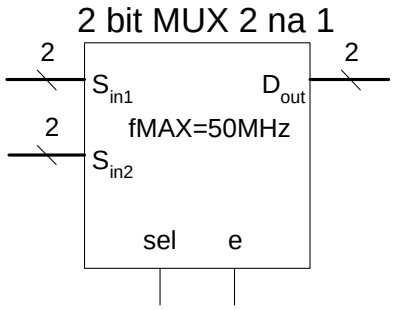
2.1.1 Sučelje sustava

Sasvim općenito, bilo kakvo projektiranje mora započeti analizom okoline u kojoj bi konačni proizvod trebao raditi. Definicija sučelja sustava je temeljni element njegovog opisa, pa se u kontekstu VHDL-a obično naziva entitetom.

Najčešći način dokumentiranja sučelja digitalnog sustava je korištenjem shematskih simbola, kao što je prikazano na sl.2.1.

Elementi sučelja multipleksera sa sl.2.1 se ugrubo mogu podijeliti u dvije glavne grupe:

- **podatkovne veze** koje prenose podatke iz sustava i prema sustavu.



Slika 2.1: Sučelje multipleksera “2 na 1”.

- **parametri sustava**, npr, širina sabirnice ili maksimalna radna frekvencija;

Dvije ključne komponente sučelja, podatkovne veze i parametri, imaju zasebne odjeljke unutar deklaracije entiteta, a oni se označavaju ključnim riječima `port` odnosno `generic`:

```
entity mux2na1 is
  generic(
    N: integer :=2;
    Td: time   :=20 ns
  );
  port(
    sin0: in  bit_vector(N-1 downto 0);
    sin1: in  bit_vector(N-1 downto 0);
    sel:  in  bit;
    e:    in  bit;
    dout: out bit_vector(N-1 downto 0)
  );
end entity mux2na1;
```

Ispis 2.1: Sučelje multipleksera “2 na 1”.

2.1.2 Implementacija sustava

Implementacija nekog sustava u VHDL-u se opisuje zasebnim odjeljkom (’arhitekturom’) koji se označava ključnom riječju `architecture`. Jezik naglašava važnost sučelja sustava na način da u zaglavlju arhitekture mora biti navedeno ime sučelja koje se implementira. Tako bi skica implementacije multipleksera bila:

```
-- komentar: moguće je definirati
-- više implementacija za jedno sučelje!
architecture A1 of Mux2na1 is
  -- ...
begin
  -- ...
end architecture A1;
```

Ispis 2.2: Skica izvedbe multipleksera “2 na 1”.

Implementacija sustava može biti dana u terminima njegovog ponašanja (*što sustav radi?*) i strukture (*kako postići željenu funkcionalnost?*). Obično se tokom razvoja koriste oba načina, na način da se očekivana funkcionalnost prvo specificira i formalizira ponašajnim modelom, da bi se taj model kasnije transformirao u strukturni ekvivalent koji je bliži alatima za sintezu stvarnog sklopovlja. Velik dio postupka sinteze se može obaviti automatski, međutim potpuno funkcionalna sinteza još nije dostupna.

Ponašajni opis sustava definira njegovu funkcionalnost, tj, kakve podatke očekujemo na izlazu, za zadani ulaz i stanje sustava. Ovakav opis ne daje nikakvu natuknicu o tome kako bi se zadana funkcionalnost trebala postići pa je jasno sa kakvim problemima se suočavaju alati za sintezu.

Strukturni opis definira koje komponente sustav koristi te kako bi one trebale biti međusobno spojene kako bi se postigao željeni rezultat. Komponente mogu biti različite složenosti, od podsustava sa zasebnim sučeljem i implementacijom do jednostavnih logičkih vrata. Strukturni opis je znatno lakše sintetizirati jer opisuje konkretne fizičke komponente. Međutim njegova izrada je obično složeniji i dugotrajniji proces nego što je to slučaj sa ponašajnim opisom.

Budući da različite implementacije mogu obavljati isti zadatak, jednom sučelju je moguće pridružiti više implementacija. Različite implementacije će biti korisne u različitim kontekstima, kao što su razvoj, testiranje ili sinteza.

2.1.3 Biblioteka standardnih komponenti

Neke temeljne komponente se javljaju u gotovo svim digitalnim sustavima: registri, brojila, multiplekseri itd. VHDL definira način za građenje biblioteka takvih komponenti i ostalih stvari koje mogu biti potrebne a nisu definirane standardom, upotrebom koncepta zvanog `package`.

Nažalost, standardizacija jezika nije još došla do te točke da je biblioteka bogata kao kod programskih jezika C++ ili Java, a sastoji se od sljedećih temeljnih paketa (package):

- `standard`: definira temeljne deklaracije i definicije jezičnih konstrukata, implicitno je uključen u sve VHDL modele;
- `textio`: definira tekstualne ulazno-izlazne operacije nad ugrađenim tipovima (koristi se pri testiranju modela);
- `std_logic_1164`: definira viševrijednosne logičke operacije (operacije sa vrijednostima 1,0,Z,X,...);
- `std_logic_textio`: definira tekstualne ulazno-izlazne operacije nad tipovima iz paketa `std_logic_1164`.

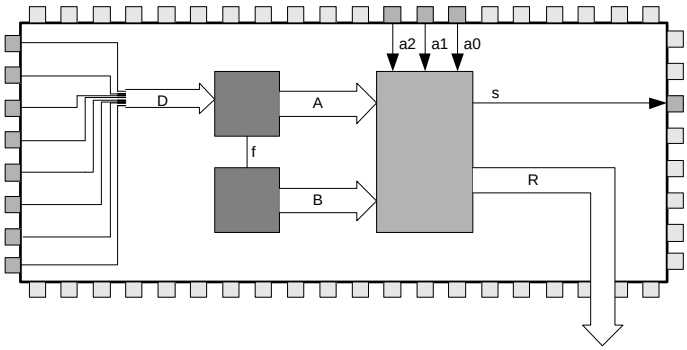
Da bi se objekti iz nekog paketa koristili, potrebno ih je prethodno deklarirati u dva koraka. U prvom koraku navodi se ime biblioteke, a u drugom konkretni objekt kojeg se želi koristiti:

```
library ImeBiblioteke;
use ImeBiblioteke.Paket5.Objekt;
```

Ispis 2.3: Sintaksa za uključivanje vanjskih paketa.

2.2 Pojam signala

U kontekstu VHDL-a, signal je koncept koji je jedinstveno pridružen nekoj fizičkoj liniji. Može se reći da signal odražava protok informacija kroz pridruženu liniju. Temeljna podjela signala koji se javljaju unutar neke komponente je na vanjske i unutrašnje. Vanjski signali su vidljivi i izvan matične komponente (tj, dio su njenog sučelja), dok se unutrašnji signali koriste isključivo u implementaciji komponente. Signali igraju ključnu ulogu u opisu komunikacije među komponentama složenog sustava i zato ovaj kratki opis VHDL-a počinje upravo od njih.




```
entity sustav is
  port(
    D:        in bit_vector(7 downto 0);
    a2,a1,a0: in bit;
    R: out bit_vector(7 downto 0);
    s: out bit
  );
end entity sustav

architecture struktura of sustav is
  signal A:bit_vector(7 downto 0);
  signal B:bit_vector(7 downto 0);
  signal f:bit;
begin
  ...
end architecture struktura
```

Slika 2.2: Signali u složenom digitalnom sustavu.

2.2.1 Višestruke signalne linije

Elektronički sustavi se temelje na komponentama koje su međusobno povezane signalnim linijama. Te signalne linije mogu biti izvedene kao jednostruke ili višestruke ožičene veze. Jednostruka ožičena veza odgovara liniji koja ima jedinstvenu logičku vrijednost u svakom trenutku. Primjer takvog signala je signal vremenskog vođenja koji sinkronizira sve komponente unutar sustava. Kod nekih sustava, javlja se potreba za višestrukim signalnim linijama (sabirnicama), koje prenose informacije kao kombinaciju logičkih vrijednosti. Tako na primjer kažemo 32-bitni procesori mogu baratati sa podacima širine 32 bita upravo zato jer im je se podatkovna sabirnica sastoji od 32 nezavisne signalne linije.

2.2.2 Temeljni tipovi signala

Formalna specifikacija signala u VHDL-u se obavlja pomoću dva temeljna tipa: `bit` (za jednostruke signalne linije) i `bit_vector` (za sabirnice). `bit_vector` umnogome odgovara polju iz klasičnih programskih jezika, jer postoje jezični konstrukti za pristup njegovim pojedinim elementima, a dimenzije im se ne mogu mijenjati tokom izvođenja simulacije. Oba tipa implicitno koriste binarnu logiku pa logička vrijednost pojedinačnih linija u svakom trenutku mora biti ili 0 ili 1.

Deklaracija jednostrukih signala je jednostavan posao: treba samo reći "tip signala `f` jest `bit`". Kod višestrukih signala, pored toga je potrebno odrediti i širinu sabirnice i redosljed bitova sabirnice koji određuje da li značajnost bitova raste s desna na lijevo (kao kod brojeva, big endian) ili s lijeva na desno (little endian). Tako, uz konvenciju da i-ti element sabirnice ima težinu 2^i, podatak `"10101010"` na sabirnici tipa `bit_vector(7 downto 0)` odgovara broju 170_{10}, dok je na sabirnici `bit_vector(0 to 7)` njegova interpretacija 53_{10}.

2.2.3 Deklaracija signala

Svaki objekt u VHDL programu mora prije upotrebe biti deklariran. Signali se deklariraju u ovisnosti o tome jesu li dio sučelja komponente (vanjski) ili se koriste isključivo unutar njenih granica (unutrašnji). Vanjski signali se deklariraju unutar `port` odjeljka entiteta, dok se unutrašnji deklariraju isključivo u arhitekturama komponente. Deklaracija signala se sastoji od jedinstvenog imena te tipa signala. Vanjski signali pored toga mogu imati specificiran i smjer toka podataka (podrazumijevani smjer je ulazni). Sintaksa deklaracije vanjskog signala stoga je:

```
entity xyz is
  generic(
    ...
  );
  port(
    ime1: smjer tip; -- vanjski signal
    ...
  );
end entity xyz;
```

Ispis 2.4: Skica deklaracije vanjskog signala.

Smjer toka podataka može biti ulazni (`in`), izlazni (`out`), ulazno-izlazni (`inout`), ili još neki rijeđe korišteni načini prijenosa.

Unutrašnji signali se deklariraju u implementaciji komponente, po sljedećoj sintaksi (smjer toka podataka se ne navodi):

```
architecture a3 of xyz is
  ...
  signal ime2: tip; -- unutrašnji signal
  ...
begin
  ...
end architecture a3;
```

Ispis 2.5: Skica deklaracije unutrašnjeg signala.

Leksički doseg signala je definiran mjestom deklaracije:

- signal deklariran u vanjskom paketu (`package`) je vidljiv u svim jedinicama koje paket koriste (`use`);
- signal deklariran u sučelju komponente `entity`, `port` se vidi u svim implementacijama te komponente;
- signal deklariran u deklarativnom dijelu implementacije `architecture` se vidi samo unutar te implementacije;

Ta pravila izravno slijede iz hijerarhijskog pristupa oblikovanju sustava: svemu što je deklarirano na nekoj hijerarhijskoj razini može se pristupiti unutar te razine i u svim hijerarhijski nižim razinama.

2.3 Modeli kašnjenja

Rezultat operacije se može pridružiti odredišnom signalu operatorom pridruživanja: <=. Valja obratiti pažnju da isti simbol <= označava i relacijski operator te da interpretacija ovisi o kontekstu.

```
architecture A3 of E1 is
  signal sel: bit;
  ...
  sel <= '1';
end architecture A3
```

Ispis 2.6: Pridruživanje vrijednosti signalu.

Stvarne komponente i veze uvijek imaju konačno vrijeme propagacije signala, a važnost tog kašnjenja jako ovisi o konkretnoj situaciji. VHDL omogućava ponašajni opis kašnjenja na razini pridruživanja vrijednosti pojedinim signalima ključnom rječju `after`. Pri tome je omogućeno modeliranje kašnjenja na dva načina: inercijskim i transportnim modelom. Neka je zadan primjer:

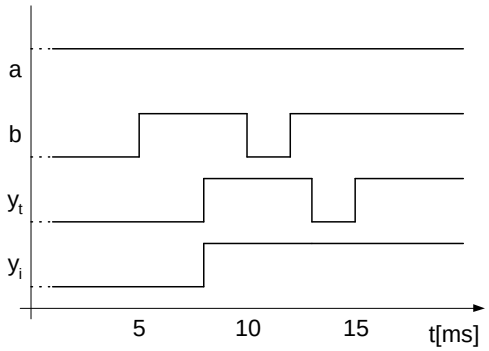
```
entity and2 is
  port(
    a,b: in bit;
    y: out bit);
end entity and2

architecture transport of and2 is
  y <= transport a and b after 3 ns;
end architecture transport

architecture inertial of and2 is
  -- ključna riječ inertial se može izostaviti
  y <= inertial a and b after 3 ns;
end architecture inertial
```

Ispis 2.7: Modeli kašnjenja signala.

Tada bismo mogli očekivati ovakav slijed događaja:



Slika 2.3: Odziv komponenti sa transportnim i inercijalnim modelima kašnjenja

Karakteristika inercijalnog modela je da dvije uzastopne promjene na ulazu u vremenu koje je kraće od vremena propagacije signala ne utiču na izlaz. Taj model dobro opisuje većinu stvarnih sustava pa se on podrazumijeva ako se model eksplicitno ne naznači.

U transportnom modelu, svaka promjena na ulazu utječe na izlaz, neovisno o duljini vremenskog intervala u kojem se promjena događa. Taj model stoga dobro opisuje kašnjenje prospojnih linija.

2.4 Tipovi podataka

Svaki podatak u digitalnom sustavu se pohranjuje u obliku jednog ili više bitova pa su zato `bit` i `bit_vector` temeljni tipovi podataka u VHDL-u. Međutim, takav oblik je često nepraktičan za upotrebu jer složeniji sustavi obično rade sa podacima od 8, 16 i više bita. Mnogo je lakše definirati vrijednost na sabirnici ASCII znakom ili heksadekadskim brojem nego nizom bitova. Slično, složene podatkovne strukture se često mogu elegantno opisati poljima i strukturama pa VHDL predviđa složene tipove `array` i `record`. Takvi složeni tipovi podataka su, pored opisa podataka koji se fizički javljaju u sustavu (signali), vrlo korisni i za opis međuvrijednosti u ponašajnom modelu sustava koje nakon optimizirajuće sinteze ne moraju imati i fizičko mjesto u sustavu (varijable, konstante).

2.4.1 Skalarni tipovi

Skalarni tipovi podataka se sastoje od konačnog uređenog skupa dozvoljenih vrijednosti koje nisu strukturirane. Skalarni tipovi se dijele na diskretne (npr. `boolean`, `character`, `integer`), fizičke (`time`) i tipove s pomičnim zarezom (`real`). Za svaki skalarni tip moguće je izvesti podtipove koji odgovaraju nekom povezanom intervalu osnovnog tipa.

Slično kao i u konvencionalnim programskim jezicima, moguće je definirati pobrojane korisničke tipove. Takvi tipovi su posebno korisni za opis stanja nekog konačnog automata. Tako bi se stanje nekog procesora moglo opisati sljedećim tipom:

```
type CPUstate is (Fetch, Decode, FetchOperand,
                  Execute, WriteBack);
```

Fizički tipovi se razlikuju od diskretnih po tome što su objekti određeni parom (vrijednost, jedinica), što računske operacije nad takvim tipovima uzimaju u obzir. VHDL standard predviđa samo jedan fizički tip - time (vrijeme), a on je u standardnoj biblioteci definiran kao (granice intervala dozvoljenih vrijednosti tipova najčešće nisu definirane standardom):

```
type time is range -2147483647 to 2147483647
  units
    fs;
    ps  = 1000 fs;
    ns  = 1000 ps;
    us  = 1000 ns;
    ms  = 1000 us;
    sec = 1000 ms;
    min =   60 sec;
    hr  =   60 min;
  end units;
```

Ispis 2.8: Formalna deklaracija tipa podataka za opis vremena.

Iz deklaracije se vidi da postoje dvije vrste jedinica, primarna (`fs`) i izvedene koje su definirane kao cjelobrojni višekratnici primarne jedinice. Interval dozvoljenih vrijednosti u definiciji tipa se, naravno, odnosi na primarnu jedinicu.

2.4.2 Složeni tipovi

VHDL predviđa višedimenzionalna polja s proizvoljnim intervalima dozvoljenih indeksa. Na primjer, skup pristupnih linija memorijskog polja od 16 puta po 8 bitova se može deklarirati kako slijedi:

```
type MemField: array(0 to 15, 0 to 7) of bit;
signal mf: MemField;
...
mem(0,0) <- '1';
```

Tip `bit_vector` je definiran u standardnoj biblioteci kao:

```
subtype natural is integer range 0 to 2147483647;
...
type bit_vector is array (natural range<>) of bit;
```

Drugi predefinirani tip polja je niz znakova:

```
subtype positive is integer range 1 to 2147483647;
...
type string is array (positive range<>) of character;
```

Strukturirani tipovi se tvore ključnom rječju `record`. Primjerice, strukturirani tip za pohranjivanje informacija o pribavljenoj instrukciji u procesoru bi bio:

```
type opcodes is (add, sub, load, ...);
-- neka je NRegisters zadan kao parametar (generic)!
type register is range 0 to NRegisters;
type instruction is record
  opcode: opcodes;
  reg   : register;
  offset: integer;
end record instruction;
```

2.5 Operacije nad podacima

Slično modernim konvencionalnim programskim jezicima, kombinacije tipova operanada nad kojima se operator može primijeniti su strogo određene. Tako je operacija zbrajanja definirana isključivo za parove operanada koji imaju iste numeričke tipove (npr. `integer`, `real`). Svaka dozvoljena kombinacija određuje semantiku (značenje) operacije koja se nad izrazom primjenjuje. Područje definicije operatora se može proširiti (overload) na nove kombinacije tipova uvođenjem posebnih procedura koje definiraju odgovarajuću semantiku.

Logički operatori

VHDL definira 6 binarnih (`and`, `or`, `nand`, `nor`, `xor` i `xnor`), i jedan unarni logički operator (`not`) sa uobičajenom semantikom. Oba operanda moraju biti istog tipa, koji je onda jednak i tipu rezultata operacije. Operandi su definirani nad tipovima `bit`, `boolean` i `bit_vector`.

Aritmetički operatori

Predefiniran je uobičajeni skup aritmetičkih operatora `+`, `-`, `*`, `/`, `mod`, `rem`, `abs` i `**`. Dozvoljeni tipovi operanada su `integer` i `real`. Tipovi oba operanda moraju biti isti, osim kod operatora potencije `**` kojem drugi argument mora biti `integer`. Dodatna iznimka su fizički tipovi (npr. `time`) za koje drugi operand pri množenju i dijeljenju mora biti numerički tip, dok je tip rezultata jednak tipu prvog operanda.

Relacijski operatori

Predefiniran je uobičajeni skup relacijskih operatora `=`, `/=`, `<`, `<=`, `>` i `>=`. Oba operanda moraju biti istog tipa, a rezultat je tipa `boolean`. Operator je definiran nad tipovima `boolean`, `bit`, `character`, `integer`, `real`, `time`, `string` i `bit_vector`.

Operatori posmaka

Predviđen je skup od šest operatora posmaka koji se mogu primijeniti samo na jednodimenzionalna polja tipova `bit` i `boolean`. Lijevi argument operacije je polje dok je desni cijeli broji koji označava željeni broj posmaka. Kod logičkih posmaka, ispražnjena mjesta se popunjavaju s vrijednostima `0` ili `false`, ovisno o tipu polja. Kod aritmetičkih posmaka, na ta se mjesta upisuju vrijednosti koje odgovaraju najbližem elementu početnog polja.

operator	naziv	primjer za $10010110_{(2)}$
`sll`	logički posmak ulijevo	00101100
`sla`	aritmetički posmak ulijevo	00101100
`rol`	kružni posmak ulijevo	00101101
`srl`	logički posmak udesno	01001011
`sra`	aritmetički posmak udesno	11001011
`ror`	kružni posmak udesno	01001011

Tablica 2.1: Operatori posmaka.

Operator povezivanja

Operator povezivanja "`&`" tvori jednodimenzionalna polja povezivajući svoj lijevi i desni operand. Operandi mogu biti skalarni tipovi ili jednodimenzionalna polja sa prikladnim tipom elemenata.

lijevi operand	desni operand	rezultat
"abc"	’d’	"abcd"
b"0"	’1’	b"01"
X"F"	’0’	b"11110"

Tablica 2.2: Primjeri korištenja operatora povezivanja.

Poglavlje 3

Modeliranje ponašanja sustava

3.1 Deklaracija entiteta

Formalna sintaksa deklaracije entiteta je:

```
deklaracija_entiteta ⇐
  entity ime_komponente is
    [ generic( lista_parametara ); ]
    [ port( lista_vanjskih_signala ); ]
    -- ...
  end entity ime_komponente;
```

Ispis 3.1: Sintaksa deklaracije entiteta.

U gornjem primjeru, `ime_komponente` identificira komponentu, a tvori se od slova, brojki i podvlaka kao što je uobičajeno u konvencionalnim programskim jezicima. Dobra programerska praksa je dokumentirati važne odluke pri oblikovanju složenog sustava komentarima. Komentari u VHDL-u počinju sa dvije crtice, a završavaju na kraju tekuće linije izvornog kôda. Komentari bi trebali opisivati način korištenja svakog objekta koji se javlja u sučelju komponente. Pored toga, komentari trebaju dokumentirati i važne odluke pri oblikovanju sučelja, alternativne mogućnosti te po čemu je odabrana metoda bolja (gora) od alternativa. Česta je praksa da se oblikuje uniformirano zaglavlje koje onda mora biti uključeno na početku svake datoteke koja je dio složenog sustava. Takvo zaglavlje obično predviđa ispunjavanje slijedećih podataka o sadržaju datoteke: imena komponenti koje su definirane u datoteci (najčešće jedna), namjena komponente, ograničenja, nedostaci, greške, korištene biblioteke, autori, naziv simulatora s kojim je obavljeno testiranje.

Odjeljak `port` deklaracije entiteta definira signale koji čine sučelje komponente. Svaki vanjski signal bi trebao imati pažljivo odabrano ime koje opisuje njegovu funkciju. Detaljniji opis signala bi trebao biti naznačen u komentaru iznad ili na kraju linije koja sadrži signal.

Odjeljak `generic` sadrži generičke parametre komponente koji moraju biti specifirani od strane okoline koja dotičnu komponentu koristi. Parametri `generic` odjeljka omogućuju pisanje parametriziranih komponenti korištenjem parametara kao što su širina sabirnice ili vrijeme kašnjenja građevnih komponenata. Parametriziranjem se dobivaju komponente koje se mogu upotrijebiti u većem broju konteksta na način koji je sličan (iako manje moćan) korištenju `template` mehanizama u jeziku `C++`. Parametri komponente se zadaju u obliku liste odvojene točkom i zarezom, pri čemu svaki element liste ima oblik:

```
ime: tip [:=podrazumijevana\_vrijednost].
```

Za primjer, sučelje komponente koja opisuje N-bitni zaporni sklop je dano ispisom 3.2.

```
-- ovo je n-bitovni registar sa upisom na
-- uzlaznom bridu signala vremenskog vođenja
entity reg_n is
  -- konvencija: parametri imaju sufiks _g
  generic(
    cb_g: integer :=4;    -- broj bitova registra
    td_g: time :=100 ns   -- kašnjenje pri upisu
  );
  -- konvencija: vanjski signali imaju sufiks _p
  port(
    -- ulazna riječ
    d_p:   in  bit_vector(cb_g-1 downto 0);
    -- signal vrmenskog vođenja
    clk_p: in  bit;
    -- izlazna riječ
    q_p:   out bit_vector(cb_g-1 downto 0)
  );
end entity reg_n;
```

Ispis 3.2: Sučelje zapornog sklopa.

3.2 Opis sekvencijalnog ponašanja sustava

Pri razvoju ponašajnog modela sustava, često se javlja potreba za opisom slijednih operacija koje valja izvesti kako bi se postigao traženi rezultat. Sljedeći analogiju iz operacijskih sustava, jezični konstrukt za opis takvih operacija u VHDL-u se naziva procesom. Formalna sintaksa opisa procesa je:

```
stavak_procesa ⇐
  naziv: process (lista osjetljivosti)
    deklaracije...
  begin
    sljedne_naredbe
  end process naziv;
```

Ispis 3.3: Stavak za opis procesa.

U ispisu 3.3, javljaju se sljedeći novi pojmovi:

- `naziv`: opcionalni identifikator;
- `lista osjetljivosti`: definira signale koji utječu na pokretanje procesa;
- `deklaracije`: varijable i konstante koje se javljaju unutar bloka;
- `sljed operacija`: lista operacija koje se izvode slijedno.

Upotreba procesa se može ilustrirati na primjeru multipleksera “2 na 1”:

```
entity mux2na1 is
  generic(
    N: integer :=2;
    Td: time   :=20 ns
  ); port(
    sin0: in  bit;
    sin1: in  bit;
    sel:  in  bit;
    dout: out bit
  );
end entity mux2na1;

architecture sljedna of mux2na1 is
begin
  p: process(sin0,sin1,sel)
  begin
    if (sel='1') then
      dout <= sin1;
    else
      dout <= sin0;
    end if;
  end process p;
end architecture sljedna;
```

Ispis 3.4: Izvedba multipleksera “2 na 1” upotrebom procesa.

3.2.1 Tijek izvođenja naredbi procesa

Za razliku od računalnih programa, elektroničke komponente ne poznaju termine kao što su početak ili kraj izvođenja. Registri, multiplekseri, memorije, itd, su uvijek u pogonu, osluškujući ulazne linije i postavljajući odgovarajuće vrijednosti na izlaznim linijama. Zato se, ako se ne naznači drukčije, operacije `process` bloka obavljaju kontinuirano od početka prema kraju i onda ponovo na početak.

Ponašanje digitalnog sustava se često može opisati slijedećim pseudokôdom:

```
čekaj prozivanje;
  (npr, dekodiranjem viših bitova adresne sabirnice)
analiziraj ulaz;
  (npr, dekodiranjem nižih bitova adresne sabirnice)
postavi izlaz;
  (npr, postavljanjem podatka na sabirnicu podataka).
```

Za omogućavanje takvih opisa, u VHDL je uvedena naredba `wait <uvjet>`. Kad slijed izvođenja narebi procesa dođe na tu naredbu, izvođenje procesa se privremeno prekida sve dok se uvjet ne ispuni. Predviđena su tri načina za specificiranje uvjeta:

- čekanje da istekne zadani vremenski interval:
  - `wait for 10 ns;`
  - `wait for Clk/2;`

- čekanje da se ispuni logički uvjet:
  - `wait until Clk='1';`
  - `wait until Enable and (not Sel);`

  (uvjet se provjerava tek kad se neki od argumenata promijeni!):
- čekanje da se jedan od zadanih signala promijeni:
  - `wait on Clk;`

Naredbe `wait` se mogu nalaziti bilo gdje unutar procesa, međutim najčešće će biti jedna naredba oblika `wait on` na samom kraju procesa, pa je za tu kombinaciju predviđena specijalna sintaksa korištenjem liste osjetljivosti.

```
naziv: process(                          naziv: process
  lista_osjetljivosti                      deklaracije...
)                                        begin
  deklaracije...            ≡            sljed operacija
begin                                      wait on
  sljed operacija                          lista_osjetljivosti;
end process naziv;                       end process naziv;
```

Slika 3.1: Lista osjetljivosti i ekvivalentna `wait on` naredba na kraju procesa.

3.2.2 Korištenje varijabli u procesima

Što se modeliranja i simulacije tiče, vrijeme izvođenja procesa između dvije `wait` naredbe je trenutno. Iz toga slijede bitna ograničenja na korištenje signala u procesima:

1. signali ne mogu biti deklarirani unutar procesa;
2. bilo koje pridruživanje vrijednosti signalu dolazi do izražaja tek kad se izvođenje procesa privremeno prekine;
3. u slučaju višestrukog pridruživanja nekom signalu, samo će se posljednje pridruživanje stvarno i izvesti.

Problemi koji mogu nastati ne vođenjem računa o neintuitivnoj semantici pridruživanja signalima unutar procesa su ilustrirani na sl.3.2.

Iz opisanog slijedi bitna posljedica da signali ne mogu biti korišteni za spremanje privremenih vrijednosti unutar procesa, pa su za tu svrhu u VHDL uvedene varijable. Varijable su vrlo slične signalima, a mogu se upotrebljavati isključivo unutar procesa. Ključna razlika između varijable i signala je što se pridruživanje varijabli (koristi se poseban operator `:=`) unutar procesa događa odmah. Varijable omogućavaju pisanje intuitivno nedvosmislenih procesa, poštivanjem sljedećih pravila:

- na početku složenog procesa, svakom izlaznom signalu se dodijeljuje privremena varijabla;
- svi međurezultati se upisuju u privremene varijable;

```
process(C,D)
begin
  E <= 3;
  A <= D+1;
  B <= A+C; -- ???
  E <= A*2; -- ???
end process;
```

	A	B	C	D	E
početna vrijednost	2	3	1	1	4
promjena			1	2	
krajnja vrijednost	3	3	1	2	4

Slika 3.2: "Dvosmisleno" pridruživanje vrijednosti signalu unutar procesa.

```
process(C,D)
  variable Av,Bv,Ev: integer;
begin
  Ev := 3;
  Av := D+1;
  Bv := Av+C;
  Ev := Av*2;
  E <= Ev;
  A <= Av;
  B <= Bv;
end process;
```

	A	B	C	D	E
početna vrijednost	2	3	1	1	4
promjena			1	2	
krajnja vrijednost	3	4	1	2	6

Slika 3.3: "Dvosmisleno" pridruživanje vrijednosti signalu unutar procesa — rješenje.

- na samom kraju procesa, vrijednosti iz privremenih varijabli se upisuju u odredišne signale uz procijenjeno kašnjenje.

Primjena tih pravila na rješenje problema sa sl.3.2 je prikazana na sl.3.3.
Razlike i sličnosti među varijablama i signalima su sažete u tablici 3.1.

	mjesto deklaracije	pridruživanje u procesu	kašnjenje
signali	-port odjeljak sučelja, -deklaracijski dio izvedbe, -ne unutar procesa!	-efekti su vidljivi tek nakon izvršavanja naredbe `wait`;	-podržano (`after`);
varijable	-unutar procesa, -unutar potprograma;	-efekti su vidljivi odmah;	-nema smisla, jednako je nuli.

Tablica 3.1: Usporedba signala i varijabli unutar procesa.

3.2.3 Naredbe za upravljanje izvođenjem procesa

Pored pridruživanja pojedinim signalima i varijablama, definiran je veći broj naredbi za upravljanje tokom izvođenja procesa. Te naredbe su isključivo sljednog karaktera i stoga se mogu koristiti isključivo unutar procesa.

Uvjetno izvođenje (`if`)

Pojednostavljena sintaksa:

```
if izraz_1 then
  sljedne_naredbe
{ elsif izraz_i then
    sljedne_naredbe
}
[ else
    sljedne_naredbe
]
end if;
```

Ispis 3.5: Skica sintakse uvjetne naredbe.

Primjer: D bistabil sa `reset` ulazom.

```
entity ffD_1 is
  generic(
    td_g: time :=100 ns);
  port(
    d_p:   in  bit;
    clk_p: in  bit;
    rst_p: in  bit;
    q_p:   out bit);
begin
end entity ffD_1;

architecture beh of ffD_1 is
begin
  ff: process(clk_p) begin
    if (rst_p='1') then
      q_p <= '0';
    elsif (clk_p'event and clk_p='1') then
      q_p <= d_p;
    end if;
  end process;
end architecture beh;
```

Ispis 3.6: D bistabil sa `reset` ulazom.

Uvjetno izvođenje s višestrukim izborima (`case`)

Pojednostavljena sintaksa:

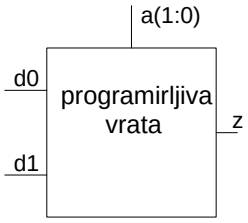
```
case izraz is
when izbor_1 =>
  sljedne_naredbe
{ when izbor_i =>
    sljedne_naredbe
}
```

```
  end case;

  izbor ⇐
    (izraz | diskretni_interval | others) {| ...}
  diskretni_interval ⇐
    izraz (to | downto) izraz
```

Ispis 3.7: Skica sintakse uvjetne naredbe s višestrukim izborima.

Primjer: programirljiva vrata zadana slikom 3.4.



a(1)	a(0)	operacija
0	0	i
0	1	ili
1	0	iskljucivo ili
1	1	ni

Slika 3.4: Programirljiva logička vrata

```
entity gate is
  port(
    a_p:  in  bit_vector(1 downto 0);
    d0_p: in  bit;
    d1_p: in  bit;
    z_p:  out bit);
begin
end entity gate;

architecture beh of gate is
begin
  pg: process (a_p, d0_p, d1_p)
  begin
    case a_p is
      when "00" =>   z_p <= d0_p and  d1_p;
      when "01" =>   z_p <= d0_p or   d1_p;
      when "10" =>   z_p <= d0_p xor  d1_p;
      when others => z_p <= d0_p nand d1_p;
    end case;
  end process pg;
end architecture beh;
```

Ispis 3.8: Izvedba programirljivih logičkih vrata naredbom `case`.

Uvjetna petlja (`while`)

Pojednostavljena sintaksa:

```
  while izraz loop
    sljedne_naredbe
```

```
end loop;
```

Ispis 3.9: Skica sintakse uvjetne petlje.

Petlja s brojačem (`for`)

Pojednostavljena sintaksa:

```
for identifikator in diskretni_interval
loop
  sljedne_naredbe
end loop;

diskretni_interval ⇐
  izraz (to | downto) izraz

-- identifikator ne bi smio biti prethodno deklariran,
-- a unutar petlje se ponaša kao konstanta;
```

Ispis 3.10: Skica sintakse petlje s brojačem.

Primjer: prospajanje izvorne na odredišnu sabirnicu uz preokretanje redoslijeda bitova.

```
-- pretpostavlja se da su SrcBus i DstBus deklarirane
-- kao signali širine N linija, pri čemu je N konstanta
-- ili parametar komponente (generic)
okreni: process (SrcBus)
begin
  for i in 0 to N-1 loop
    DstBus(N-1-i) <= SrcBus(i);
  end loop;
end process okreni;
```

Ispis 3.11: Preokretanje redosljeda bitova sabirnice pomoću petlje s brojačem.

Dodatno upravljanje petljama (`next`, `exit`)

Pojednostavljena sintaksa:

```
next [ when uvjet ] ;

exit [ when uvjet ] ;
```

Ispis 3.12: Skica sintakse naredbi za upravljanje petljama.

Rani izlazak iz petlje se postiže naredbom `exit`, koja je analogna naredbi `break` u `C`-u. Preskakanje ostatka tijela petlje se postiže naredbom **next** koja je stoga analogna naredbi `continue` u `C`-u.

3.3 Opis usporednih operacija u sustavu

Složeni sustavi se najčešće ne mogu opisati samo jednim slijednim algoritmom. Zato VHDL omogućava izvedbe sa većim brojem jednakopravnih procesa koji se usporedno[1] izvode. U skladu s tim, opća sintaksa za opis izvedbe sustava glasi:

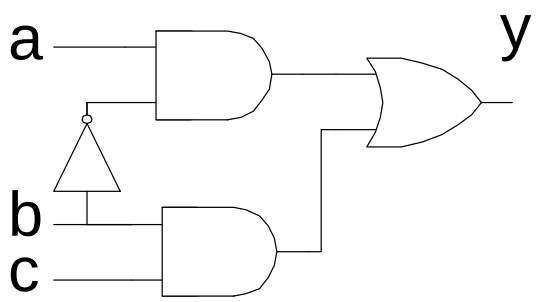
```
izvedba ⇐
  architecture ime_izvedbe of ime_entiteta is
    { deklaracija }
  begin
    { usporedna_naredba }
  end architecture [ime_izvedbe]

usporedna_naredba ⇐
  proces |
  usporedno pridruživanje |
  instanciranje komponente |
  naredba generate |
  ...
```

Ispis 3.13: Skica sintakse petlje s brojačem.

Naredbe iz tijela arhitekture modeliraju odvojene autonomne dijelove sustava, pa se prilikom simulacije "izvršavaju" istodobno.

Za primjer se može uzeti sustav sa sl.3.5, na način da se pojedine komponente sklopa opišu `process` blokovima. Takav opis bi mogao biti koristan za analizu hazarda koji javlja pri prijelazu `abc: 111` $\rightarrow$ `101`.



Slika 3.5: Jednostavan logički sklop.

```
entity sklop is
  generic(
    td: time := 10 ns);
  port(
```

[1] usporedan (*engl.* concurrent): koji ide, teče, kreće se jedan uz drugoga, Rječnik hrvatskoga jezika, Školska knjiga, Zagreb 2000.

```
    a,b,c: in  bit;
    y:     out bit);
end entity sklop;

architecture strukturna of sklop is
  signal and_1, or_1, or_2: bit;
begin
  vrata_i: process(and_1,a) is
  begin
    or_1 <= and_1 and a after td;
  end process vrata_i;

  vrata_i2: process(b,c) is
  begin
    or_2 <= b and c after td;
  end process vrata_i2;

  vrata_ili: process(or_1,or_2) is
  begin
    y <= or_1 or or_2 after td;
  end process vrata_ili;

  vrata_ne: process(b) is
  begin
    and_1 <= not b after td;
  end process vrata_ne;
end architecture strukturna;
```

Ispis 3.14: VHDL model za sklop sa sl.3.5.

U stvarnim modelima, rijetko će se javiti potreba za višestrukim procesima u jednoj komponenti. Pokazat će se da se koncizniji i jasniji opisi mogu postići usporednim pridruživanjem odnosno hijerarhijskom organizacijom.

3.3.1 Usporedne naredbe pridruživanja

Pri modeliranju stvarnih sustava, često se javlja potreba za procesima sa samo jednom naredbom. Za takve prilike, VHDL predviđa tzv. usporednu naredbu pridruživanja, koja je vrlo slična slijednom pridruživanju koje se koristi unutar procesa. Sljedeća izvedba komponente `sklop` je ekvivalentna prethodnoj, ali konciznija i jasnija.

```
architecture strukturna2 of sklop is
  signal and_1, or_1, or_2: bit;
begin
  or_1 <= and_1 and a after td;
  or_2 <= b and c after td;
  y <= or_1 or or_2 after td;
  and_1 <= not b after td;
end architecture strukturna2;
```

Ispis 3.15: VHDL model za sklop sa sl.3.5.

Usporedno pridruživanje je ekvivalentno procesu sa odgovarajućim slijednim pridruživanjem kao jedinom naredbom, a u čijoj se listi osjetljivosti nalaze signali koji se javljaju na desnoj strani pridruživanja.

Uvjetno pridruživanje

VHDL predviđa dva načina za specificiranje uvjeta za usporedno pridruživanje. Sintaksa prvog od njih jest:

```
uvjetno_pridruživanje ⇐
  signal <= [ model_kašnjenja ]
  { valni_oblik when uvjet else }
  valni_oblik when uvjet;

valni_oblik ⇐
  izraz [ after vremenski_izraz] {, ... }
```

Ispis 3.16: Skica sintakse uvjetnog usporednog pridruživanja.

Primjer korištenja u definiciji jednostavnog multipleksera:

```
architecture usporedna of mux2na1 is
begin
  dout <= sin0 when sel='0' else
          sin1;
end architecture usporedna;
```

Ispis 3.17: Izvedba multipleksera "2 na 1" upotrebom procesa.

Uvjetno pridruživanje sa odabirom

Uvjetno pridruživanje sa odabirom je slično `case` naredbi programskog jezika `C`, a ima sljedeću sintaksu:

```
pridruživanje_sa_odabirom ⇐
  with izraz select
    signal <= [ model_kašnjenja ]
    { valni_oblik when vrijednosti, }
    valni_oblik when vrijednosti;

vrijednosti ⇐
  izraz {| izraz} |
  izraz1 (to | downto) izraz2
```

Ispis 3.18: Skica sintakse uvjetnog usporednog pridruživanja.

3.3.2 Atributi signala

Atributi signala su specijalne funkcije kojima je moguće saznati detalje o kretanju vrijednosti signala kroz povijest. Sintaksa za poziv atributa `a` nekog signala `S` jest `S'a(parametri)`, dok je semantika nekih ugrađenih atributa dana tablicom 3.2.

S'active()	vrijednost tipa boolean koja je istinita ako se u tekućem simulacijskom ciklusu događa transakcija nad signalom S, tj. pridruživanje neke vrijednosti tom signalu;
S'event()	vrijednost tipa boolean koja je istinita ako se u tekućem simulacijskom ciklusu događa transakcija nad signalom S, koja mijenja vrijednost signala;
S'transaction()	signal tipa bit koji mijenja vrijednost u trenucima u kojima se događa transakcija na signalu S;
S'delayed(T)	signal koji odgovara signalu S, s kašnjenjem u vremenu T;
S'quiet(T)	vrijednost tipa boolean koja je istinita ako se u proteklom vremenu T nije dogodila transakcija nad S;
S'stable(T)	vrijednost tipa boolean koja je istinita ako se u proteklom vremenu T vrijednost signala S nije promijenila;
S'last_transaction()	vrijeme proteklo od posljednje transakcije nad signalom S;
S'last_event()	vrijeme proteklo od posljednje promjene signala S;
S'last_value()	vrijednost signala S prije posljednje promjene.

Tablica 3.2: Ugrađeni atributi signala.

Upotreba atributa signala

Najčešća upotreba signala je pri ponašajnom modeliranju detekcije bridova, kao što je prikazano ispisom 3.19. Valja ipak obratiti pažnju na to da je prikazani mehanizam prikladan samo za signale binarnog tipa, a ne i za realističnije tipove signala koji, pored uobičajenih logičkih stanja '0' i '1', omogućavaju modeliranje stanja visoke impedancije 'Z' i sl. Tipovi podataka koji podržavaju takvu viševrijednosnu logiku će biti detaljnije razmatrani u odjeljku 3.3.3.

```
-- uzlazni brid signala clk
if clk_s'event and clk_s='1' then
  ...
end if;

-- silazni brid signala CLK
if clk_s'event and clk_s='0' then
  ...
end if;
```

Ispis 3.19: Upotreba atributa signala za detekciju bridova.

Bistabili obično imaju dvije vrste ulaznih signala: jedan okidni signal (obično je to signal vremenskog vođenja), te jedan ili više upravljačkih signala. Da bi se željena operacija uspješno dekodirala, upravljački signali moraju biti stabilni u nekom vremenu tsetup prije aktivnog brida okidnog signala. Taj uvjet bi se mogao provjeriti slijedećim slijednim odsječkom:

```
if clk_s'event and clk_s='1'
  assert(clk_s'last_event > tSetup);
  ...
end if;
```

Ispis 3.20: Upotreba atributa signala za provjeru vremenskih zahtjeva.

Atributi polja

Slično signalima, predviđeni su i atributi za objekte složenog tipa polje. Semantika nekih ugrađenih atributa polja je dana tablicom 3.3 (podrazumijevana vrijednost jedinog parametra za sve atribute jest 1):

`A'left(N)`	lijeva granica intervala dozvoljenih indeksa za dimenziju N polja A;
`A'right(N)`	desna granica intervala dozvoljenih indeksa za dimenziju N polja A;
`A'low(N)`	donja granica intervala dozvoljenih indeksa za dimenziju N polja A;
`A'high(N)`	gornja granica intervala dozvoljenih indeksa za dimenziju N polja A;
`A'range(N)`	interval dozvoljenih indeksa za dimenziju N polja A;
`A'reverse_range(N)`	preokrenuti interval dozvoljenih indeksa za dimenziju N polja A;
`A'length(N)`	duljina intervala dozvoljenih indeksa za dimenziju N polja A;
`A'ascending(N)`	istina, ako je A'range(N) uzlazni interval (`to`);

Tablica 3.3: Ugrađeni atributi polja.

Tako se npr. petlja po svim dozvoljenim indeksima jednodimenzionalnog polja A, odnosno postavljanje svih linija polja na logičku nulu, dobiva jednom od sljedećih slijednih naredbi:

```
-- petlja po dozvoljenim indeksima polja A, unaprijed:
for i in A'range loop
  ...

-- petlja po dozvoljenim indeksima, unatrag:
for i in A'reverse_range loop
  ...

-- postavljanje svih linija polja u logičku nulu:
  A <= (A'range => 'Z');
```

Ispis 3.21: Upotreba atributa polja.

3.3.3 Viševrijednosna logika i razrješavanje konflikata

Često se pri modeliranju elektroničkih sustava javlja potreba da više izvora bude spojeno na jedan te isti sustav. Primjeri uključuju složene sustave gdje je više jedinica spojeno na jednu podatkovnu sabirnicu (CPU, RAM, ROM, DMA), ali i jednostavne sustave sa ožičenim "i" i "ili" spojevima (npr, kod TTL sklopova sa otvorenim kolektorom). Kod sabirnica će obično samo jedan od "govornika" biti aktivan u svakom trenutku simulacije, dok će kod ožičenih spojeva, uvijek biti aktivni svi govornici. U svakom od tih slučajeva, međutim, biti će potrebno definirati pravila (funkciju razrješavanja) za određivanje vrijednosti spojenog signala na temelju pojedinačnih vrijednosti.

Binarna logika nije dovoljna za opis funkcije razrješavanja jer nema opravdane interpretacije signala dobivenog spajanjem izlaza dvaju sklopova koji postavljaju različite vrijednosti (`'0'` i `'1'`). Iz toga slijedi da se u VHDL-u samo upotrebom tipova `bit` i `bit_vector` ne mogu opisati ni mikroprocesor ni sklop sa otvorenim kolektorom.

Standardna biblioteka `std_logic_1164` definira logički tip sa 8 vrijednosti koji je pogodan za razrješavanje. Pojedine vrijednosti tipa `std_logic` koji je tom bibliotekom definiran su navedene u tablici 3.4. Biblioteka nad tim tipom definira standardne logičke operatore, korisničke funkcije, te funkciju razrješavanja.

’0’	“jaka” logi čka nula
’1’	“jaka” logi čka jedinica
’X’	“jaka” nepoznata vrijednost
’L’	“slaba” logi čka nula
’H’	“slaba” logi čka jedinica
’W’	“slaba” nepoznata vrijednost
’Z’	visoka impedancija
’-’	nebitna vrijednost (don’t care)
’U’	neinicijalizirana vrijednost

Tablica 3.4: Vrijednosti predviđene tipom `std_logic`.

Prije korištenja bilo kojeg objekta iz biblioteke, potrebno je na samom početku datoteke uključiti odgovarajući paket:

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

Funkcija razrješavanja nad tipom `std_logic` se temelji na sljedećim pravilima:

- ako oba signala postavljaju istu logičku vrijednost, rezultat razrješavanja je jednak njoj;
- ako jedan od spojenih signala postavlja “slabu”, a drugi “jaku” logičku vrijednost, rezultat razrješavanja odgovara “jakom” signalu;
- ako oba spojena signala postavljaju logičku vrijednost iste snage (“jaka” ili “slaba”), rezultat razrješavanja je nepoznata vrijednost `'X'` ili `'W'`;
- ako jedan od spojenih signala postavlja vrijednost “Z”, a drugi “jaku” ili “slabu” logičku vrijednost, rezultat razrješavanja odgovara drugom signalu;
- nebitna vrijednost `'-'` u kombinaciji sa bilo kojom drugom vrijednošću osim `'U'` se razrješava kao `'X'`;
- neinicijalizirana vrijednost `'U'` u kombinaciji sa bilo kojom drugom vrijednošću se razrješava kao `'U'`;

Deklaracija funkcije razrješavanja u standardnoj biblioteci VHDL-a je dana ispisom 3.22.

```
constant resolution_table : stdlogic_table := (
-- ---------------------------------------------------
-- |  U    X    0    1    Z    W    L    H    -      |   |
-- ---------------------------------------------------
   ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- | U |
   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- | X |
   ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ), -- | 0 |
   ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ), -- | 1 |
   ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ), -- | Z |
   ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ), -- | W |
   ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ), -- | L |
   ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ), -- | H |
   ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )  -- | - |
```

```
);
```

Ispis 3.22: Deklaracija funkcije razrješavanja nad tipom std_logic.

Poglavlje 4

Strukturno modeliranje

Za razliku od čistog ponašajnog modela, koji sustav opisuje kao crnu kutiju, strukturni model raščlanjuje sustav na jednostavnije podsustave. Svaki od tih podsustava je u takvom hijerarhijskom pristupu oblikovanju potrebno opisati zasebnim modelom, koji ponovo može biti ponašajni ili strukturni.

4.1 Izravno instanciranje komponenti

Svaki modelirani podsustav se može upotrijebiti kao građevni element složenijeg ili hijerarhijski višeg sustava, upotrebom instanciranja komponente kao posebne usporedne naredbe. Instanciranje komponente je zapravo ugrađivanje jednog primjera (instance) zadane komponente u složeni sustav. Pri tome se mora naznačiti koje signale sustava treba povezati sa vanjskim signalima (portovima) komponente.

Neka su modeli dvoulaznih "i" vrata i "D" bistabila zadani ispisima 4.1 i 4.2.

```
entity vrata_i is
  port(
    a,b: bit;
    z: out bit);
end entity vrata_i;

architecture ponasajna
   of vrata_i is
begin
  z <=
    a and b
    after 2 ns;
end architecture ponasajna;
```

Ispis 4.1: Ponašajni model "i" vrata.

```
entity bb_d is
  port(
    d,clk: bit;
    q: out bit);
end entity bb_d;
architecture ponasajna of bb_d is
begin
  process(clk) begin
    if clk'event and clk='1' then
      q <= d after 2 ns;
    end if;
  end process;
end architecture;
```

Ispis 4.2: Ponašajni model "D" bistabila.

Na temelju postojećih komponenti, može se modelirati D bistabil sa ulazom za omogućavanje:

```
entity bb_de is
  port(
    d,clk,e: bit;
```

```
    q: out bit);
end entity bb_de;

architecture strukturna of bb_de is
  signal clke: bit;
begin
  vrata: entity work.vrata_i
    port map( clk, e, clke);
  bb: entity work.bb_d(ponasajna)
    port map(
      d => d,
      clk => clke,
      q => q
    );
end architecture;
```

Ispis 4.3: Strukturni model “D” bistabila s ulazom za omogućavanje.

Formalna sintaksa naredbe instanciranja predviđa mogućnost da se u složenom sustavu ne iskoriste svi formalni argumenti komponente. Na mjestima takvih argumenata je potrebno naznačiti ključnu riječ `open`. Ime arhitekture se ne mora navesti ako instancirana komponenta ima samo jednu izvedbu.

```
instanciranje_komponente ⇐
  labela:
    entity ime [ (ime_arhitekture) ]
    [ generic map ( lista_parametara ) ]
    port map ( lista_veza ) ] ;

lista_veza ⇐
  ( [ formalna_veza => ] ( signal | izraz | open ) )
  {, ... }
```

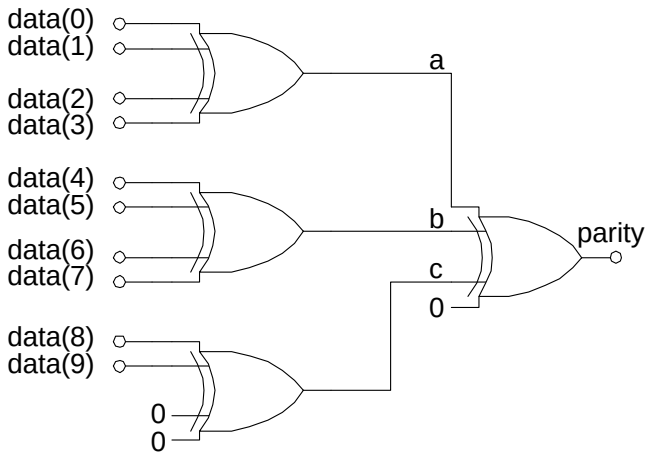
Ispis 4.4: Skica sintakse usporedne naredbe instanciranja komponente.

4.2 Prospajanje dijelova sabirnica

Ponekad se javlja potreba za prospajanjem samo nekih linija sabirnice složenog sustava na samo neke vanjske signale komponente.

Neka je potrebno modelirati brzi desetbitni sklop za provjeru pariteta korištenjem četiriju četverobitnih vrata “ekskluzivno ili”. Takva bi se potreba mogla javiti ako bismo sklop htjeli sintetizirati pomoću nekog FPGA sklopa koji podržava ekstremno brza četveroulazna vrata.

Idejna shema sklopa je predstavljena slikom 4.1, a neka su sučelja četveroulaznih vrata i konačnog sklopa dana ispisima 4.5 odnosno 4.6.



Slika 4.1: Desetoulazni sklop za provjeru pariteta.

```
entity xor4 is
  port(
    din: bit_vector(0 to 3);
    z: out bit
  );
end entity xor4;
```

Ispis 4.5: Sučelje vrata “ekskluzivno ili” sa 4 ulaza.

```
entity xor10 is
  port(
    data: bit_vector(0 to 9);
    parity: out bit
  );
end entity xor10;
```

Ispis 4.6: Sučelje vrata “ekskluzivno ili” sa 10 ulaza.

Vidi se da se prospajanje ne može obaviti jednostavnim navođenjem stvarnih argumenata za četveroulazna vrata. Pojedine linije sabirnica je zato potrebno presložiti odgovarajućim prilagodnim izrazima, kao što je prikazano ispisom 4.7.

```
architecture strukturna of xor10 is
  signal a,b,c: bit;
begin
  x5a: entity work.xor4 port map(
    din(0 to 3) => data(0 to 3),
    z           => a);
  x5b: entity work.xor4 port map(
    din(0 to 3) => data(4 to 7),
    z           => b);
  x5c: entity work.xor4 port map(
    din(0 to 1) => data(8 to 9),
    din(2 to 3) => "00",
    z           => c);
  x5d: entity work.xor4 port map(
    din(0) => a,
    din(1) => b,
    din(2) => c,
    din(3) => '0',
```

```
    z      => parity);
end architecture strukturna;
```

Ispis 4.7: Strukturna izvedba vrata “ekskluzivno ili” sa 10 ulaza.

U stvarnoj situaciji, proizvođač FPGA sklopa bi zajedno sa alatima za sintezu isporučio i model izravno podržanih četveroulaznih vrata. Taj model bi mogao izgledati kao što je prikazano ispisom 4.8.

```
architecture ponasajna of xor4 is
begin
  process(din) is
    variable rv: integer;
  begin
    rv:=0;
    for i in din'range loop
      rv := rv + bit'pos(din(i));
    end loop;
    z <= bit'val(rv mod 2);
  end process;
end architecture ponasajna;
```

Ispis 4.8: Ponašajna izvedba vrata “ekskluzivno ili” sa 4 ulaza.

4.3 Modeliranje ispitnog okruženja

Za model se kaže da je dovršen tek nakon što uspješno prođe testiranje u ispitnom okruženju (*engl.* testbench). Funkcionalnost modela se ispituje tako da mu se na ulazne vanjske signale postavlja niz vijednosti (ispitnih vektora), te provjerava da li stanje na izlazu odogovara očekivanjima.

VHDL model ispitnog okruženja se stoga sastoji od slijedećih komponenti:

1. sučelja bez vanjskih signala (ispitno okruženje je gotovo uvijek na hijerarhijski najvišoj razini);
2. instance komponente koja se testira;
3. slijedni generator ispitnih vektora koji ujedno i uspoređuje izlazne vrijednosti sa očekivanima.

```
-- prazno sučelje (točka 1)
entity model_t is
end entity model_t;

architecture test of model_t is
  -- deklaracija signala za ispitne vektore
  signal A ...
begin
  -- instanca komponente koja se testira (točka 2.)
  m: entity work.model(model_arh)
```

```
    port map ( A, ...);

  -- generator ispitnih vektora (točka 3.)
  vektori: process begin
    A <= ...
    wait for xx ns;
    assert(...);

    A <= ...
    ...

    wait;
  end process vektori
end architecture test
```

Ispis 4.9: Skica izvedbe ispitnog okruženja.

Ispitna okruženja su iznimno važna kod složenih modela koje je jako teško ispitati nesistematičnim metodama. VHDL omogućava pristup datotekama, pa će se kod takvih modela ispitni vektori dohvaćati s diska, što omogućava njihovo automatsko generiranje nezavisnom aplikacijom.

4.3.1 Naredba `assert`

Uspoređivanje dobivenih rezultata sa očekivanima se može pojednostavniti korištenjem slijedne naredbe `assert`, čija semantika je slična kao i u jeziku `C`. Iako se najčešće javlja upravo u ispitnim komponentama, može se koristiti bilo gdje u modelu. Naredba `assert` je jedan od ključnih mehanizama za rano otkrivanje grešaka u složenim modelima. Skica sintakse naredbe `assert` je prikazana ispisom 4.10.

```
naredba_assert ⇐
  assert logički_uvjet
    [ report opis_pogreške ]
    [ razina_strogosti ] ;

type razina_strogosti is}
  (note, warning, error, failure);
```

Ispis 4.10: Skica sintakse naredbe `assert`.

Ako logički uvjet naredbe nije zadovoljen, dijagnostička poruka `opis` se ispisuje na konzoli, a izvođenje simulacije se prekida ako je `razina` postavljena na `error` ili `failure`. Podrazumijevane vrijednosti za `opis_pogreške` i `razina_strogosti` su `Assertion violation` odnosno `error`.

Sljedeća assert naredba bi se mogla nalaziti na početku procesa koji opisuje SR bistabil:

```
assert not (s='1' and r='1');
```

4.3.2 Ispitno okruženje za sklop za provjeru pariteta

Važnost ispitnih okruženja u programiranju se teško može precijeniti, pa je na kraju njihovog razmatranja ponuđen kompletan primjer ispitne komponente u VHDL-u. Ispis 4.11 prikazuje ispitno okruženje za desetoulazni sklop "isključivo ili", čije sučelje je prikazano ispisom 4.6.

```
use std.textio.all;

 entity xor10_t is
 end entity xor10_t ;

 architecture test of xor10_t is
   signal a: bit_vector(0 to 9);
   signal z: bit;

   procedure printStatus is
     variable l:line;
   begin
     write(l, now, right, 10, ns);
     write(l, string'("␣in:␣a="));
     write(l, a);
     write(l, string'(",␣out:␣"));
     write(l, z);
     writeline(output, l);
   end procedure printStatus;

 begin
   t: entity work.xor10 port map(a,z);
   process begin
     a <= "1010101010";
     wait for 10 ns;
     printStatus;

     a <= "1110101010";
     wait for 10 ns;
     printStatus;

     a <= "1111101010";
     wait for 10 ns;
     printStatus;

     wait;
   end process;

 end architecture test;
```

Ispis 4.11: Ispitno okruženje za sklop čije sučelje je dano ispisom 4.6.

Bibliografija

[Ashenden96] Peter J. Ashenden, *The designer's guide to VHDL*, Morgan-Kaufmann, San Francisco, California, 1996.

[Mirkowski98] Jaroslaw Mirkowski and Marcin Kapustka, Evita - enhanced vhdl tutorial with applications, 1998, URL `http://www.aldec.com/Downloads/`, free download.