Kratki uvod u jezik VHDL

Marko Čupić

3. siječnja 2016.

Sadržaj

Predgovor

Ova skripta služi kao popratni materijal laboratorijskim vježbama iz kolegija Digitalna logika. U skripti je dan pregled podskupa jezika VHDL koji se koristi u okviru laboratorijskih vježbi. Ovaj dokument nije konačna verzija skripte (materijal je još u izradi i postupno će se dopunjavati).

Pregledao: Marko Zec

Poglavlje 1

Uvod

Povećanjem složenosti digitalnog sklopovlja pojavila se potreba za omogućavanjem njegovog formalnog opisivanja kako bi se omogućilo obavljanje simulacije funkcionalnosti prije no što se sklopovlje pošalje u proizvodni proces. Ustanove li se pogreške prilikom simulacije, puno je jednostavnije i brže (i *jeftinije*) takvu pogrešku ispraviti.

Daljnjim razvojem tehnologije te razvojem programirljivih sklopova pojavila se još jedna zgodna primjena formalno opisanog sklopovlja: razvijeni su računalom potpomognuti alati koji na temelju formalnog opisa digitalnog sklopovlja mogu programirati programirljive sklopove kako bi se oni ponašali u skladu s tim formalnim opisom. Takve alate nazivamo sintetizatorima digitalnog sklopovlja koji na temelju formalnog opisa sklopa generiraju naputak (obavljaju *sintezu*) prema kojem se programira programirljivi sklop. Rezultat sinteze najčešće je JEDEC datoteka koja se koristi kao ulaz za program koji provodi sam postupak programiranja.

Za potrebe formalnog opisivanja digitalnog sklopovlja razvijeno je mnoštvo jezika za opis sklopovlja. Danas su u najširoj uporabi dva:

- Verilog te
- VHDL.

U okviru kolegija *Digitalna logika* koristit ćemo podskup jezika VHDL koji ćemo opisati u nastavku.

VHDL je skraćenica od *VHSIC Hardware Description Language*, gdje je prvi dio kratica od *Very High Speed Integrated Circuit*. Prevedemo li ovo na hrvatski, VHDL stoji za *Jezik za opis integriranih sklopova vrlo visokih brzina*, što nas, naravno, ne treba sprječavati da ga koristimo i za opisivanje jednostavnijih i sporijih sklopova. Ponekad se u šali kaže i da je VHDL zapravo kratica od *Very Hard and Difficult Language* – no kroz ovaj tekst pokušat ćemo vas uvjeriti da to nije tako.

Poglavlje 2

Načini opisivanja sklopova

U okviru ovog poglavlja upoznat ćemo se s osnovnim načinima opisivanja digitalnog sklopovlja jezikom VHDL. Ovisno što znamo o digitalnom sklopu, opisat ćemo ga ili definiranjem njegove funkcije ili definiranjem njegove građe koja se oslanja na uporabu jednostavnijih sklopova. Stoga ćemo razlikovati tri vrste opisa, odnosno tri modela sklopa.

Model toka podataka (engl. *dataflow model*) je model sklopa kod kojega pišemo naredbe koje kombiniraju ulazne signale (i eventualno pomoćne signale) uporabom naredbi dodjeljivanja vrijednosti signalima i tako određuju vrijednosti izlaza sklopa. Arhitektura (*dio opisa sklopa koji definira funkciju ili građu sklopa*) kod ovog modela sastoji se od jedne ili više naredbi pridruživanja vrijednosti signalu.

Strukturni model (engl. *structural model*) je model sklopa koji sklop opisuje pozivajući se na njegovu građu te način na koji su jednostavnije komponente od kojih je sklop izgrađen povezane s ulazima, povezane međusobno te povezane s izlazima modeliranog sklopa. Arhitektura kod ovog modela sastoji se od niza naredbi koje predstavljaju stvaranje primjeraka korištenih komponenata i njihovo međusobno povezivanje.

Funkcijski model (zovemo ga još i **ponašajni model**, engl. *behavioral model*) je model sklopa kod kojega funkcionalnost sklopa najčešće ne opisujemo oslanjajući se logičke operatore koji bi oslikavali način na koji se ulazi transformiraju u izlaze već pišemo algoritam koji nam ne govori kako bi sklop trebalo sintetizirati već samo navodi kako se ulazi preslikavaju u izlaze. Ovo je najbliže što možemo prići klasičnom modelu "crne kutije" kod kojega ni na koji način ne opisujemo građu sklopa već samo njegovu funkcionalnost, koristeći proceduralno programiranje, naredbe poput if, case, for i slično. Arhitektura kod ovog modela sastoji se od jednog ili više blokova process.

2.1 Opis jednog sklopa

Prilikom modeliranja digitalnog sklopa pisat ćemo odgovarajući opis jezikom VHDL. Iako to nije nužno, dobra je praksa opis svakog sklopa staviti u zasebnu datoteku. Prilikom rada sa sustavom VHDLLab2, sam će nas sustav tjerati da pišemo upravo takve opise: unutar jednog projekta, za svaki ćemo sklop morati napraviti novi model koji će biti pohranjen u vlastitu datoteku.

Sam opis sklopa jezikom VHDL sastojat će se od sljedećih dijelova.

1. *Deklaracija korištenih biblioteka i paketa.* Svaki opis započet će navođenjem biblioteka i paketa

iz tih biblioteka koje ćemo koristiti u opisu. Biblioteke se uključuju uporabom ključne riječi library (redak 1 u primjeru u nastavku) a pojedini paketi uporabom ključne riječi use (redak 2 u primjeru u nastavku).

Jedna biblioteka može se sastojati od više paketa, a svaki paket može nuditi definiciju tipova podataka, funkcija i sličnoga. Želimo li iz nekog paketa uključiti sve što je u njemu definirano (kako bismo sve mogli koristiti za opisivanje sklopova), u naredbi kojom se paket uključuje na kraju ćemo napisati .all.

Cjeloviti primjer prikazan je u nastavku. Uočimo da svaka od naredbi završava znakom točka-zarez.

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

2. *Deklaracija sučelja sklopa.* Modelirani sklop, gledan izvana, prema svojoj se okolini predstavlja kao crna kutija koja ima određen broj ulaza te određen broj izlaza. Kroz svaki od ulaza odnosno izlaza sklop prima/šalje signale određenog tipa. Definiranjem sučelja sklopa korisnik mora ponuditi detaljan opis svih ulaza i izlaza.

   Definicija sučelja započinje ključnom riječi ENTITY nakon čega slijedi naziv sklopa čije se sučelje modelira, ključne riječi IS PORT(, definicije ulaza i izlaza, potom); i konačno ključna riječ END, naziv modeliranog sklopa pa znak točka-zarez. Svaki od ulaza odnosno izlaza naziva se jednim *portom* sklopa. Definicija porta započinje navođenjem naziva porta, slijedi dvotočka, ključna riječ IN (ako je port ulaz) odnosno OUT (ako je port izlaz), naziv programskog tipa koji opisuje vrstu signala koji se prima/šalje kroz taj port te na kraju znak točka-zarez (ali samo ako to nije definicija posljednjeg porta; ako je, znak točka-zarez ne smije se navesti).

   Pretpostavimo da modeliramo sučelje sklopa koji smo nazvali *SklopS1* i koji ima tri ulaza (A, B i C) te dva izlaza (f i g). Definicija sučelja takvog sklopa bila bi sljedeća.

```
1 ENTITY sklopS1 IS PORT (
2   a: IN std_logic;
3   b: IN std_logic;
4   c: IN std_logic;
5   f: OUT std_logic;
6   g: OUT std_logic
7 );
8 END sklopS1;
```

   Kao tip podataka ovdje smo koristili tip std_logic koji je definiran u biblioteci IEEE. Definicije portova koji su istog smjera (ulaz/izlaz) i kroz koje putuju signali istog tipa mogu se navesti kraće, tako da se nazivi portova nanizaju uz razdvajanje znakom zareza. Tako je, primjerice, ekvivalentna definicija sučelja prethodno navedenoj prikazana u nastavku.

```
1 ENTITY sklopS1 IS PORT (
2   a, b, c: IN std_logic;
3   f, g: OUT std_logic
4 );
5 END sklopS1;
```

   Uočite kako u oba slučaja, nakon posljednje definicije porta nema znaka točka-zarez.

3. *Deklaracija arhitekture sklopa.* U ovom djelu se daje opis unutrašnjosti "crne kutije". Kako smo prethodno napomenuli, taj opis može biti ostvaren na više načina. U određenim slučajevima, u opisu ćemo kombinirati elemente različitih načina opisivanja – takav ćemo opis onda zvati *hibridni*.

   Deklaracija arhitekture sklopa započinje navođenjem ključne riječi ARCHITECTURE, nakon čega slijedi naziv arhitekture, ključna riječ OF, naziv sklopa čija je to arhitektura te ključna riječ IS. Slijedi dio u kojem po potrebi možemo definirati pomoćne (unutarnje odnosno interne) signale,

konstante i nove tipove podataka. Nakon toga dolazi ključna riječ BEGIN, jedna ili više naredbi koje čine model toka podataka, strukturni model ili ponašajni model (svaka naredba završava s točkom-zarezom), i konačno dolazi ključna riječ END, naziv arhitekture i točka-zarez.

Primjer jedne arhitekture dan je u nastavku.

```
1 ARCHITECTURE arch1 OF sklopS1 IS
2   -- ovdje bi došle deklaracije pomoćnih signala
3 BEGIN
4   -- ovdje dolaze naredbe ponašajnog
5   -- ili strukturnog opisa
6   f <= a AND b AFTER 10 ns;
7   g <= (a OR c) AND b AFTER 10 ns;
8 END arch1;
```

U prethodnom primjeru sve što je navedeno iza -- čini komentar. Naziv arhitekture može biti bilo kakav valjan identifikator. Treba napomenuti da opći oblik opisa jednog sklopa može imati još neke elemente, ali na ovom mjestu ih nećemo navoditi.

Operator <= (retci 6 i 7 u prethodnom primjeru) je operator kojim se vrijednost izraza navedenog s desne strane pridružuje izlazu (odnosno općenito nekom signalu) navedenom s lijeve strane. Ako prije znaka točka-zarez dođe ključna riječ AFTER, tada se definira da se to pridruživanje događa uz kašnjenje koje je navedeno nakon ključne riječi, čime imamo mogućnost modeliranja kašnjenja u digitalnim sklopovima. Vrijeme koje se navodi ima iznos i mjernu jedinicu. Primjerice, u prethodnom primjeru oba izlaza za svojim ulazima kasne po 10 nanosekundi.

2.2 O jeziku VHDL

Jezik VHDL nije osjetljiv na velika i mala slova. Tako, primjerice, ključnu riječ kojom započinjemo arhitekturu sklopa možemo pisati ARCHITECTURE, architecture, ArChItEcTuRe ili na bilo koji drugi način.

Prilikom rada sa sustavom VHDLLab2 treba se ipak držati jednog ograničenja koje nameće taj sustav: naziv sklopa (tj. naziv modela) mora se uvijek pisati na isti način. Ako smo napravili model sklopS1, onda ga na svim mjestima moramo navoditi upravo tako.

Od logičkih operatora, na raspolaganju nam stoje sljedeći operatori.

not	logičko NE
and	logičko I
or	logičko ILI
nand	logičko NI
nor	logičko NILI
xor	logičko isključivo-ILI
xnor	komplement logičkog isključivo-ILI

Operator NOT višeg je prioriteta od ostalih operatora. Svi preostali binarni operatori navedeni u prethodnoj tablici istog su prioriteta. Posljedica toga je da prilikom pisanja logičkih izraza *moramo* koristiti zagrade za definiranje prioriteta operatora jer ćemo u slučaju da ih ne koristimo a u izrazu imamo više različitih operatora dobiti prijavljenu pogrešku prilikom prevođenja modela. Tako je, primjerice, sljedeći isječak kôda kojim pokušavamo definirati da je $f = \bar{a} \cdot b + c$ pogrešan.

```
1   f <= NOT a AND b OR c;
```

Prilikom prevođenja ovog modela, sustav će nas upozoriti da ne zna jesmo li htjeli najprije izračunati logičko I između $\bar{a}$ i b pa s tim rezultatom napraviti logičko ILI s c, ili je najprije trebalo izračunati logičko ILI između b i c pa s tim rezultatom napraviti logičko I s $\bar{a}$. Uz pretpostavku da smo htjeli da se izraz protumači na nama uobičajen način (I je većeg prioriteta od ILI), stavili bismo zagrade kako je prikazano u nastavku.

```
1  f <= (NOT a AND b) OR c;
```

2.3 Tip podataka std_logic

Tip std_logic definira vrijednosti signala koje jedan digitalni sklop može slati drugom digitalnom sklopu. Razmislite li, brzo ćete doći do zaključka da smo i mi na kolegiju osim logičke nule i logičke jedinice već spominjali i neke druge vrijednosti. Tako smo, primjerice, spomenuli da izlaz nekog sklopa može biti u stanju visoke impedancije (Z). Spomenuli smo i da nas ponekad uopće nije briga u kojem je stanju izlaz sklopa (*don't care*).

Prisjetite se sada cjeline s predavanja na kojoj smo govorili o implementacijama logičkih sklopova. Na izlazu logičkog sklopa visoka naponska razina može biti ostvarena na dva načina: jedan je ostvaren tako da je izlaz *pull-up*-otpornikom pritegnut prema napajanju (a tranzistor koji izlaz priteže prema masi je blokiran), a drugi je ostvaren tako da od izlaza prema napajanju imamo uključen tranzistor a od izlaza prema masi isključen tranzistor (*pasivno* vs. *aktivno* generiranje). Koja je posljedica ovih dvaju načina? Izlaze dva sklopa koja imaju izlaz pritegnut na napajanje preko *pull-up* otpornika možemo spojiti zajedno, i time ćemo dobiti novu Booleovu funkciju. To isto s izlazima koji imaju aktivno generiranje napona ne smijemo napraviti jer ćemo uništiti sklopove.

Kako bi omogućio opisivanje svih takvih vrsta sklopova, tip std_logic definira čak 9 različitih vrijednosti signala. Vrijednosti su pobrojane u nastavku.

1. `1`: predstavlja aktivno generiranu logičku jedinicu (engl. *forcing 1*). Ako se na izlazu sklopa pojavi ovakav signal, možemo zamisliti da sklop ima izlaz tranzistorom pritegnut na napajanje.

2. `0`: predstavlja aktivno generiranu logičku nulu (engl. *forcing 0*). Ako se na izlazu sklopa pojavi ovakav signal, možemo zamisliti da sklop ima izlaz tranzistorom pritegnut na masu.

3. `H`: predstavlja pasivno generiranu logičku jedinicu (engl. *weak 1*). Ako se na izlazu sklopa pojavi ovakav signal, možemo zamisliti da sklop ima izlaz pritegnut na napajanje preko *pull-up* otpornika.

4. `L`: predstavlja pasivno generiranu logičku nulu (engl. *weak 0*). Ako se na izlazu sklopa pojavi ovakav signal, možemo zamisliti da sklop ima izlaz pritegnut na masu *pull-down* otpornikom.

5. `X`: obično predstavlja stanje koje se dobije ako se dva izlaza koja aktivno generiraju izlaz spoje zajedno a sklopovi generiraju komplementarne izlaze (u praksi, označava nastanak kratkog spoja od napajanja prema masi). Naziv ove vrijednosti je *forcing Unknown*.

6. `W`: obično predstavlja stanje koje se dobije ako se dva izlaza koja pasivno generiraju izlaz spoje zajedno a sklopovi generiraju komplementarne izlaze (u praksi, označava situacije kada je kroz izlaze sklopova otvoren put od napajanje prema masi, ali koje obično nemaju fatalne posljedice jer je struja ograničena sumom iznosa *pull-up* i *pull-down* otpornika). Napon koji se dobije bit će vjerojatno negdje u zabranjenom području pa sljedeći sklop neće dobro protumačiti što mu se šalje. Naziv ove vrijednosti je *weak Unknown*.

7. `Z`: predstavlja izlaz koji je i od napajanja i od mase izoliran velikim otporom; to je izlaz koji je u trećem stanju (stanju visoke impedancije).

8. `-`: predstavlja situaciju u kojoj nas nije briga što je na izlazu (engl. *don't care*).

9. `U`: predstavlja situaciju u kojoj simulator ne zna što je na izlazu (engl. *uninitialized*). Primjerice, ako modeliramo sklop ILI tako da ima kašnjenje od 10 nanosekundi i na početku simulacije (od t=0) na ulaze dovedemo vrijednosti koje odgovaraju logičkoj nuli, što će biti na izlazu tog sklopa? Kako god da radimo simulaciju, simulator će temeljem našeg opisa moći zaključiti da će od 10. nanosekunde na izlazu biti vrijednost nula; međutim, što će biti od trenutka $t = 0ns$ do $t = 10ns$? Na to ne možemo odgovoriti, pa ćemo takve situacije označavati s vrijednosti `U`.

Konstante tipa std_logic pišu se pod jednostrukim navodnicima. Primjerice:

```
1  f <= '1';
```

Uz tip std_logic koji predstavlja jedan signal, biblioteka IEEE nudi nam i tip std_logic_vector koji se može koristiti za opisivanje više signala istog imena i tipa. Sjetimo se primjera multipleksora 8/1: to je sklop koji ima 8 podatkovnih ulaza koje obično označavamo slovom D i indeksima od 0 do 7, ima 3 adresna ulaza koje obično označavamo slovom A i indeksima od 2 do 0 te ima jedan izlaz (obično y). Uporabom tipa std_logic_vector u VHDL-u možemo napisati upravo takvu definiciju. Primjerice, sučelje ovakvog multipleksora definirali bismo na sljedeći način.

```
1  ENTITY mux81 IS PORT (
2    d: IN std_logic_vector(0 to 7);
3    a: IN std_logic_vector(2 downto 0);
4    y: OUT std_logic
5  );
6  END mux81;
```

Uočite kako se nakon naziva tipa std_logic_vector u oblim zagradama navodi raspon indeksa; ako je početni broj manji od završnog, raspon se piše s TO a ako je početni broj veći od završnog, dolazi ključna riječ DOWNTO. Pojedinim signalima može se pristupati indeksiranjem; primjerice možemo zatražiti vrijednost d(6) što je predzadnji od signala ulaza d. Možemo uzimati i podraspone; primjerice: d(1 to 4) predstavlja signale d(1), d(2), d(3) i d(4), i to je po tipu opet std_logic_vector.

Konstante koje su tipa std_logic_vector navode se pod dvostrukim navodnicima. Primjerice, ako je izlaz f definiran kao std_logic_vector(0 to 3), primjer naredbe kojom ćemo vrijednosti dodijeliti svim četirima signalima je:

```
1  f <= "001U";
```

što je ekvivalentno kao da smo dali četiri naredbe:

```
1  f(0) <= '0';
2  f(1) <= '0';
3  f(2) <= '1';
4  f(3) <= 'U';
```

2.3.1 Booleove operacije nad tročlanim skupom {U, 0, 1}

Osim djelovanja Booleovih operatora nad dvočlanim skupom {0, 1}, za potrebe laboratorijskih vježbi moramo se upoznati s djelovanjem tih operatora nad nešto širim skupom. Biblioteka IEEE u okviru paketa std_logic_1164 definira djelovanje Booleovih operatora nad svih devet vrijednosti signala std_logic. Mi ćemo se upoznati s minimalnim proširenjem kod kojeg ćemo dvočlani skup {0, 1} proširiti na tročlani skup {0, 1, U} uključivanjem još i vrijednosti U.

Nad tako definiranim tročlanim skupom, vrijednost `U` će i dalje označavati situaciju u kojoj ne znamo koja je vrijednost signala (ali ako su jedine druge moguće vrijednosti `0` i `1`, znamo da signal mora biti nešto od toga – samo ne znamo što).

Imajući to u vidu, proširenje djelovanja operatora je trivijalno, i dano je u nastavku.

Logičko I

- `0` I `0` = `0`
- `0` I `1` = `0`
- `0` I `U` = `0`
- `1` I `0` = `0`
- `1` I `1` = `1`
- `1` I `U` = `U`
- `U` I `0` = `0`
- `U` I `1` = `U`
- `U` I `U` = `U`

Logičko ILI

- `0` ILI `0` = `0`
- `0` ILI `1` = `1`
- `0` ILI `U` = `U`
- `1` ILI `0` = `1`
- `1` ILI `1` = `1`
- `1` ILI `U` = `1`
- `U` ILI `0` = `U`
- `U` ILI `1` = `1`
- `U` ILI `U` = `U`

Logičko NE

- NE `0` = `1`
- NE `1` = `0`
- NE `U` = `U`

Pogledajmo zašto su rezultati uz vrijednost `U` takvi kakvi jesu. Prema aksiomima Booleove algebre, $0 \cdot A = 0$; stoga, ako je A bilo nula, bilo jedan, rezultat ne ovisi o njemu i uvijek je nula. Stoga je i $0 \cdot U = 0$ jer nam `U` samo govori da ne znamo je li konkretna vrijednost nula ili jedan.

Oslanjajući se na jednako razmišljanje, dolazimo i do toga da je $1 \cdot U = U$; naime, ako je signal (čiju vrijednost ne znamo) zapravo u logičkoj jedinici, tada bi rezultat logičkog I bio 1; ako je pak u logičkoj nuli, rezultat bi bio nula. Kako nismo sigurni u rezultat logičke operacije (odnosno rezultat ovisi o konkretnoj vrijednosti tog signala), moramo i za sam rezultat reći da njegovu vrijednost ne znamo.

Tragom jednakog razmišljanja slijedi i da je $U \cdot U = U$. Preostale situacije nećemo razmatrati jer su ili podudarne s dvočlanom Booleovom algebrom, ili su rješive oslanjajući se na komutativnost operatora I.

Pogledajmo u nastavku dva primjera.

Primjer 1.
Što će biti dodijeljeno u signal f nakon izvođenja naredbe pridruživanja f <= (a and b) or c; ako je a='0', b='U', c='0'?

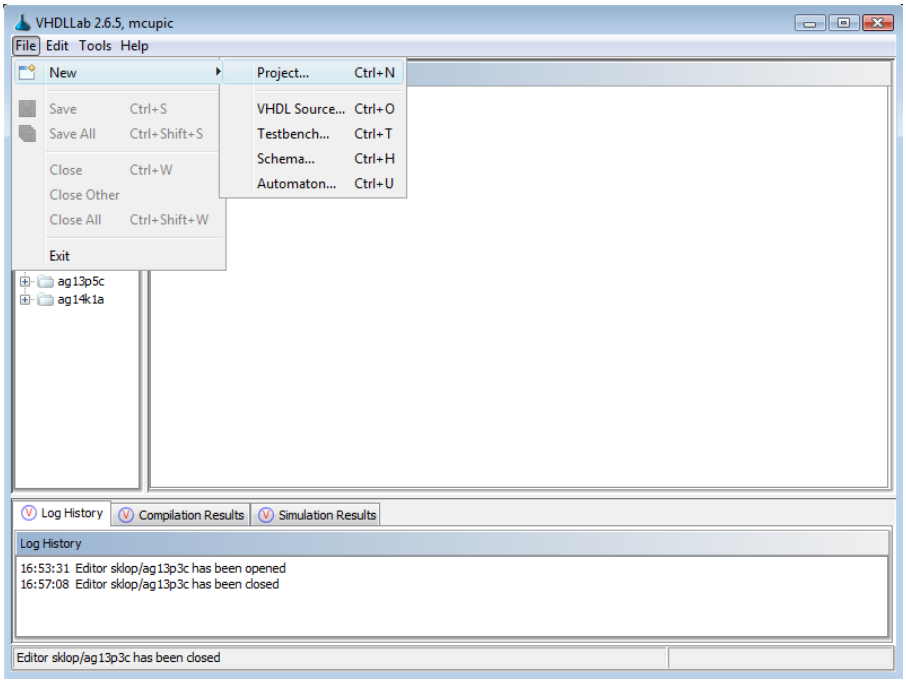
Rješenje:
Prema prethodno definiranom djelovanju operatora, najprije računamo a and b = '0' and 'U' = '0'. Potom računamo '0' or '0' = '0'. Signalu f bit će pridjeljena vrijednost '0'.

Primjer 2.
Što će biti dodijeljeno u signal f nakon izvođenja naredbe pridruživanja f <= a and not a; ako je a='U'?

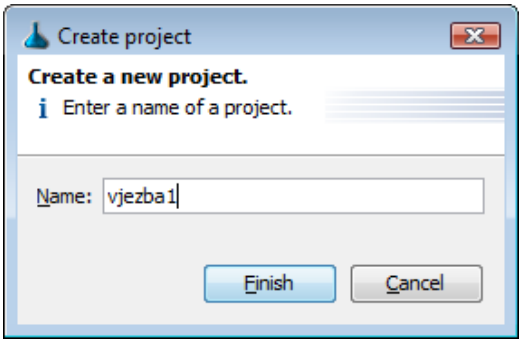
Rješenje:
Prema prethodno definiranom djelovanju operatora, najprije računamo not a = not 'U' = 'U'. Potom računamo 'U' and 'U' = 'U'. Signalu f bit će pridjeljena vrijednost 'U'. Uočite da u ovako definiranom tročlanom skupu ne vrijedi da je $A \cdot \bar{A} = 0$ ili da je $A + \bar{A} = 1$.

2.4 Modeliranje jednostavnog sklopa

U nastavku ćemo proći kroz modeliranje jednostavnog sklopa i to uporabom modela toka podataka, potom strukturnog modela i konačno ponašajnog modela. Sve primjere isprobajte i u sustavu VHDLLab2. Kao prvi korak, u sustavu VHDLLab2 napravite novi projekt (`vjezba1`): iz izbornika `File` odaberite `New` pa `Project`.



U dijalogu koji se otvori unesite kao ime projekta `vjezba1` i pritisnite `Finish`.



S lijeve strane u popisu projekata pojavit će se novostvoreni projekt. Sve što je opisano u nastavku ovog poglavlja radite u tom projektu.

2.4.1 Opis sklopa modelom toka podataka

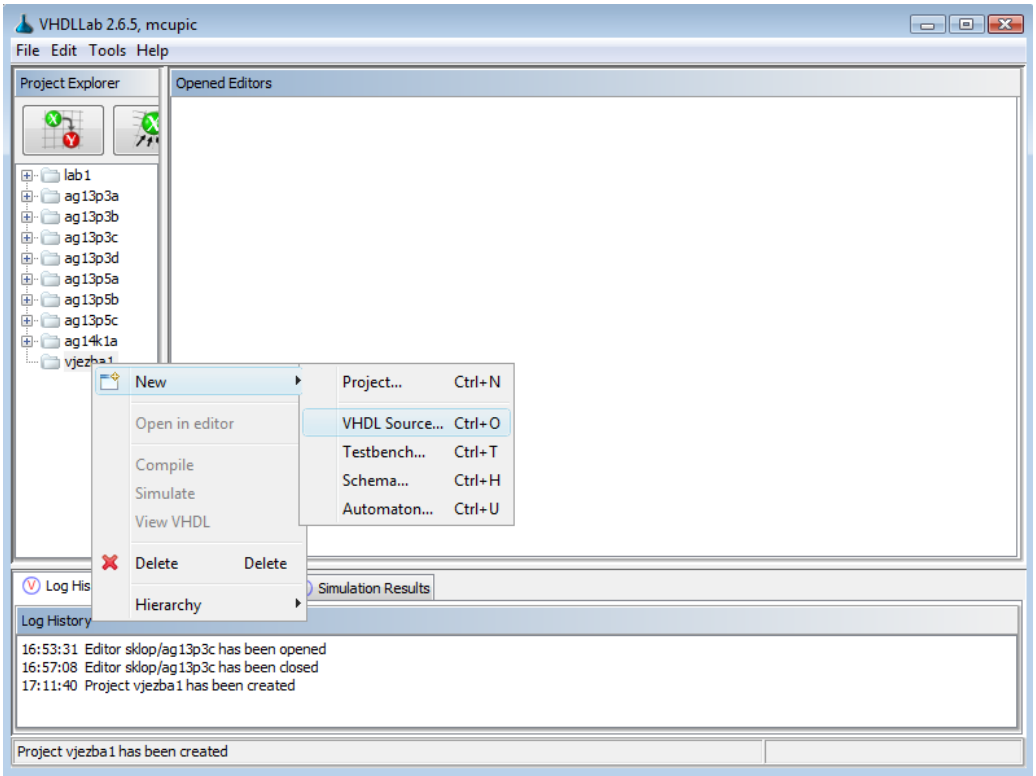
Pretpostavimo da je potrebno napisati model sklopa koji ima četiri ulaza a, b, c i d te jedan izlaz f, i koji ostvaruje sljedeću Booleovu funkciju: $f = (a \cdot b + c) \cdot d$. Također, neka sklop kao "crna kutija" ima kašnjenje od 10 nanosekundi.

Model toka podataka takvog sklopa dan je u nastavku. Sklop smo nazvali `sklop1`. Model se sastoji od jedne naredbe pridruživanja vrijednosti signalu.

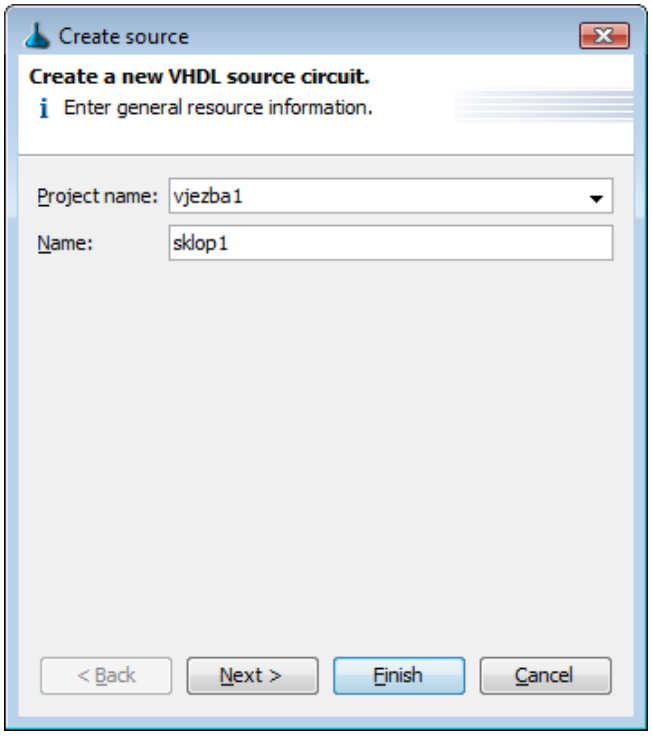
Listing 2.1 : Opis sklopa modelom toka podataka

```
-- uključivanje korištenih biblioteka
library IEEE;
use IEEE.std_logic_1164.all;

-- definicija sučelja sklopa
ENTITY sklop1 IS PORT (
   a: IN std_logic;
   b: IN std_logic;
   c: IN std_logic;
   d: IN std_logic;
   f: out std_logic);
END sklop1;

-- definicija arhitekture sklopa
-- dan je funkcijski model
ARCHITECTURE arch1 OF sklop1 IS
BEGIN
   f <= ((a AND b) OR c) AND d AFTER 10 ns;
END arch1;
```

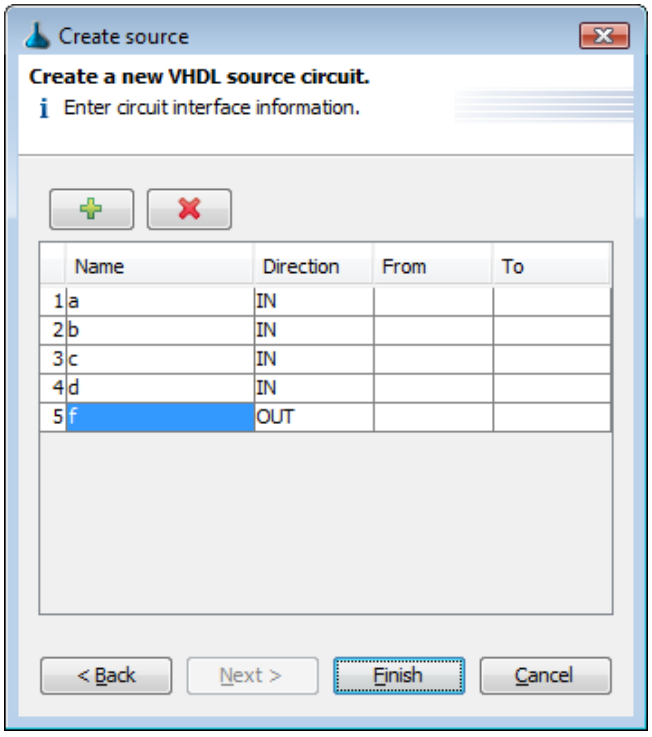
U projektu koji ste pripremili otiđite na `New` pa `VHDL source`.



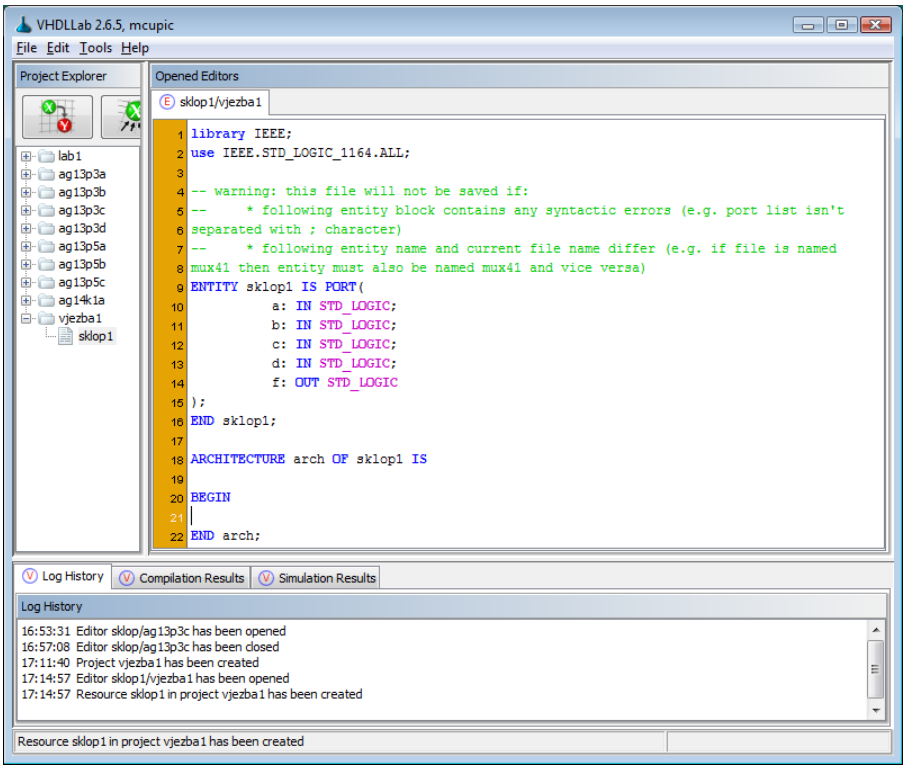
U dijalogu koji će se otvoriti unesite naziv sklopa (`sklop1`) i pritisnite `Next`.



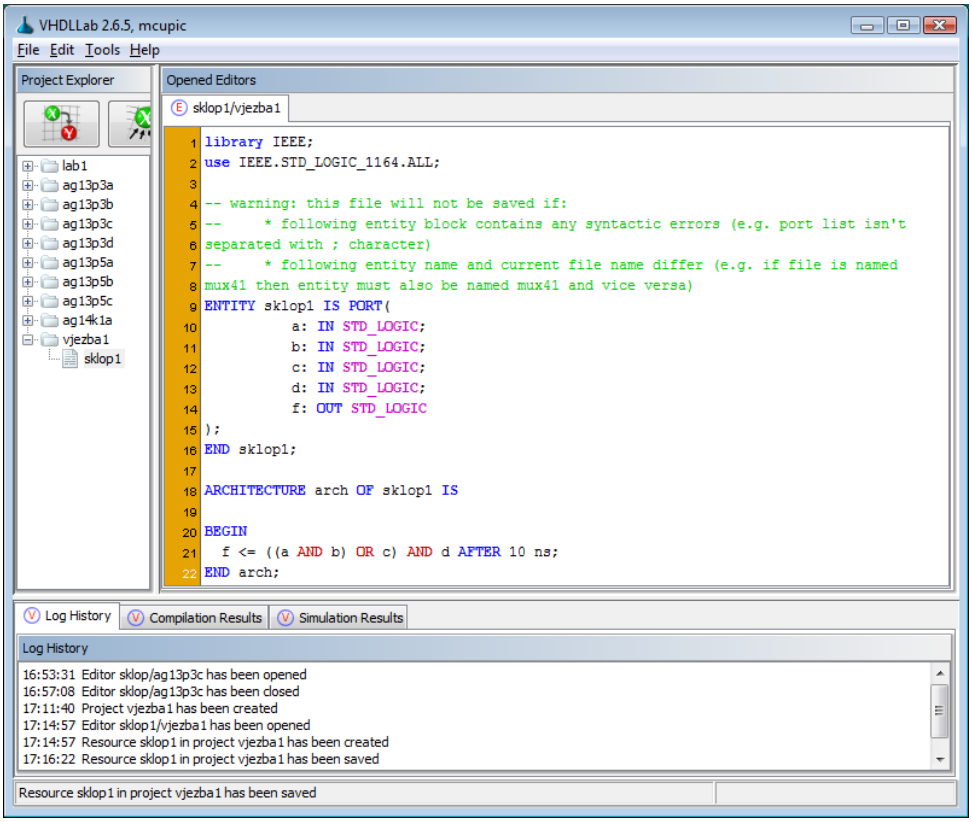
Na sljedećoj stranici dijaloga definirajte sučelje (dodajte četiri ulaza i jedan izlaz pritiskom na gumb `+` i podesite imena). Kad ste gotovi, pritisnite gumb `Finish`.



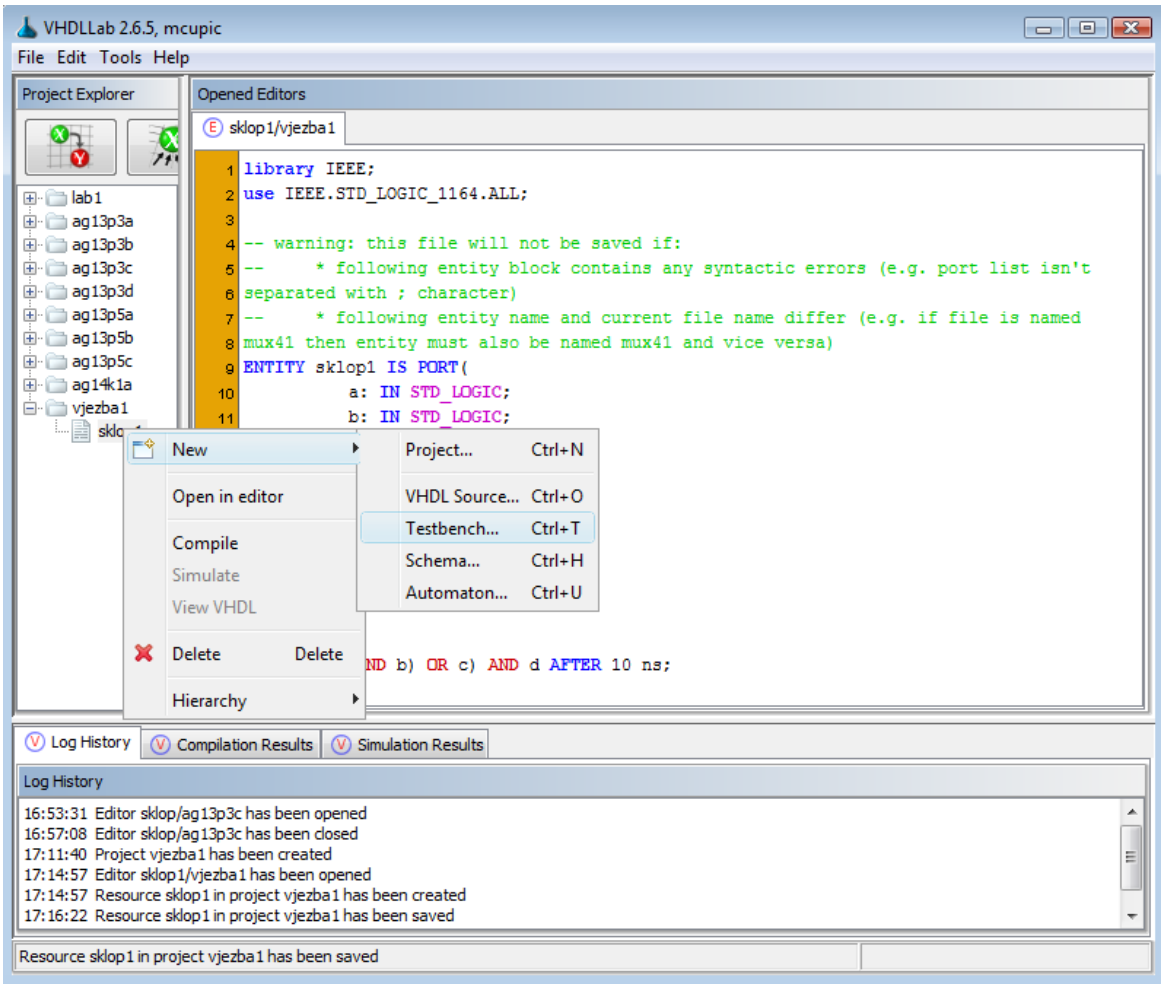
Otvorit će se uređivač VHDL modela, kako je prikazano na sljedećoj slici.



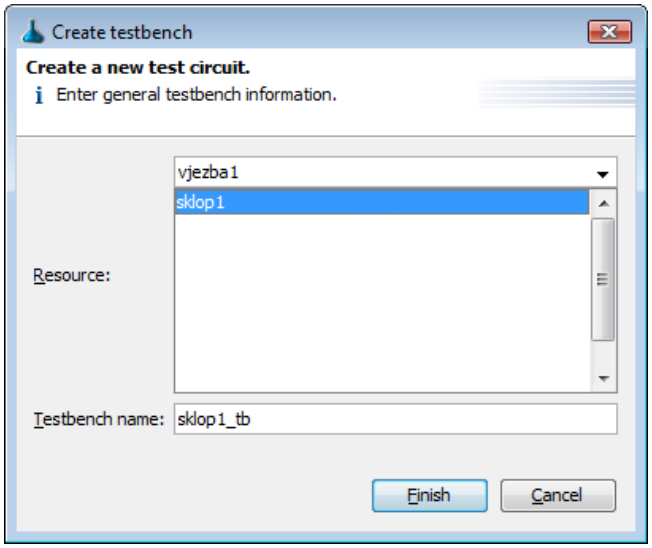
Otiđite u arhitekturu sklopa i tamo definirajte izraz prema kojem se računa vrijednost izlaza f uz zadano kašnjenje od 10 nanosekundi.



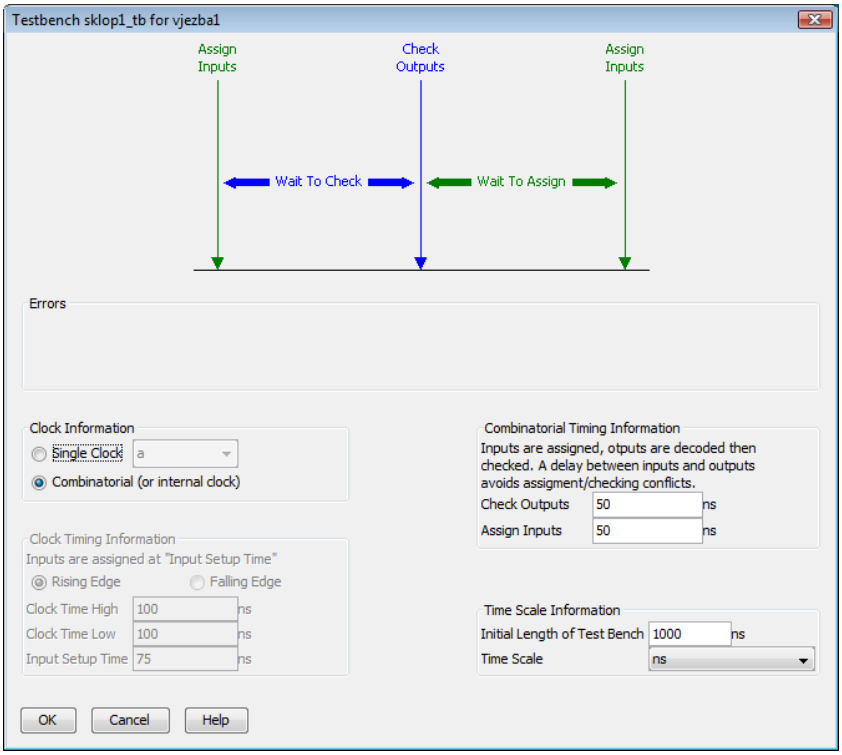
Jednom kada smo napravili model sklopa, isti bi trebalo i ispitati. Stoga u istom projektu napravite novi ispitni sklop: `New`, `Testbench`.



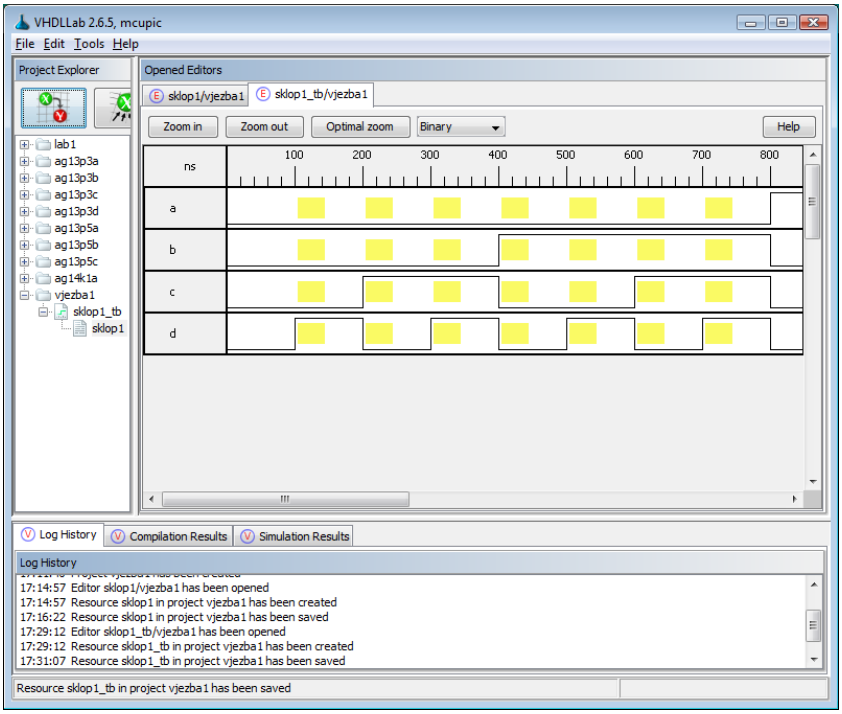
U dijalogu koji se otvori provjerite da je kao odabrani projekt doista označen `vjezba1` (ako nije, odaberite ga), i potom odaberite da želite ispitati sklop čije je ime `sklop1`. Na dnu dijaloga sustav će automatski predložiti da se ispitni sklop zove `sklop1_tb`, što je uobičajena konvencija koje se valja držati. Pritisnite `Finish`.



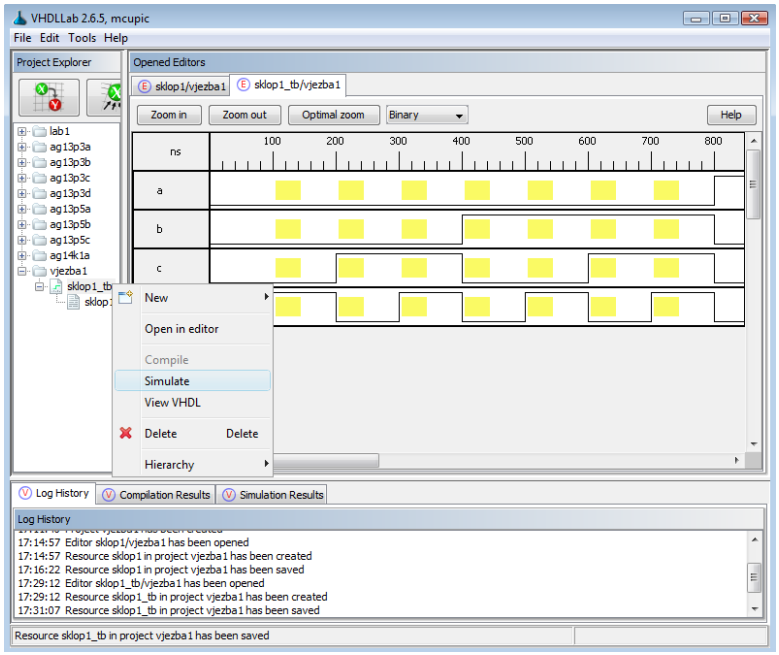
Potom će Vas sustav pitati je li to kombinacijski ili sekvencijski sklop, i koje vremenske parametre želite koristiti. Ostavite sve stavke kako su podešene, i samo pritisnite `OK`.



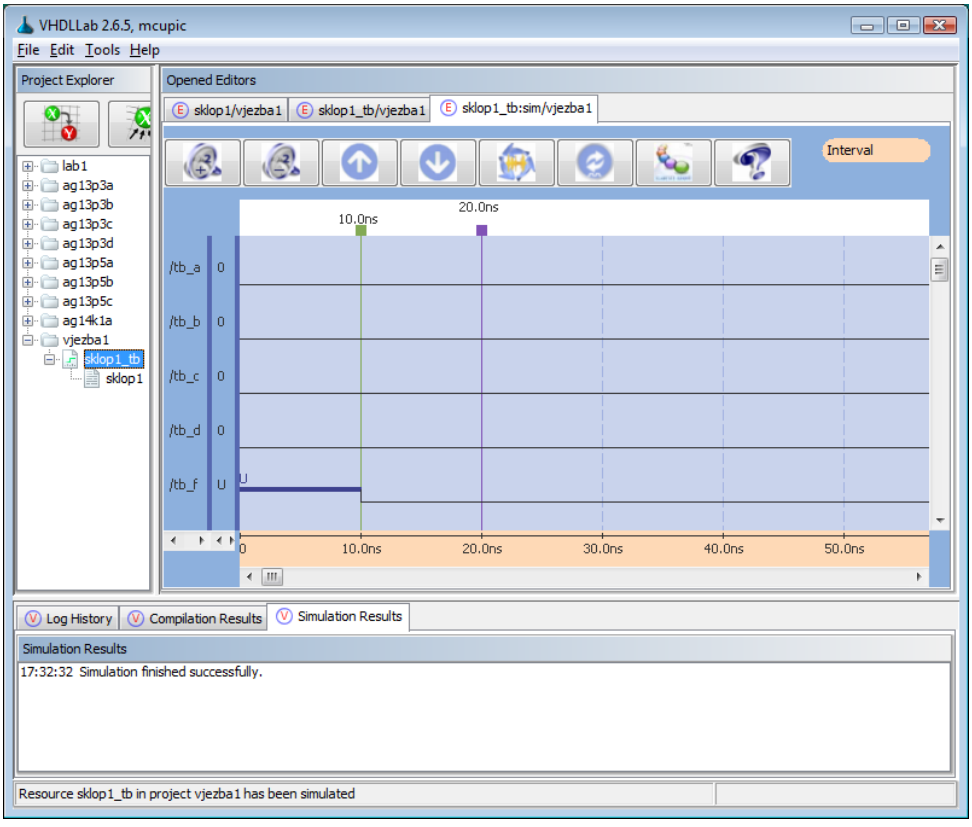
Pred Vama će se otvoriti uređivač ispitnog sklopa. Uređivač će Vam omogućiti da svakih 100 nanosekundi promijenite pobudu koja će biti dovedena na ispitivani sklop. Vaš je zadatak osigurati da rad sklopa ispitate u cijelosti: to znači da u trenutku t=0 ns na ulaze a, b, c i d dovodite vrijednosti 0, 0, 0, 0; u trenutku t=100 ns vrijednosti 0, 0, 0, 1, i tako redom sve do trenutka t=1500 ns kada na ulaze treba dovesti 1, 1, 1, 1. Kad ste gotovi, snimite ispitni sklop.



Nakon što je ispitni sklop napravljen i snimljen, možemo ga pokrenuti. Napravite desni klik na naziv ispitnog sklopa u stablu projekta s lijeve strane, i odaberite stavku `Simulate`.



Sustav će na poslužitelju napraviti simulaciju. Otvorit će se novi prozor u kojem će biti prikazani rezultati simulacije.

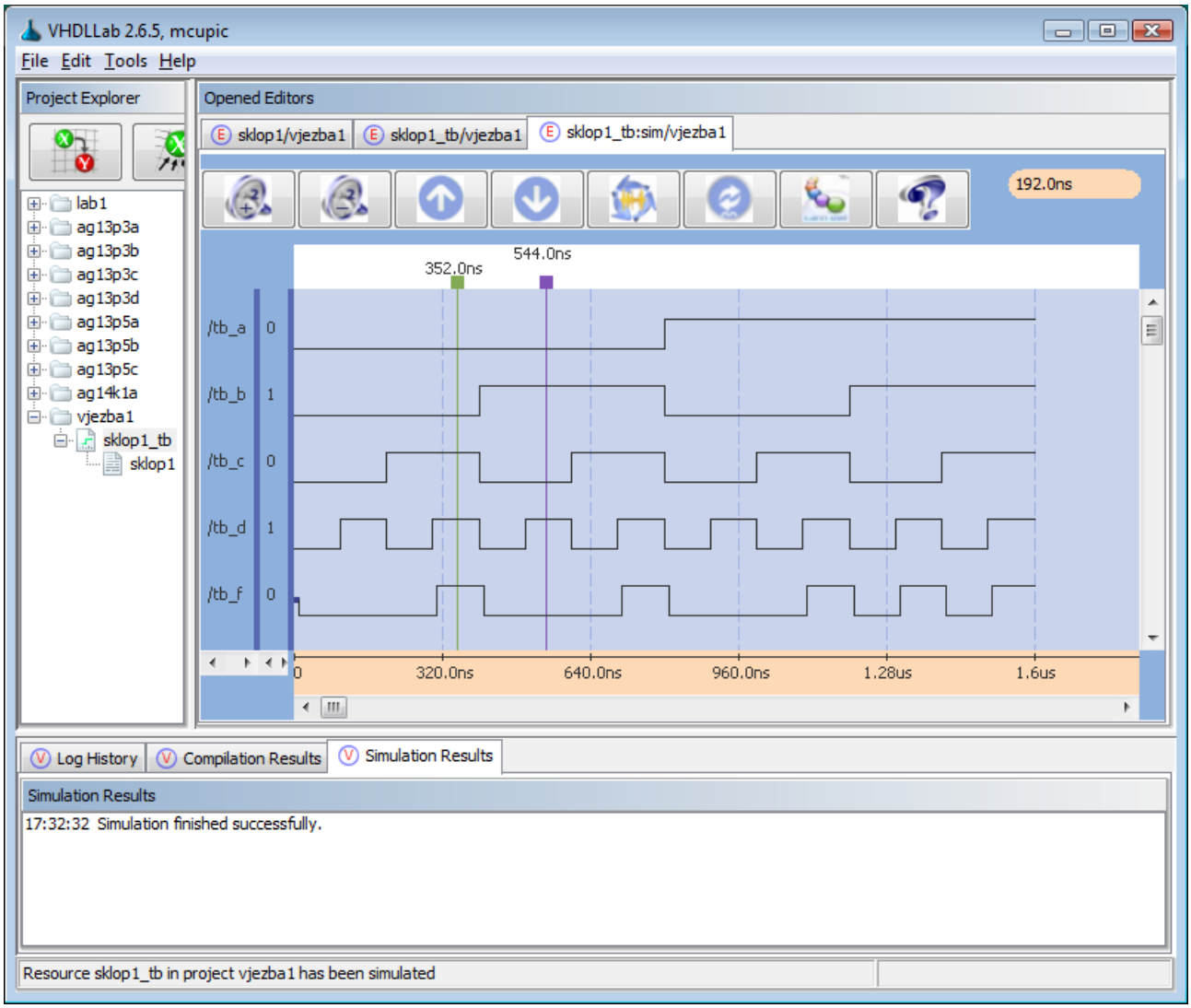


Pogledajmo malo bolje te rezultate. Inicijalno je odabran takav faktor uvećanja uz koji se na zaslonu prikazuje prvih pedesetak nanosekundi simulacije. Naime, analizom rezultata, sustav je zaključio da

se već u tom dijelu počinje događati nešto zanimljivo. Prva četiri retka prikazuju kako se kroz vrijeme mijenjaju signali a, b, c i d: mijenjaju se upravo onako kako smo ih u uređivaču ispitnog sklopa podesili. U t=0 ns sva četiri signala postavljena su na vrijednost '0', i čeka se odgovor našeg ispitivanog sklopa da postavi nešto na izlaz f.

Pogledajte što se događa na izlazu f. U trenutku t=0 ns izlaz f postavljen je na vrijednost U, i tako ostaje još neko vrijeme prije no što se promijeni i postane '0'. Vidite li sa slike koliko je vremena na izlazu vrijednost 'U'? Možete li to povezati s načinom na koji smo modelirali naš sklop? Koja je točno naredba odgovorna za to?

Ako ste odgovorili na prethodna pitanja, sada možete malo odzumirati simulaciju. U tu svrhu koristite alat za odzumiravanje (malo plavo povećalo sa znakom minusa u alatnoj traci). Odzumiravajte sve dok ne dobijete prikaz kao na sljedećoj slici.



Na prikazanoj slici jedan od kursora je prebačen na t=544 ns a drugi (zeleni, aktivan) na t=352 ns. Vrijednosti svih signala u trenutku u kojem se nalazi zeleni kursor ujedno su prikazana (očitana) uz lijevi rub simulacije, između naziva signala i njihovih valnih oblika.

Iz simulacije vidimo da funkcija postaje '1' u 5 slučajeva: kada su a, b, c i d redom 0,0,1,1, zatim kada su 0,1,1,1, zatim kada su 1,0,1,1, zatim kada su 1,1,0,1 i konačno kada su 1,1,1,1. Je li to u redu?

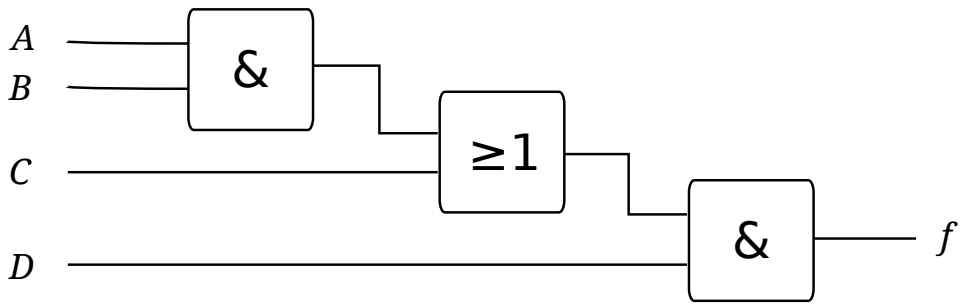
Definirate li ispitni sklop na prikazani način (tako da navedete sve kombinacije ulaza i to redosljedom kako biste ih inače pisali u tablici istinitosti, u određenom smislu simulacija Vam daje prikaz stvarne tablice istinitosti kroz vrijeme: svakih 100 nanosekundi izmjenjuje se jedan redak tablice). Ako ste funkciju prethodno i sami tabelirali, tada bi se provjera radi li sklop ispravno trebala svesti na usporedbu Vaše tablice i tablice koju je dala simulacija.

2.4.2 Strukturni opis sklopa

Strukturni opis sklopa polazi od pretpostavke da ćemo sklop opisivati uporabom jednostavnijih gotovih komponenata i njihovim povezivanjem. Strukturni model sklopa konceptualno je upravo ono što radimo kada sklop definiramo uporabom shematica.

Za ilustraciju ćemo u nastavku napraviti strukturni model sklopa koji ostvaruje već prethodno razmatranu funkciju: $f = (a \cdot b + c) \cdot d$. Sklop ćemo nazvati sklop2.

Shematski prikaz ovog sklopa dan je u nastavku.



Stoga ćemo pretpostaviti da u projektu već imamo napravljene gotove jednostavnije komponente koje odgovaraju sklopovima od kojih se sastoji prikazana shema. Konkretno, pretpostavit ćemo da u projektu imamo napravljen model `SklopAND` koji predstavlja sklop koji ima dva ulaza i jedan izlaz i računa logičko I, te da imamo napravljen model `SklopOR` koji predstavlja sklop koji ima dva ulaza i jedan izlaz i računa logičko ILI.

S obzirom da te sklopove još nemamo u projektu, evo njihovih definicija u nastavu (uočite, te smo sklopove opisali modelom toka podataka). Dodajte ih u projekt (napravite dva nova VHDL modela).

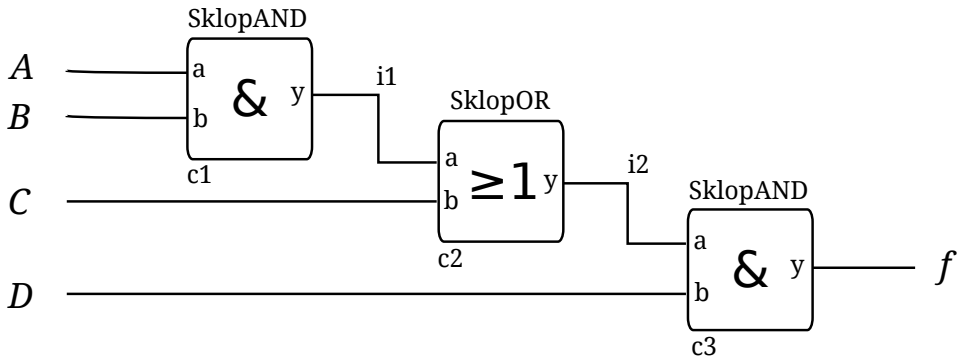
Listing 2.2 : Model toka podataka sklopa SklopAND

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  ENTITY SklopAND IS PORT (
5    a, b: IN std_logic;
6    y: out std_logic);
7  END SklopAND;
8
9  ARCHITECTURE arch1 OF SklopAND IS
10 BEGIN
11   y <= a AND b AFTER 10 ns;
12 END arch1;
```

Listing 2.3 : Model toka podataka sklopa SklopOR

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

ENTITY SklopOR IS PORT (
   a, b: IN std_logic;
   y: out std_logic);
END SklopOR;

ARCHITECTURE arch1 OF SklopOR IS
BEGIN
   y <= a OR b AFTER 10 ns;
END arch1;
```

Da bismo prikazali strukturni model sklopa sklop2, najprije ćemo nadopuniti prethodnu shemu, kako je prikazano u nastavku.



Što smo napravili?

- Iznad svakog sklopa dopisali smo naziv komponente koja u projektu predstavlja model tog sklopa.
- Ispod svakog sklopa naveli smo jedinstveno ime primjerka komponente. U shemi imamo ukupno tri primjerka od dvije komponente: imamo dva primjerka komponente SklopAND (nazvali smo ih c1 i c3) te jedan primjerak komponente SklopOR (nazvali smo ga c2).
- U svaki sklop ucrtali smo "lokalna" imena ulaza i izlaza. U skladu s prethodno danom definicijom, sklopovi SklopAND i SklopOR imaju ulaze a i b te izlaz y.
- Svim žicama koje nisu niti ulazi sklopa niti izlazi sklopa (odnosno koje se izvana ne vide i ne čine sučelje sklopa) dali smo imena. Te će žice postati interni signali: vodiči kojima ćemo povezivati komponente na način koji se izvana ne vidi.

Sada smo spremni za pisanje strukturnog modela. Model je prikazan u nastavku.

Listing 2.4 : Strukturni model sklopa

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  ENTITY sklop2 IS PORT (
5    a,b,c,d: IN std_logic;
6    f: out std_logic);
7  END sklop2;
8
9  ARCHITECTURE arch1 OF sklop2 IS
10   SIGNAL i1, i2: std_logic;
11 BEGIN
12   c1: ENTITY work.SklopAND PORT MAP (a, b, i1);
13   c2: ENTITY work.SklopOR PORT MAP (i1, c, i2);
14   c3: ENTITY work.SklopAND PORT MAP (i2, d, f);
15 END arch1;
```

Usporedite `sklop1` i `sklop2`: prvog smo opisali modelom toka podataka a drugog strukturnim modelom. *Sučelja* su u oba slučaja jednaka: radi li se o ponašajnom ili o strukturnom opisu, nije moguće zaključiti iz sučelja sklopa već isključivo iz arhitekture sklopa. Stoga u strukturnom opisu sve do retka 10 nema nikakvih razlika u odnosu na ponašajni opis.

U retku 10 dodali smo deklaraciju dvaju signala: `i1` i `i2`, oba tipa std_logic. Interni signali deklariraju se u arhitekturi sklopa prije ključne riječi BEGIN uporabom ključne riječi SIGNAL.

Nakon ključne riječi BEGIN započinje strukturni opis. Za svaki od primjeraka sklopova koje imamo u shemi naveli smo po jednu naredbu kojom se stvara primjerak komponente. Naredba uvijek započinje nazivom primjerka komponente, slijedi znak dvotočke, ključna riječ ENTITY, potom `work.` pa naziv naše komponente; potom dolaze ključne riječi PORT MAP, otvorena obla zagrada, definicija povezivanja komponente, zatvorena obla zagrada pa točka-zarez.

Prethodni primjer u sva tri retka koristi *pozicijsko povezivanje*: u oblim zagradama redosljedom kojim su ulazi i izlazi definirani u sučelju komponente čiji primjerak stvaramo navodimo ulaze, izlaze odnosno interne signale iz našeg sklopa koje spajamo. Pogledajmo pažljivije redak 13: u oblim zagradama piše (i1,c,i2). Iz sučelja komponente `SklopOR` pak znamo da sklop najprije ima ulaz a, potom ulaz b pa izlaz y. Slijedi da smo na ulaz a komponente `SklopOR` doveli interni signal $i1$, na ulaz b komponente `SklopOR` naš vlastiti ulaz c a na izlaz y komponente `SklopOR` naš interni signal $i2$.

Osim pozicijskog povezivanja, VHDL nam dozvoljava i uporabu *povezivanja putem imena*. U tom slučaju u oblim zagradama navodimo lokalno ime ulaza/izlaza komponente, znak `=>` pa ime ulaza/izlaza/internog signala našeg sklopa koji spajamo na taj port komponente. Stoga smo tijelo arhitekture mogli napisati i ovako.

```
1    c1: ENTITY work.SklopAND PORT MAP (a=>a, b=>b, y=>i1);
2    c2: ENTITY work.SklopOR PORT MAP (a=>i1, b=>c, y=>i2);
3    c3: ENTITY work.SklopAND PORT MAP (a=>i2, y=>f, b=>d);
```

Pri pisanju povezivanja putem imena redosljed navođenja signala više nije bitan. To je ilustrirano u prethodnom primjeru kod primjerka `c3`, gdje je povezivanje izlaza komponente navedeno u sredini. Kako kod ovog povezivanja prevoditelju dajemo sve potrebne informacije, redosljed kojim ih navodimo postaje nebitan.

Valja napomenuti da se primjerci komponenti mogu stvarati na još jedan način. Kako se tu zahtjevaju promjene na dva mjesta, cjelovit modificirani primjer prikazan je u nastavku.

Listing 2.5 : Strukturni model sklopa uz deklaraciju komponenti

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  ENTITY sklop2 IS PORT (
5    a,b,c,d: IN std_logic;
6    f: out std_logic);
7  END sklop2;
8
9  ARCHITECTURE arch1 OF sklop2 IS
10   component SklopAND is port (
11     a : IN STD_LOGIC;
12     b : IN STD_LOGIC;
13     y : OUT STD_LOGIC
14   );
15   end component;
16   component SklopOR is port (
17     a : IN STD_LOGIC;
18     b : IN STD_LOGIC;
19     y : OUT STD_LOGIC
20   );
21   end component;
22   SIGNAL i1, i2: std_logic;
23 BEGIN
24   c1: SklopAND PORT MAP (a, b, i1);
25   c2: SklopOR PORT MAP (i1, c, i2);
26   c3: SklopAND PORT MAP (i2, d, f);
27 END arch1;
```

U čemu su razlike? Počev od retka 10 u arhitekturi smo uz definiciju potrebnih internih signala ponudili i definiciju sučelja svih korištenih komponenata. Ova se definicija radi uporabom ključne riječi COMPONENT, nakon čega je ostatak jednak kao kod definicije sučelja komponente. Tako u retcima 10 do 15 dajemo definiciju sučelja komponente `SklopAND` a u retcima 16 do 21 definiciju sučelja komponente `SklopOR`.

Uz te definicije sada više nema potrebe da se u arhitekturi sklopa prilikom stvaranja primjeraka komponenata piše ENTITY work.; dovoljno je samo navesti naziv komponente (ilustrirano u retcima 24–26).

Postoji još razlika između ova dva načina opisivanja, ali te nam razlike na ovom kolegiju nisu značajne pa ih nećemo niti navoditi.

2.4.3 Ponašajni opis sklopa

Kod ponašajnog modela sklopa njegovu funkcionalnost ne opisujemo oslanjajući se isključivo na logičke operacije, već koristimo proceduralno programiranje i naredbe za kontrolu toka izvođenja. U arhitekturi sklopa, za to se koriste blokovi process. Svaki blok process može imati svoje ime (piše se ispred ključne riječi process i odvaja dvotočkom), može imati *listu osjetljivosti* (to je popis signala koji simulatoru kažu da se pri promjenama tih signala program naveden u bloku process mora ponovno izvesti), može definirati vlastite varijable (varijable su slične signalima samo što se deklariraju unutar bloka process prije ključne riječi begin, vidljive su samo u tom bloku, ne omogućavaju modeliranje kašnjenja i vrijednost im se pridružuje znakom := a ne kao kod signala znakom <=), i konačno unutar tijela bloka koje je omeđeno ključnim riječima begin i end process dolazi programski opis.

Primjer ponašajnog opisa sklopa za koji smo već prikazali i model toka podataka i strukturni model prikazan je u nastavku.

Listing 2.6 : Ponašajni model sklopa

```
1  library IEEE;
2  use IEEE.std_logic_1164.all;
3
4  ENTITY sklop3 IS PORT (
5    a: IN std_logic;
6    b: IN std_logic;
7    c: IN std_logic;
8    d: IN std_logic;
9    f: out std_logic);
10 END sklop3;
11
12 ARCHITECTURE arch1 OF sklop3 IS
13 BEGIN
14
15 fja: process(a,b,c,d)
16        variable tmp: std_logic;
17      begin
18        tmp := '0';
19        if d='1' then
20          if a='1' and b='1' then
21            tmp := '1';
22          elsif c='1' then
23            tmp := '1';
24          end if;
25        end if;
26        f <= tmp after 10 ns;
27      end process;
28
29 END arch1;
```

Sve naredbe napisane unutar bloka process na početku simulacije izvedu se jednom i potom se njegovo izvođenje suspendira tako dugo dok se ne promijeni neki od signala unutar navedene liste osjetljivosti (u našem primjeru čine je signali a, b, c i d). Kada se bilo koji od tih signala promijeni, naredbe u bloku process ponovno se izvedu i blok se opet zamrzne na kraju. Blok process s listom osjetljivosti ponaša se kao beskonačna petlja koja na dnu ima naredbu koja zamrzava izvođenje sljedeće iteracije sve dok se ne detektira promjena vrijednosti na nekom od signala iz liste osjetljivosti. Ekvivalentni blok process onome iz prethodnog primjera prikazan je u nastavku: sada nemamo listu osjetljivosti već eksplicitnu naredbu koja zamrzava prelazak u novu iteraciju sve do promjene na bilo kojem od navedenih signala.

Listing 2.7 : Blok process bez liste osjetljivosti

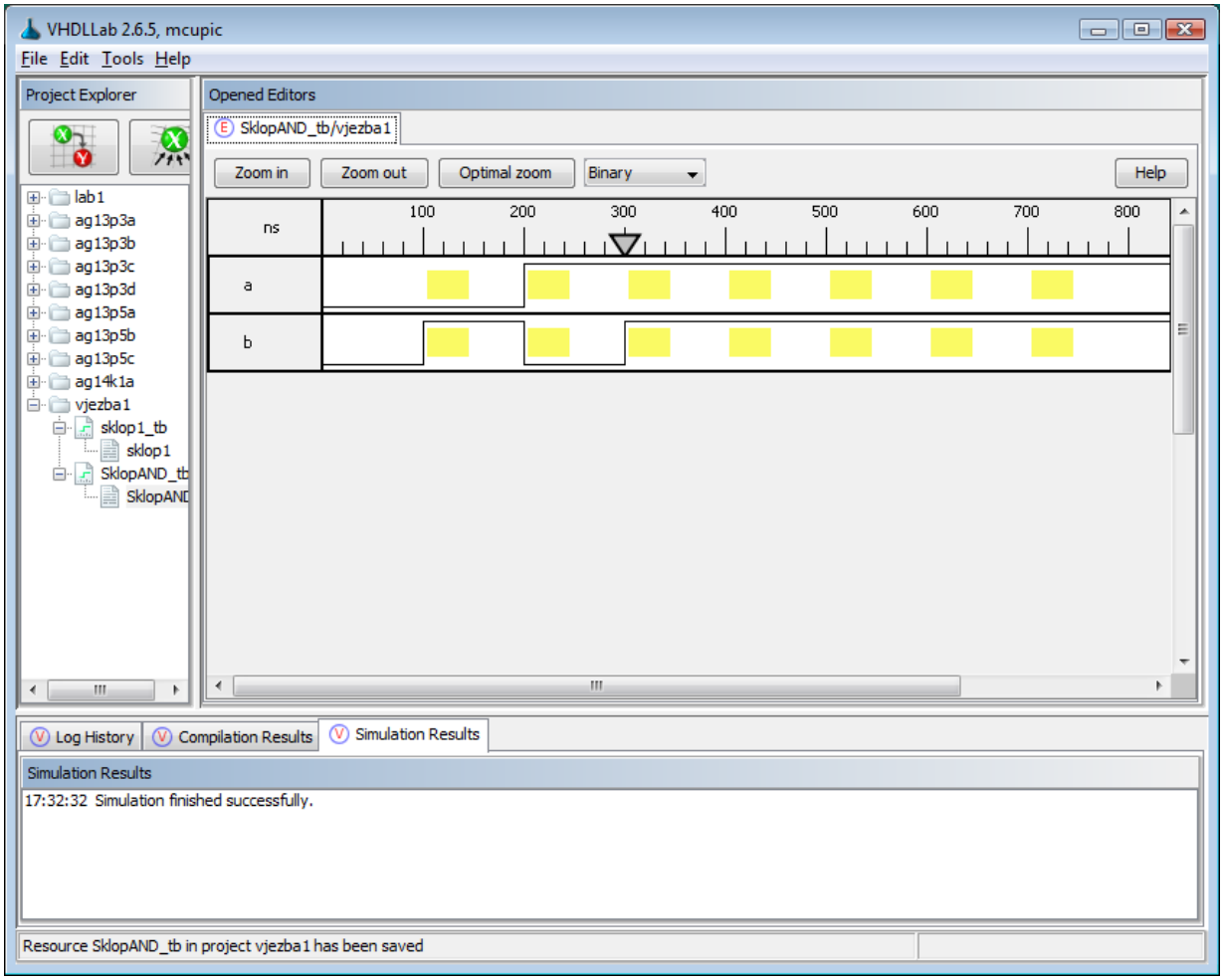
```
1  fja: process
2         variable tmp: std_logic;
3       begin
4         tmp := '0';
5         if d='1' then
6           if a='1' and b='1' then
7             tmp := '1';
8           elsif c='1' then
9             tmp := '1';
10          end if;
11        end if;
12        f <= tmp after 10 ns;
13        wait on a, b, c, d;
14      end process;
```

Pogledajmo sada ovaj posljednji opis. U bloku process definirali smo pomoćnu varijablu tmp (redak 2). Potom je u tijelu bloka najprije inicijaliziramo na vrijednost '0' pa nizom if naredbi provjeravamo i

po potrebi korigiramo njezinu vrijednost na '1' (retci 5 do 11). Konačno, u retku 12 imamo naredbu koja izlazu f pridružuje izračunatu vrijednosti uz kašnjenje od 10 nanosekundi. U retku 13 imamo naredbu koja suspendira prelazak u sljedeću iteraciju sve dok se ne detektira promjena na bilo kojem od signala a, b, c ili d. Kod verzije bloka process koji ima eksplicitno navedenu listu osjetljivosti ovaj redak nije potreban.

2.5 Ispitni sklop

Ostalo nam je za odgovoriti na još jedno pitanje: što je zapravo *ispitni sklop*? Do sada smo se s ispitnim sklopovima upoznali koristeći "čarobnjake" sustava VHDLLab2 koji su nam nudili lijepo grafičko sučelje za njihovo uređivanje, i kada smo pokrenuli simulaciju, magija se dogodila i mi smo dobili rezultate. Da bismo demistificirali ispitne sklopove, vratimo se osnovama. Kroz prethodne primjere napravili ste i u projekt dodali model toka podataka sklopa `SklopAND`. Napravimo za njega ispitni sklop pomoću ugrađenog čarobnjaka, i podesimo pobudu tako da u t=0 ns postavi ulaze a i b na '0', u t=100 ns postavi a na '0' a b na '1', u t=200 ns postavi a na '1' a b na '0' te u t=300 ns postavi a na '1' i b na '1'. Kad ste gotovi, snimite ispitni sklop.



U stablu projekta s lijeve strane odaberite sada mišem sam ispitni sklop `SklopAND_tb` i iz iskočnog izbornika odaberite stavku `View VHDL`. Iznenađenje: ispitni sklop nije ništa drugo doli još jedan sklop koji je opisan u VHDL-u. VHDL kôd koji ćete dobiti prikazan je i ovdje u nastavku.

Listing 2.8 : Ispitni sklop

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity SklopAND_tb is
5  end SklopAND_tb;
6
7  architecture structural of SklopAND_tb is
8    component SklopAND is port (
9      a : IN STD_LOGIC;
10     b : IN STD_LOGIC;
11     y : OUT STD_LOGIC);
12   end component;
13
14   signal tb_a : STD_LOGIC;
15   signal tb_b : STD_LOGIC;
16   signal tb_y : STD_LOGIC;
17 begin
18
19     -- primjerak sklopa koji se ispituje
20     uut: SklopAND port map ( tb_a, tb_b, tb_y );
21
22     -- sklop koji generira zadane ispitne uzorke
23   process
24   begin
25     tb_a <= '0';
26     tb_b <= '0';
27     wait for 100 ns;
28
29     tb_b <= '1';
30     wait for 100 ns;
31
32     tb_a <= '1';
33     tb_b <= '0';
34     wait for 100 ns;
35
36     tb_b <= '1';
37     wait for 100 ns;
38     wait;
39   end process;
40
41 end structural;
```

Uočimo: ispitni sklop je sklop čije je sučelje prazno; on nema niti ulaza, niti izlaza (retci 4 i 5). U arhitekturi sklopa prije ključne riječi BEGIN dana je deklaracija sučelja komponente `SklopAND_tb`: to je *ispitivana komponenta*. Potom je još deklarirano upravo onoliko internih signala koliko ispitivani sklop ima ulaza i izlaza. Tijelo arhitekture sastoji se od samo dvije naredbe (redak 20 te redak 23).

U retku 20 dana je naredba koja stvara jedan primjerak ispitivane komponente i interne signale spaja na njezine ulaze i izlaze. Tom je primjerku dano ime `uut` (što je kratica od engleske fraze *Unit Under Test*).

Druga naredba je naredba process: ona se proteže od retka 23 pa sve do retka 39, i VHDL čitav njezin sadržaj tretira kao jednu naredbu, odnosno ta čitava naredba modelira novi sklop koji se također nalazi u našem ispitnom sklopu. Specifičnost ove naredbe (bloka process) je da nam omogućava da dio funkcionalnosti sklopa opisujemo gotovo kao u klasičnom programskom jeziku: naredbe koje se navedu unutar bloka process sustav izvršava slijedno. Blok process kakav je napisan u ovom primjeru ponaša se kao beskonačna petlja čije bi se naredbe neprestano ponavljale (više ćemo naučiti nešto kasnije).

Pogledajmo sada malo detaljnije naredbe unutar bloka process. Retci 25 i 26 na interne signale `tb_a` i `tb_b` postavljaju vrijednost '0'. No ti isti interni signali spojeni su na ulaze `a` i `b` naše ispitivane komponente, čime ovaj blok process postavlja vrijednosti koje se kao ulazi dovode na ispitivanu komponentu koja će odreagirati na odgovarajući način i svoj izlaz `y` postaviti na odgovarajuću vrijednost (uz zadano kašnjenje); ta će vrijednost biti vidljiva i na internom signalu `tb_y` jer je izlaz `y` spojen na njega.

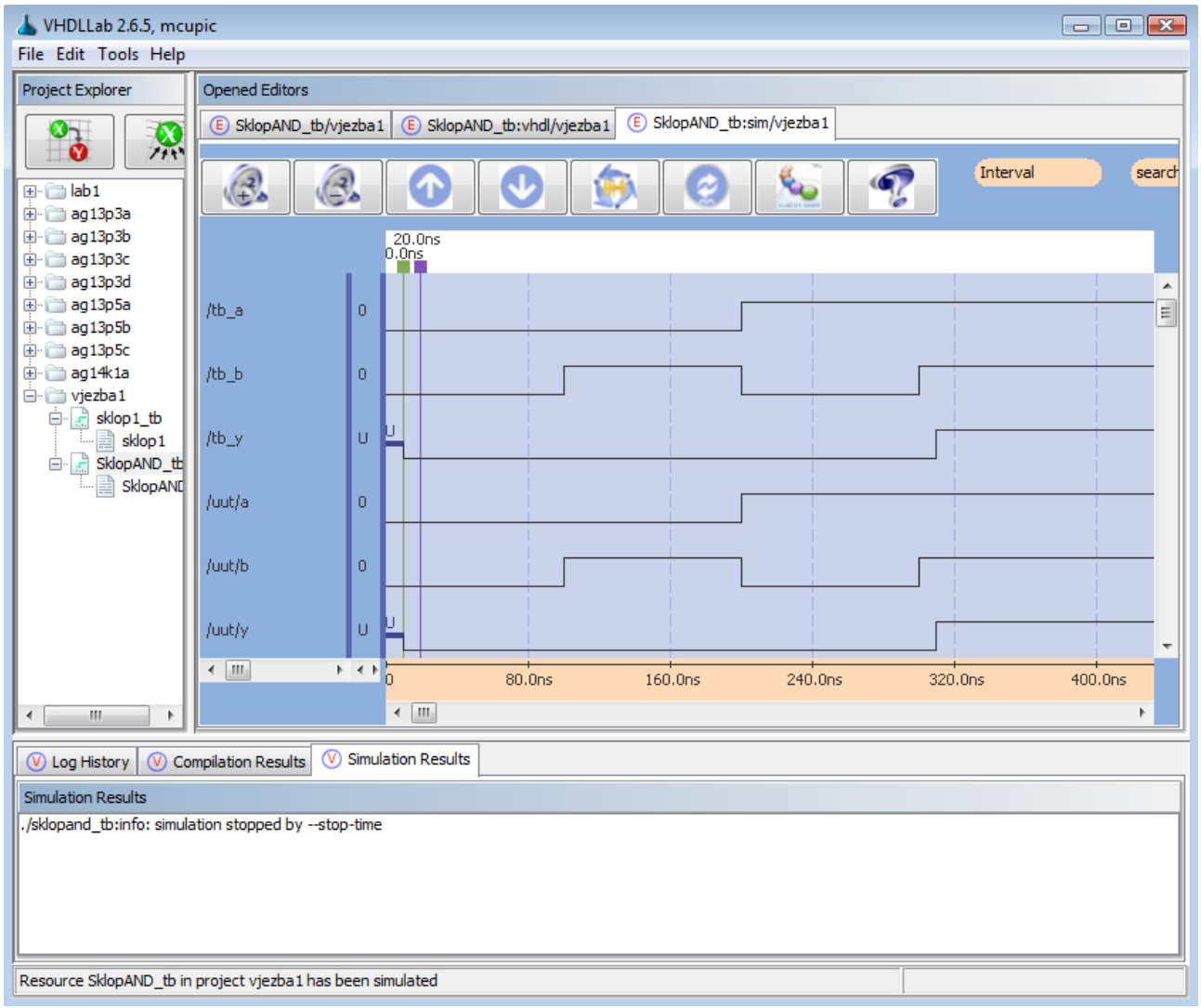
Izvođenje ovog bloka potom se zamrzava u trajanju od 100 ns (redak 27, naredba wait for), nakon čega naredba u retku 29 mijenja vrijednost internog signala `tb_b` na '1' (`tb_a` ostaje kakav je bio) i blok process se opet zaglavljuje na sljedećih 100 ns (redak 30). Postupak se ponavlja još dva puta (retci 32 do 34, pa retci 36 do 37). Na ovaj način blok process na interne je signale postavio ulazne signale upravo u skladu s onime kako smo ih mi naklikali u grafičkom uređivaču. Kako bi se spriječilo da se ova pobuda sada ciklički ponavlja, blok process terminira se u retku 38 naredbom bezuvjetnog čekanja wait; iz koje blok process više nikada ne može izaći.

Ako u simuliranom sustavu u tom trenutku više nema nikakvih promjena, ovo je i trenutak u kojem će završiti i simulacija.

Još nam je ostalo da pokrenemo simulaciju i promotrimo rezultate (slika u nastavku). Pogledajte sada malo pažljivije popis svih signala koje prikazuje simulator. Uočite da postoji 6 signala.

- `/tb_a`: to je interni signal koji se nalazi u ispitnom sklopu; vrijednosti mu postavlja blok process a čita naš ispitivani sklop.
- `/tb_b`: to je interni signal koji se nalazi u ispitnom sklopu; vrijednosti mu postavlja blok process a čita naš ispitivani sklop.
- `/tb_y`: to je interni signal koji se nalazi u ispitnom sklopu; vrijednosti mu postavlja naš ispitivani sklop.
- `/uut/a`: to je ulaz sklopa `SklopAND`; vrijednosti mu izvana postavlja naš ispitni sklop, a vrijednost čita naredba u arhitekturi sklopa `SklopAND` koja određuje vrijednost izlaza `y`.
- `/uut/b`: to je ulaz sklopa `SklopAND`; vrijednosti mu izvana postavlja naš ispitni sklop, a vrijednost čita naredba u arhitekturi sklopa `SklopAND` koja određuje vrijednost izlaza `y`.
- `/uut/y`: to je izlaz sklopa `SklopAND`; vrijednosti mu postavlja naredba u arhitekturi sklopa `SklopAND` koja određuje vrijednost tog izlaza.

Da smo imali bogatije strukturne modele (sklop koji sadrži sklop koji sadrži sklop) popis signala bio bi veći i bogatiji. U rezultatima simulacije za svaki od postojećih signala mogli biste vidjeti vrijednosti u bilo kojem trenutku.



Primjetimo: ispitni sklop modeliran je hibridnim modelom.

2.6 O kašnjenjima

Jezik VHDL uz niz ključnih riječi koje služe za modeliranje funkcionalnosti sklopova nudi i neke jezične konstrukte koji omogućavaju specificiranje kašnjenja. Primjerice, već smo se upoznali s naredbom oblika:

```
1   f <= a AND b AFTER 10 ns;
```

Treba napomenuti da ti jezični konstrukti nisu sintetizabilni: opišete li digitalni sklop na takav način i potom zatražite od sintetizatora digitalnog sklopovlja da takav opis "utoči" u Vaš programirljivi sklop (primjerice, FPGA), sintetizator će naredbe kašnjenja zanemariti. Naime, prilikom sinteze sklopa iz VHDL opisa sintetizator na raspolaganju ima programirljiv sklop koji ste mu zadali i koji je ostvaren u određenoj tehnologiji. U toj tehnologiji, komponente će imati kašenjenja koja su određena samom tehnologijom i na koja Vi (niti sintetizator) ne možete utjecati.

Zapitate li se zašto onda takve naredbe uopće postoje, odgovor je jednostavan: kako bi se omogućilo izvođenje što je moguće realnijih simulacija. Naime, svaki ozbiljniji alat za modeliranje i sintezu

digitalnog sklopovlja korisnicima nudi mogućnost izvođenja nekoliko različitih vrsta simulacija. Najjednostavnija simulacija je simulacija ponašajnog modela (engl. *Behavioral model simulation*) kod koje simulator simulira Vaš opis upravo onako kako ste ga definirali. Međutim, takav alat možete zatražiti i provođenje simulacije koja uzima u obzir fizički razmještaj komponenata u programirljivom sklopu i način na koji su komponente međusobno povezane (engl. *Post place and route simulation*). Ono što će se u tom trenutku dogoditi jest da će alat zanemariti Vaš polazni opis i generirati novi VHDL opis koji u obzir uzima građu programirljivih komponenata i njihova stvarna kašnjenja koja će opisivati koristeći naredbe poput prethodne navedene uz AFTER 10 ns. Takav opis potom će poslati simulatoru na simulaciju čime ćete dobiti vjerniju sliku onoga što će se u stvarnosti događati jednom kada se konačni sklop sintetizira. Ciljani korisnik ovih primitiva niste dakle Vi kao dizajner sklopa (iako Vam nitko ne brani da ih koristite, tako dugo dok znate da se to ne može tako sintetizirati) već je to sam alat u kojem radite razvoj i sintezu.

Iz upravo opisanog razloga niti sljedeći isječak kôda napisanog u VHDL-u koji se u ispitnim sklopovima često koristi za generiranje signala takta nije sintetizabilan.

Listing 2.9 : Nesintetizabilan generator signala takta

```
process
begin
  cp <= '1';
  wait for 50 ns;
  cp <= '0';
  wait for 50 ns;
end process;
```

Naime, rad opisanog sklopa temelji se na pretpostavci da je sintetizator sposoban osigurati upravo navedena kašnjenja, što naravno nije slučaj. Stoga će ovakvi dijelovi kôda često biti korišteni u ispitnim sklopovima koje će simulator moći simulirati, ali ih nikada nećemo imati u opisima sklopova koje fizički treba sintetizirati: programirljivi sklopovi za ovu će svrhu morati imati unaprijed proizveden sklop čija je zadaća generiranje signala takta, ili će u jednostavnijoj varijanti imati zaseban ulaz preko kojeg će korisnik izvana morati dovesti prikladno generiran signal takta (nekim primjerice relaksacijskim oscilatorom ili kvarcnim oscilatorom).

Poglavlje 3

Standardni kombinacijski moduli

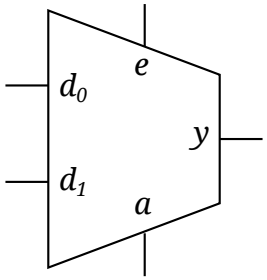
Pojam *standardni kombinacijski moduli* predstavlja zajednički nazivnik za nekoliko različitih često korištenih kombinacijskih modula, poput:

- multipleksori,
- dekoderi,
- koderi i prioritetni koderi,
- pretvornici kôda te
- komparatori.

Kroz ovo poglavlje osvrnut ćemo se na modeliranje nekih od ovih sklopova uporabom jezika VHDL.

3.1 Modeliranje multipleksora

Simbol multipleksora 2/1 s ulazom za omogućavanje prikazan je na slici u nastavku.



Prisjetimo se, kada je onemogućen ($e =' 0'$), multipleksor mora na izlazu generirati vrijednost '0'. Kada je omogućen ($e =' 1'$), na izlaz treba proslijediti vrijednost dovedenu na podatkovni ulaz d_0 ako je adresni ulaz $a =' 0'$, odnosno vrijednost dovedenu na podatkovni ulaz d_1 ako je adresni ulaz $a =' 1'$.

Ponašajni model ovog sklopa prikazan je u nastavku.

Listing 3.1 : Ponašajni model multipleksora 2/1 s ulazom za omogućavanje

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY mux21e IS PORT(
5     d: IN STD_LOGIC_VECTOR(0 TO 1);
6     a: IN STD_LOGIC;
7     e: IN STD_LOGIC;
8     y: OUT STD_LOGIC);
9  END mux21e;
10
11 ARCHITECTURE arch OF mux21e IS
12 BEGIN
13    process(d,a,e)
14    begin
15       if e='0' then
16          y <= '0';
17       elsif a='0' then
18          y <= d(0);
19       elsif a='1' then
20          y <= d(1);
21       else
22          y <= '0';
23       end if;
24    end process;
25
26 END arch;
```

U sučelju sklopa, podatkovne ulaze smo modelirali dvobitnim vektorom (redak 5 prethodnog modela) dok su za sve ostalo korišteni jednobitni signali tipa std_logic (retci 6, 7 i 8).

Arhitektura je u cijelosti opisana ponašajno, jednim blokom process čija je zadaća na temelju ulaza d, a i e odrediti vrijednost izlaza y; stoga se u listi osjetljivosti bloka process nalaze upravo ulazi d, a i e.

Za potrebe opisa funkcionalnosti koristi se if-pitalica. Ako je multipleksor onemogućen, na izlaz se stavlja vrijednost 0 (retci 15, 16); u suprotnom se provjerava je li adresni ulaz postavljen na '0' i ako je, na izlaz se prosljeđuje podatak doveden na podatkovni ulaz d_0 (retci 17, 18); inače se provjerava je li a postavljeno na '1' pa ako je, na izlaz se prosljeđuje podatak doveden na podatkovni ulaz d_1 (retci 19, 20); konačno, ako ništa od prethodnoga nije bilo zadovoljeno (*razmislite zašto je to moguće*), na izlaz se postavlja vrijednost '0' (retci 21, 22).

Uočite sintaksu naredbe if kada ispitujemo više opcija: koristi se elsif (ne else if) te za posljednju opciju else. Čitav izraz zatvara se s end if;.

Umjesto ovakve if-pitalice mogli smo iskoristiti i naredbu case. Ekvivalentan blok process za taj slučaj prikazan je u nastavku.

Pogledajmo i sintaksu naredbe case. Nakon ključne riječi case dolazi signal ili varijabla čija se vrijednost razmatra, pa ključna riječ is. Potom se navode opcije. Svaka opcija započinje ključnom riječi when, slijedi vrijednost pa znak => nakon čega dolaze naredbe pridruživanja (u našem slučaju dodjeljivanje vrijednosti izlazu y). Naredba case završava navođenjem ključnih riječi end case;.

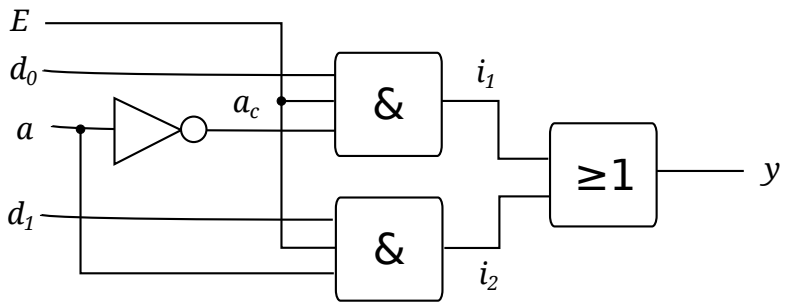
Ako se kroz eksplicitno navođenje opcija ne pokriju sve vrijednosti koje razmatrani signal/varijabla može imati, nužno je dodati još opciju koja će pokupiti te sve vrijednosti, što se radi navođenjem ključne riječi others. Prisjetite se, ulaz a je tipa std_logic pa kao takav može poprimiti jednu od 9 različitih vrijednosti. Ako bismo se u opisu izjasnili samo o vrijednostima '0' i '1', preostalo bi nam još 7 nepokrivenih vrijednosti. Stoga je u ovom slučaju navođenje when others nužno; njegovim

izostavljanjem napisani se kôd neće moći prevesti i generirat će se odgovarajuća poruka pogreške.

Listing 3.2 : Opis koji se temelji na naredbi case

```
process(d,a,e)
begin
  if e='0' then
    y <= '0';
  else
    case a is
      when '0'    => y <= d(0);
      when '1'    => y <= d(1);
      when others => y <= '0';
    end case;
  end if;
end process;
```

Da bismo prikazali model multipleksora koji se temelji na toku podataka, prikažimo njegovo ostvarenje osnovnim logičkim sklopovima, kako je prikazano na slici u nastavku.



Internim signalima (žicama koje nisu niti ulazi niti izlazi) dana su imena ac, i1 te i2. Sada možemo prikazati model toka podataka.

Listing 3.3 : Opis koji se temelji na toku podataka

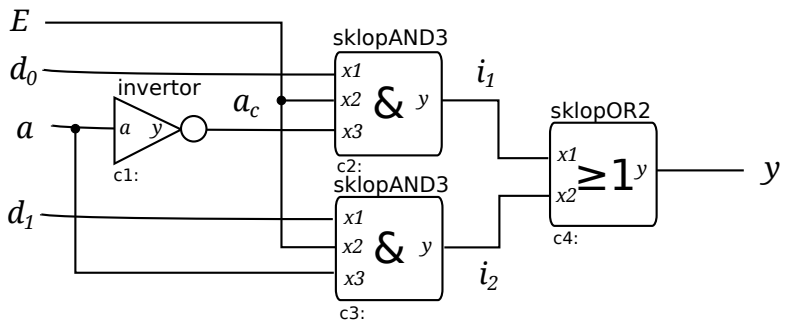
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY mux21e IS PORT(
  d: IN STD_LOGIC_VECTOR(0 TO 1);
  a: IN STD_LOGIC;
  e: IN STD_LOGIC;
  y: OUT STD_LOGIC);
END mux21e;

ARCHITECTURE arch OF mux21e IS
  SIGNAL ac, i1, i2: std_logic;
BEGIN
  ac <= not a;
  i1 <= e and ac and d(0);
  i2 <= e and a and d(1);
  y <= i1 or i2;
END arch;
```

Umjesto uprabe internih signala, s obzirom da se radi o kombinacijskom sklopu, možemo napisati i arhitekturu koja sadrži samo konačan izraz kojim je određen izlaz multipleksora, što je ilustrirano u nastavku.

Listing 3.4 : Kraći opis koji se temelji na toku podataka

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY mux21e IS PORT(
   d: IN STD_LOGIC_VECTOR(0 TO 1);
   a: IN STD_LOGIC;
   e: IN STD_LOGIC;
   y: OUT STD_LOGIC);
END mux21e;

ARCHITECTURE arch OF mux21e IS
BEGIN
   y <= e and ((not a and d(0)) or (a and d(1)));
END arch;
```

Naposlijetku, dajmo još i strukturni model ovakvog multipleksora. Pretpostavit ćemo da u projektu već imamo na raspolaganju komponente invertor (invertor), troulazni sklop I (sklopAND3) i dvoulazni sklop ILI (sklopOR2). Nazivi komponenata te nazivi primjeraka komponenata prikazani su na slici u nastavku.



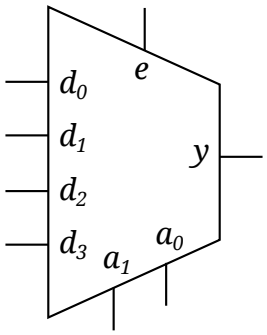
Pripadni VHDL model dan je u nastavku.

Listing 3.5 : Strukturni model multipleksora

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY mux21e IS PORT(
   d: IN STD_LOGIC_VECTOR(0 TO 1);
   a: IN STD_LOGIC;
   e: IN STD_LOGIC;
   y: OUT STD_LOGIC);
END mux21e;

ARCHITECTURE arch OF mux21e IS
  SIGNAL ac, i1, i2: std_logic;
BEGIN

   c1: ENTITY work.invertor PORT MAP(a=>a, y=>ac);
   c2: ENTITY work.sklopAND3 PORT MAP(x1=>d(0), x2=>e, x3=>ac, y=>i1);
   c3: ENTITY work.sklopAND3 PORT MAP(x1=>d(1), x2=>e, x3=>a, y=>i2);
   c4: ENTITY work.sklopOR2 PORT MAP(x1=>i1, x2=>i2, y=>y);

END arch;
```

U nastavku ćemo još pogledati dva modela multipleksora 4/1: najprije ponašajni model a potom strukturni model koji oslikava izvedbu ovog multipleksora multipleksorskim stablom koje kao osnovnu gradbenu komponentu koristi prethodno opisani multipleksor 2/1 s ulazom za omogućavanje.

Simbol multipleksora 4/1 s ulazom za omogućavanje prikazan je na slici u nastavku.



U sučelju sklopa sada ćemo i podatkovne i adresne ulaze modelirati vektorom.

Ponašajni model multipleksora 4/1 sličan je ponašajnom modelu multipleksora 2/1 i prikazan je u nastavku.

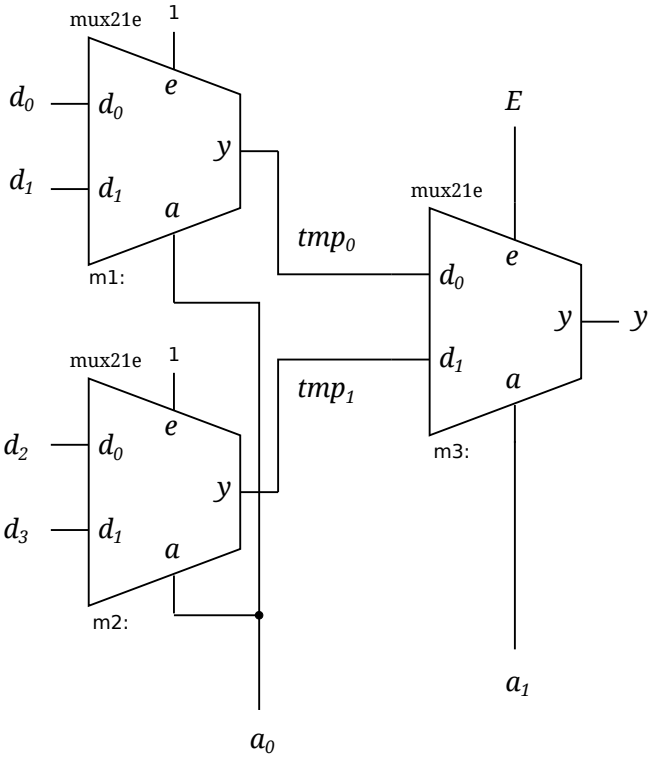
Listing 3.6 : Ponašajni model multipleksora 4/1

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY mux41e IS PORT(
   d: IN STD_LOGIC_VECTOR(0 TO 3);
   a: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
   e: IN STD_LOGIC;
   y: OUT STD_LOGIC);
END mux41e;

ARCHITECTURE arch OF mux41e IS
BEGIN

   process(d,a,e)
   begin
     if e='0' then
       y <= '0';
     else
       case a is
         when "00"   => y <= d(0);
         when "01"   => y <= d(1);
         when "10"   => y <= d(2);
         when "11"   => y <= d(3);
         when others => y <= '0';
       end case;
     end if;
   end process;

END arch;
```

U ovom slučaju naredba case imat će četiri opcije koje provjerava (uz defaultnu opciju): opcije za vrijednosti adresnih ulaza "00", "01", "10" i "11". Uočite da se, s obzirom da je a sada vektor, vrijednosti pišu pod dvostrukim navodnicima.

Za potrebe specificiranja strukturnog modela prikažimo najprije shemu multipleksorskog stabla izvedenog multipleksorima 2/1 (slika u nastavku). Iznad svake komponente napisat ćemo vrstu komponente a ispod svake komponente naziv primjerka komponente. Kako se ovdje radi o malim multipleksorima, koristit ćemo imena m1, m2 i m3 (u stablu imamo 3 primjerka malih multipleksora).

Za potrebe povezivanja dvije razine multipleksora koristit ćemo interni signal tmp koji ćemo definirati kao dvobitni vektor.



Strukturni model napravljen temeljem ove sheme prikazan je u nastavku.

Listing 3.7 : Strukturni model multipleksora 4/1 ostvarenog stablom

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY mux41e IS PORT(
5     d: IN STD_LOGIC_VECTOR(0 TO 3);
6     a: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
7     e: IN STD_LOGIC;
8     y: OUT STD_LOGIC);
9  END mux41e;
10
11 ARCHITECTURE arch OF mux41e IS
12    SIGNAL tmp: std_logic_vector(0 TO 1);
13 BEGIN
14
15    m1: ENTITY work.mux21e PORT MAP (d => d(0 to 1), a => a(0), e => '1', y => tmp(0));
16    m2: ENTITY work.mux21e PORT MAP (d => d(2 to 3), a => a(0), e => '1', y => tmp(1));
17    m3: ENTITY work.mux21e PORT MAP (d => tmp, a => a(1), e => e, y => y);
18
19 END arch;
```

Za vježbu napišite model multipleksora 4/1 s ulazom za omogućavanje koji se temelji na toku podataka (prikažite shemu ovog multipleksora koristeći samo osnovne logičke sklopove, kako je napravljeno u predavanjima).

3.2 Modeliranje binarnog dekodera

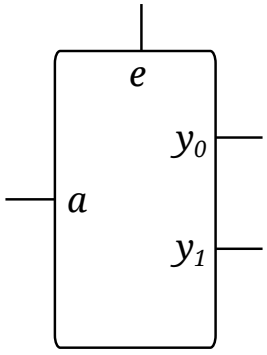
Binarni dekoder je sklop koji obavlja funkciju suprotnu onoj binarnog kodera: on na ulazu dobiva n-bitnu binarnu kodnu riječ te aktivira jedan od izlaza koji je odabran ulaznom kodnom riječi. Ulaz na koji se dovodi kodna riječ još se naziva i adresni ulaz jer podatak doveden na taj ulaz u određenom smislu "adresira" izlaz koji će biti aktiviran.

Primjerice, imamo li jednobitni binarni dekoder, on će imati jedan (jednobitni) ulaz a te dva izlaza: y_0 i y_1. Ako je $a = 0$, izlaz y_0 postavit će se u stanje 1 a ako je $a = 1$, izlaz y_1 postavit će se u stanje 1 (preostali izlaz bit će u 0).

Dvobitni binarni dekoder imat će četiri izlaza: y_0 do y_3; uz $a =$ "00" aktivan će biti y_0, uz $a =$ "01" aktivan će biti y_1, uz $a =$ "10" aktivan će biti y_2 te uz $a =$ "11" aktivan će biti y_3 (svi neaktivni izlazi bit će postavljeni na 0).

Ako dekoder raspolaže i ulazom za omogućavanje E, tada će, za slučaj da je E postavljen na 0 svi izlazi biti neaktivni (u nuli), neovisno o kodnoj riječi na adresnom ulazu. Kada je $E = 1$, dekoder će obavljati svoju "uobičajenu" funkciju.

Simbol dekodera 1/2 s ulazom za omogućavanje prikazan je na slici u nastavku.

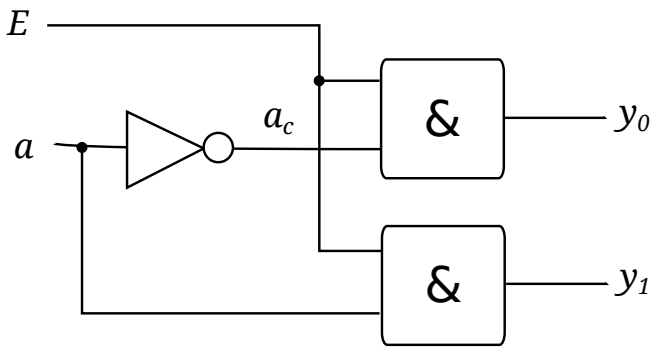


Ponašajni model ovakvog dekodera prikazan je u nastavku.

Listing 3.8 : Ponašajni model binarnog dekodera 1/2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY dek21e IS PORT(
   a: IN STD_LOGIC;
   e: IN STD_LOGIC;
   y: OUT STD_LOGIC_VECTOR(0 TO 1));
END dek12e;

ARCHITECTURE arch OF dek12e IS
BEGIN
   process(a,e)
   begin
      if e='0' then
        y <= "00";
      else
        case a is
          when '0' =>     y <= "10";
          when '1' =>     y <= "01";
          when others => y <= "00";
        end case;
      end if;
   end process;
END arch;
```

Implementacija binarnog dekodera 1/2 s ulazom za omogućavanje koja se temelji na uporabi osnovnih logičkih sklopova prikazana je na slici u nastavku.

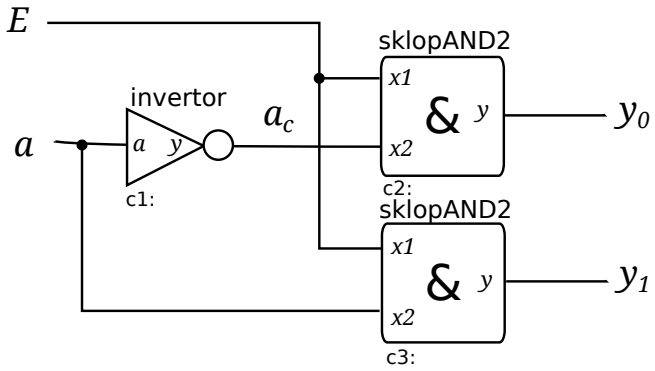


Temeljem ove slike možemo ponuditi model toka podataka.

Listing 3.9 : Model toka podataka binarnog dekodera 1/2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY dek21e IS PORT(
   a: IN STD_LOGIC;
   e: IN STD_LOGIC;
   y: OUT STD_LOGIC_VECTOR(0 TO 1));
END dek12e;

ARCHITECTURE arch OF dek12e IS
  SIGNAL ac: std_logic;
BEGIN
   ac <= not a;
   y(0) <= e and ac;
   y(1) <= e and a;
END arch;
```

I konačno, dajmo još i strukturni model ovog dekodera. Za tu svrhu najprije ćemo dodatno označiti shemu ostvarenja dekodera, kako je prikazano na slici u nastavku. Pretpostavit ćemo da su nam na raspolaganju komponenta invertor te dvoulazni sklopovi I (komponenta sklopAND2); drugim riječima, prije pisanja ovog modela, ti bi sklopovi već trebali biti definirani u trenutnom projektu.



Strukturni model ostvaren uporabom ovih komponenata prikazan je u nastavku.

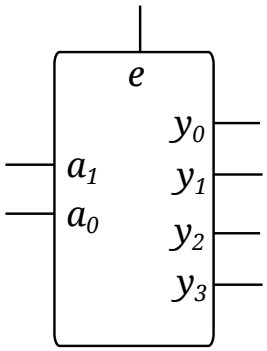
Listing 3.10 : Model toka podataka binarnog dekodera 1/2

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY dek21e IS PORT(
   a: IN STD_LOGIC;
   e: IN STD_LOGIC;
   y: OUT STD_LOGIC_VECTOR(0 TO 1));
END dek12e;

ARCHITECTURE arch OF dek12e IS
  SIGNAL ac: STD_LOGIC;
BEGIN
   c1: ENTITY work.invertor PORT MAP (a=>a, y=>ac);
   c2: ENTITY work.sklopAND2 PORT MAP (x1=>e, x2=>ac, y=>y(0));
   c3: ENTITY work.sklopAND2 PORT MAP (x1=>e, x2=>a,  y=>y(1));
END arch;
```

Pogledat ćemo još i binarni dekoder 2/4, i to najprije njegov ponašajni model a potom strukturni

model ostvaren na temelju dekoderskog stabla.

Sučelje binarnog dekodera 2/4 s ulazom za omogućavanje prikazano je na slici u nastavku.



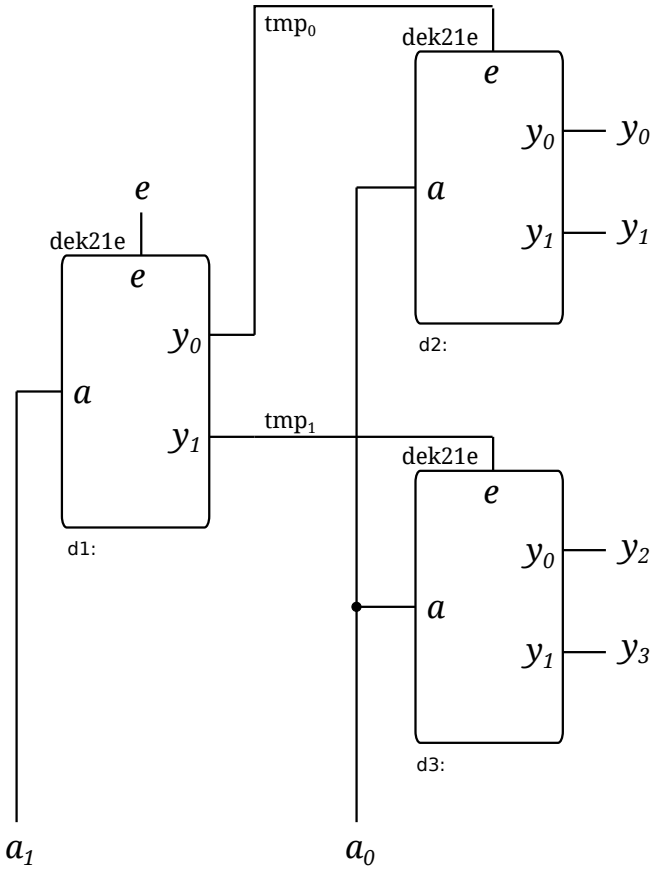
Kako sklop sada ima dvobini adresni ulaz, modelirat ćemo ga vektorom čiji je raspon indeksa padajući.

Ponašajni model binarnog dekodera 2/4 prikazan je u nastavku.

Listing 3.11 : Ponašajni model binarnog dekodera 2/4

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY dek24e IS PORT(
  a: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
  e: IN STD_LOGIC;
  y: OUT STD_LOGIC_VECTOR(0 TO 3)
);
END dek24e;

ARCHITECTURE arch OF dek24e IS
BEGIN
  process(a,e)
  begin
    if e='0' then
      y <= "0000";
    else
      case a is
        when "00" =>   y <= "1000";
        when "01" =>   y <= "0100";
        when "10" =>   y <= "0010";
        when "11" =>   y <= "0001";
        when others => y <= "0000";
      end case;
    end if;
  end process;

END arch;
```

Da bismo prikazali strukturni model, najprije je potrebno prikazati dekodersko stablo multipleksora 1/2 kojim ćemo ostvariti multipleksor 2/4. Ovo stablo prikazano je na slici u nastavku. Za sve korištene komponente iznad njih je napisan naziv komponente, ispod njih naziv primjerka komponente (ovaj puta, s obzirom da se radi o dekoderima, odlučili smo ih imenovati d1, d2 i d3), a za sve žice koje nisu niti ulazi niti izlazi odabrana su prikladna imena koja će postati interni signali.



Strukturni model napravljen temeljem ove slike prikazan je u nastavku.

Listing 3.12 : Strukturni model binarnog dekodera 2/4 ostvarenog stablom

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY dek24e IS PORT(
5     a: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
6     e: IN STD_LOGIC;
7     y: OUT STD_LOGIC_VECTOR(0 TO 3)
8  );
9  END dek24e;
10
11 ARCHITECTURE arch OF dek24e IS
12   SIGNAL tmp: std_logic_vector(0 to 1);
13 BEGIN
14    d1: ENTITY work.dek12e PORT MAP (a=>a(1), e=>e, y=>tmp);
15    d2: ENTITY work.dek12e PORT MAP (a=>a(0), e=>tmp(0), y=>y(0 to 1));
16    d3: ENTITY work.dek12e PORT MAP (a=>a(0), e=>tmp(1), y=>y(2 to 3));
17 END arch;
```

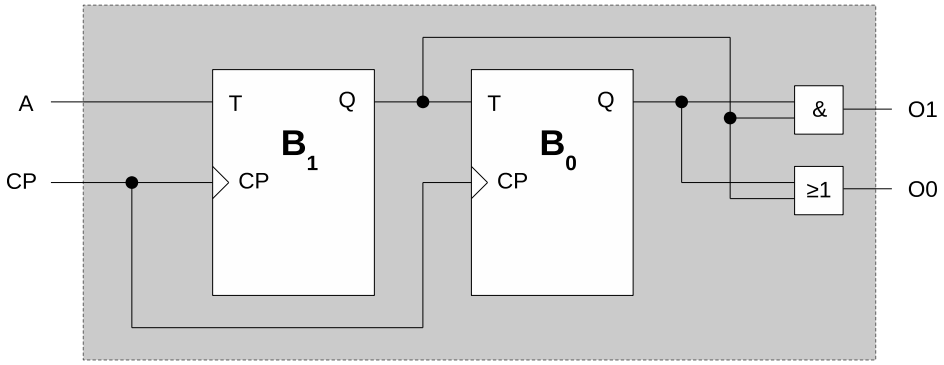
Poglavlje 4

Modeliranje bistabila

Kroz prethodna poglavlja dali smo kratak osvrt na modeliranje kombinacijskih sklopova jezikom VHDL. Sada ćemo razmotriti modeliranje sekvencijskog sklopovlja, i to krenuvši od njegovog osnovnog građevnog elementa: bistabila.

Kod kombinacijskog sklopovlja izlaz je uvijek i isključivo funkcija samo njegova ulaza. Tako, primjerice, razmotrimo li dvoulazni logički sklop I, pitamo li se što je na njegovu izlazu kada su ulazi 0 i 0, odgovor je uvijek 0; isto vrijedi i ako su ulazi 0 i 1 ili pak 1 i 0. Ako su oba ulaza 1, izlaz je također 1. Vrijednost izlaza nikada ne ovisi ni o čemu drugome osim o trenutnim ulazima sklopa.

Sekvencijsko sklopovlje je drugačije. Kod sekvencijskog sklopovlja izlaz sklopa može ovisiti o onome što je u tom trenutku na ulazima sklopa, ali može ovisiti i o stanju u kojem se sklop nalazi, a to je nešto što izvana promatraču nije vidljivo. Štoviše, izlaz sekvencijskog sklopa može ovisiti ne o onome što je u promatranom trenutku na njegovom ulazu, već o vrijednostima koje su bile na njegovu ulazu u nekom ranijem trenutku (primjerice, posljednji puta kada se dogodio padajući brid signala takta). Tako, primjerice, ako pogledamo bistabil tipa SR, i zapitamo se što je na njegovu izlazu Q u trenutku kada je na ulazima S i R vrijednost 0, ne možemo dati odgovor: odgovor će ovisiti o onome što je zapisano u sam bistabil, odnosno podatku koji bistabil u tom trenutku pamti. Ograničimo li se samo na bistabil tipa SR, tada je teško vidjeti kako to da stanje sustava može biti skriveno od promatrača, kada je kod bistabila njegovo stanje po definiciji ujedno i njegov izlaz. Pogledajmo stoga sklop koji je prikazan na slici u nastavku.



Promatrač izvana vidi sklop koji ima dva ulaza (ulaz za takt CP, te ulaz A) te dva izlaza (izlazi O1 i O0). Što će biti na izlazima tog sklopa direktno je ovisno o stanju sustava koje pamte bistabili B1 i B0. To stanje promatrač izvana ne može uvijek odrediti čak i ako zna što je na izlazima sklopa. Primjerice, razmotrite situaciju u kojoj je na izlazu O1 vrijednost 0, a na izlazu O0 vrijednost 1;

možete li jednoznačno zaključiti u kojem je stanju sustav (dakle, što je na izlazu bistabila B1 a što na izlazu bistabila B0)? Odgovor je, naravno, ne. Čak i poznavajući shemu samog sklopa (što vanjski promatrač obično ne zna) nemamo dovoljno informacija da bismo odgovorili na ovo pitanje. Moguće je da je uz zadane izlaze na izlazu bistabila B1 vrijednost 0 a na izlazu bistabila B0 vrijednost 1. Ali to nije nužno. Moguće je da je na izlazu bistabila B1 vrijednost 1 a na izlazu bistabila B0 vrijednost 0. U oba slučaja izlazi bi bili onakvi kakve smo zadali.

Razmotrimo još jedno pitanje: ako znamo da su izlazi O1=0 i O0=1, znamo li što će biti izlazi nakon sljedećeg impulsa takta (odnosno, gledajući shemu, korektnije bi bilo pitati nakon sljedećeg rastućeg brida signala takta)? I opet je odgovor *ne*. Evo samo kao primjer razmotrimo dvije hipoteze: ako je A=1, i ako je izlaz od bistabila B1 bio 0 a izlaz od bistabila B0 bio 1, tada će se nakon rastućeg brida izlazi O1 i O0 promijeniti tako da će oba biti 1. Ali ako je izlaz od bistabila B1 bio 1 a izlaz od bistabila B0 bio 0, izlazi se neće promijeniti. Ili, ako je izlaz od bistabila B1 bio 0 a izlaz od bistabila B0 bio 1 i ako je pri tome na A bila dovedena 0, izlazi se također neće promijeniti.

U općem slučaju, vidimo da je, dakle, ono što promatrač vidi izvana na izlazima sekvencijskog sklopa, odnosno slijed promjena koje se tu vide ovisno i o vrijednostima dovedenima na ulaze sekvencijskog sklopa (u našem slučaju ulaz A koji promatrač vidi) i o stanju u kojem se čitav sustav nalazi (stanje čine podatci zapamćeni u svim memorijskim elementima, a u našem slučaju čine ga izlazi svih bistabila od kojih je sustav izgrađen, i to je nešto što promatrač izvana ne vidi).

Kako su građevni elementi svakog sekvencijskog sklopa memorijski elementi, u ovom ćemo se poglavlju osvrnuti na modeliranje bistabila.

4.1 Model toka podataka osnovnog bistabila

Na predavanjima smo vidjeli da se uporabom kombinacijskih sklopova može izgraditi sklop koji više nije kombinacijski, ako se u sklop uvede petlja (odnosno povratna veza). Na taj smo način uporabom dvaju dvoulaznih logičkih sklopova NI ostvarili sklop koji ima dva stabila stanja: dobili smo osnovni bistabil tipa $\bar{S}\,\bar{R}$. Povezivanjem dvaju dvoulaznih logičkih sklopova NILI dobili bismo osnovni bistabil tipa $S\,R$. Model toka podataka osnovnog bistabila tipa $\bar{S}\,\bar{R}$ prikazan je u nastavku.

Listing 4.1 : Model toka podataka osnovnog bistabila tipa $\bar{S}\,\bar{R}$

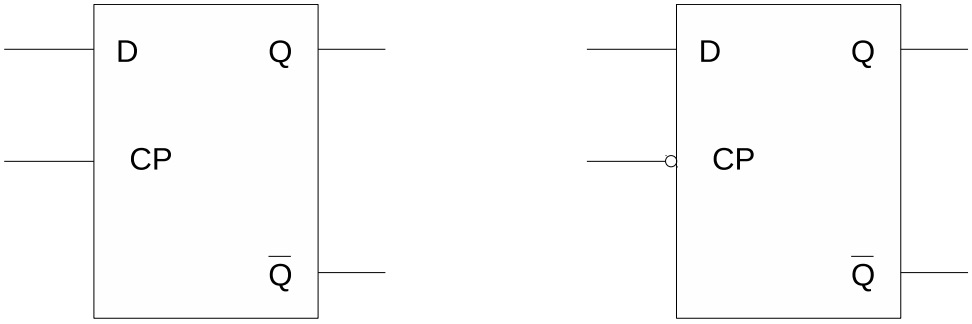
```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY SROsnovni IS PORT(
5    nS, nR: IN STD_LOGIC;
6    Q, nQ: OUT STD_LOGIC);
7  END SROsnovni;
8
9  ARCHITECTURE arch OF SROsnovni IS
10   SIGNAL intQ: std_logic := '0';
11   SIGNAL intQn: std_logic := '1';
12 BEGIN
13
14   intQ <= nS nand intQn;
15   intQn <= nR nand intQ;
16
17   Q <= intQ;
18   nQ <= intQn;
19
20 END arch;
```

Prethodni kôd za potrebe modeliranja osnovnog bistabila uvodi dva interna signala kojima je ostvarena povratna veza. Redak 14 definira način na koji se ulaz nS kombinira sa signalom intQn kako bi se definirala vrijednost signala intQ, a redak 15 definira način na koji se ulaz nR kombinira sa signalom intQ kako bi se definirala vrijednost signala intQn (čime je zatvorena povratna veza). Razlog iz kojeg na ovom mjestu koristimo interne signale, a ne direktno izlaze bistabila Q i nQ leži u ograničenju jezika VHDL koji ne dopušta da se vrijednost izlaza čita. Ako bismo sklopom NI direktno pokušali odrediti vrijednost izlaza, onda nam ta vrijednost ne bi bila dostupna u retku 15 za čitanje. Jedno rješenje je signale Q i Qn proglasiti ulazno-izlaznima (koristi se ključna riječ INOUT) ali time korisnicima ovog modela bistabila zapravo šaljemo krivu informaciju: Q i nQ jesu izlazi bistabila, a ulazima bismo ih proglasili samo zbog potreba modeliranja. Bolje rješenje je čuvati sučelje koje ima jasno razgraničene ulaze i izlaze, a opisani problem riješiti uporabom internih signala (koji po definiciji nisu niti ulazi niti izlazi pa im vrijednost možemo bez ograničenja i zapisivati i čitati).

Daljnjom razradom osnovnog bistabila mogli bismo doći do razinom upravljanih bistabila i potom do bridom upravljanih bistabila. Za ove sklopove, međutim, nećemo razmatrati niti modele toka podataka niti strukturne modele. Umjesto toga, pri modeliranju sekvencijskih sklopova preferirat ćemo ponašajne modele i njih ćemo razraditi u sljedećem podpoglavlju.

4.2 Ponašajni model bistabila

Osnovni bistabil po definiciji je asinkron sklop: on nema ulaz za takt i na pobudu reagira trenutno (uzevši u obzir, naravno, vrijeme kašnjenja logičkih sklopova od kojih je izgrađen). Dodamo li bistabilu ulaz za sinkronizacijski signal (signal takta), možemo izgraditi dvije porodice bistabila. Prvu porodicu čine *bistabili upravljani razinom signala takta*. Simboli bistabila tipa D iz ove porodice prikazani su na slici u nastavku.



Lijevi dio slike prikazuje simbol bistabila D koji svoj ulaz D "sluša" kad god (i sve dok je) na ulazu za signal takta CP visoka razina, odnosno logička jedinica. U engleskoj terminologiji, ovaj se sklop naziva *transparent latch*: tako dugo dok je CP=1, bistabil kao da ne postoji, odnosno sve što se dovodi na njegov ulaz uz vrijeme kašnjenja bistabila propušta se direktno na njegov izlaz. U trenutku kada CP padne na 0, ono što je u tom trenutku bilo upisano u bistabil zaključava se i održava na izlazu; promjene na ulazu D sada nemaju nikakvog efekta na izlaz. Na desnom dijelu slike prikazan je simbol sklopa koji svoj ulaz D "sluša" tako dugo dok je na ulazu za signal takta niska razina. Kod tog sklopa podatak se zaključava u trenutku kada se signal takta promijeni iz niske razine u visoku. Razlika između ta dva simbola je u kružiću koji je na desnom simbolu nacrtan uz ulaz za signal takta, i koji govori da ulaz djeluje niskom razinom.

Prilikom modeliranja ovog bistabila u arhitekturi ćemo koristiti jedan blok process i lokalnu varijablu koja će pamtiti stanje. To stanje mijenjat ćemo samo kada nam to signal takta dopusti, a izlaze

bistabila generirat ćemo temeljem tog stanja. Ponašajni model visokom razinom upravljanog bistabila D prikazan je u nastavku.

Listing 4.2 : Ponašajni model visokom razinom upravljanog bistabila D

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY DLatch IS PORT(
5    D, CP: IN STD_LOGIC;
6    Q, nQ: OUT STD_LOGIC);
7  END DLatch;
8
9  ARCHITECTURE arch OF DLatch IS
10 BEGIN
11
12   PROCESS(D, CP)
13     VARIABLE stanje: std_logic := '0';
14   BEGIN
15     IF CP='1' THEN
16       stanje := D;
17     END IF;
18     Q <= stanje;
19     nQ <= not stanje;
20   END PROCESS;
21
22 END arch;
```

Lista osjetljivosti bloka process sadrži i ulaze bistabila i signal takta. Naime, kad god se signal takta promijeni, izlaz bistabila bi se mogao promijeniti pa signal takta mora biti u listi osjetljivosti. Ali to mora i ulaz D: ako je signal takta u visokoj razini, svaka promjena ulaza D mora se prenijeti na izlaze bistabila pa i taj ulaz mora biti u listi osjetljivosti.

Za potrebe bloka process definirali smo jednu varijablu tipa std_logic koju smo nazvali stanje i koju smo za potrebe simuliranja inicijalizirali na vrijednost '0'. Inicijalizacija koju smo ovdje napisali događa se u trenutku t=0, odnosno samo na početku simulacije.

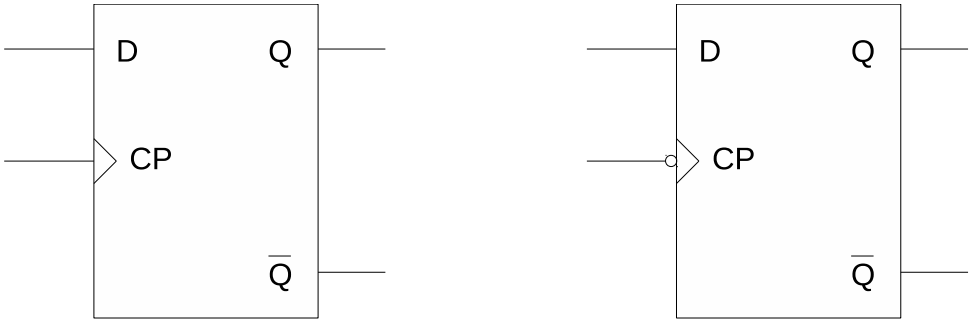
U retku 15 provjeravamo je li signal takta u visokoj razini; ako je, vrijednost koja je trenutno dovedena na ulaz D zapisuje se kao trenutno stanje (redak 16). Retci 18 i 19 temeljem trenutnog stanja definiraju što treba postaviti na izlaze bistabila; na izlaz Q zapisuje se stanje a na izlaz nQ njegov komplement. Želimo li za potrebe simuliranja modelirati bistabil s kašnjenjem, u ova dva retka dodali bismo AFTER 10 ns (ili slično).

Zašto ovaj opis rezultira sekvencijskim a ne kombinacijskim sklopom? Razmotrite redak 18. U njemu se izlaz Q definira temeljem onoga što je zapisano u varijabli stanje. A što je u tom trenutku zapisano u toj varijabli? Ako je kojim slučajem CP bio 1, onda je u trenutnom izvođenju bloka process u tu varijablu bila zapisana vrijedost s ulaza D. Međutim, ako je CP bio 0, u trenutnom izvođenju bloka process ništa nije bilo zapisano u tu varijablu - stoga je potrebno generirati memorijski element koji će zapamtiti što je prije toga bilo zapisano i koji će taj podatak staviti na raspolaganje za izračun izlaza Q. Na ovaj način definirali smo, dakle, memorijski element. Memorijski elementi nastat će uvijek kada u bloku process čitamo vrijednost neke varijable za koju postoji put kroz taj blok process kojim se vrijednost varijabli ne definira. U prethodnom kôdu dali smo lijep primjer toga: redak 18 čita vrijednost varijable, ali toj se varijabli u kôdu iznad vrijednost nekad definira (ako je CP=1) a nekad ne (ako je CP=0).

Želimo li modelirati bistabil čiji je simbol na slici 4.2 prikazan desno, u prethodnom kôdu morali bismo napraviti samo jednu izmjenu. Ispitivanje IF CP='1' u retku 15 zamijenili bismo ispitivanjem

IF CP='0'.

Druga porodica bistabila su *bridom upravljani bistabili*. To su bistabili koji svoje ulaze slušaju u vrlo kratkom vremenskom periodu (relativno u odnosu na trajanje signala takta): oni stanje mijenjaju ili u trenutku rastućeg brida signala takta, ili u trenutku padajućeg brida signala takta. Slika u nastavku prikazuje simbole bridom upravljanih (kažemo još *okidanih*) bistabila. Bridom okidani bistabili uz ulaz za signal takta imaju nacrtan trokutić. Na lijevom dijelu slike prikazan je simbol bistabila koji je okidan rastućim bridom signala takta (onim bridom koji vodi prema visokoj razini) a na desnom dijelu slike prikazan je simbol bistabila koji je okidan padajućim bridom signala takta (bridom koji vodi prema niskoj razini signala takta, na što nas upućuje kružić).



Ponašajni model ovakvog bistabila prikazan je u nastavku.

Listing 4.3 : Ponašajni model rastućim bridom upravljanog bistabila D

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY DFlipFlop IS PORT(
5    D, CP: IN STD_LOGIC;
6    Q, nQ: OUT STD_LOGIC);
7  END DFlipFlop;
8
9  ARCHITECTURE arch OF DFlipFlop IS
10 BEGIN
11
12   PROCESS(CP)
13     VARIABLE stanje: std_logic := '0';
14   BEGIN
15     IF rising_edge(CP) THEN
16       stanje := D;
17     END IF;
18     Q <= stanje;
19     nQ <= not stanje;
20   END PROCESS;
21
22 END arch;
```

U odnosu na prethodni model razinom upravljanog bistabila, ovdje ćete uočiti tri promjene.

1. Modelu smo dali novo ime (*DFlipFlop*).
2. Umjesto ispitivanja je li vrijednost signala CP jednaka 1 (ili 0), u retku 15 sada koristimo funkciju rising_edge i kao argument joj predajemo signal takta. Funkcija provjerava je li na predanom signalu u trenutku poziva funkcije nastupio rastući brid, i vraća istinu ako je. Slijedi

da ćemo vrijednost s ulaza D u varijablu stanje upisati samo ako je na signalu takta u tom trenutku rastući brid. Da smo htjeli modelirati padajućim bridom upravljani bistabil, koristili bismo funkciju falling_edge nad signalom takta.

3. Iz liste osjetljivosti izbacili smo ulaz D. Naime, nemoguće je da promjena na tom ulazu išta promjeni, osim ako se baš u tom trenutku nije dogodila promjena na signalu takta; a kako je signal takta već u listi osjetljivosti, ulaz D ne mora biti. Možemo zapamtiti i općenitije pravilo: kod sklopova koji su upravljani bridom signala takta, u listi osjetljivosti mora biti jedino signal takta (i eventualno drugi asinkroni ulazi, ako ih sklop ima); sinkroni ulazi ne moraju biti dio liste osjetljivosti.

Također, nije loše skrenuti još jednom pažnju na redak 16 u prethodnom kôdu: uočite da je to jednadžba promjene stanja bistabila D. Ovo nam direktno daje naputak kako modelirati i druge vrste bistabila.

Razmotrimo sada bistabil tipa D koji ima još dva dodatna (pomoćna) ulaza: ulaz S za postavljanje bistabila te ulaz R za njegovo resetiranje. Uobičajeno, ako dodatni ulazi djeluju, oni imaju prioritet nad standardnim ulazima modeliranog bistabila. Pogledajmo model jednog takvog bistabila koji je prikazan u nastavku. Što možemo reći o modeliranom bistabilu?

Listing 4.4 : Model bistabila D za analizu

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY DFlipFlop IS PORT(
5    D, S, R, CP: IN STD_LOGIC;
6    Q, nQ: OUT STD_LOGIC);
7  END DFlipFlop;
8
9  ARCHITECTURE arch OF DFlipFlop IS
10 BEGIN
11
12   PROCESS(CP,S,R)
13     VARIABLE stanje: std_logic := '0';
14   BEGIN
15     IF R='0' THEN
16       stanje := '0';
17     ELSIF S='0' THEN
18       stanje := '1';
19     ELSIF falling_edge(CP) THEN
20       stanje := D;
21     END IF;
22     Q <= stanje;
23     nQ <= not stanje;
24   END PROCESS;
25
26 END arch;
```

Promotrite građu bloka process i način na koji je kôd napisan. U retku 15 provjerava se je li ulaz R postavljen na vrijednost 0, i ako je, u stanje se upisuje 0. Uočite, ništa drugo se ne provjerava (posebice ne signal takta); možemo reći i ovako: čim se R postavi u 0, u stanje će se upisati 0. Ulaz R djeluje dakle asinkrono (neovisno o signalu takta), niskom razinom i resetira bistabil. Uočite također, tako dugo dok je R postavljen u 0, ostatak ispitivanja se ne izvodi (jer su grane označene s ELSIF). Slijedi da je ulaz R ulaz najvećeg prioriteta: njegovim aktiviranjem svi ostali ulazi (niti S, niti D, niti CP) više nisu bitni i nemaju utjecaja na stanje bistabila.

Tek ako R ne djeluje (nije 0), ispitivanje će propasti na sljedeću granu, i provjeriti je li možda S=0 (redak 17). Ako je, stanje se postavlja u 1 i prestaju daljnja ispitivanja. Uočite: i ulaz S je asinkron; on djeluje čim ga se aktivira (niskom razinom) ali je nižeg prioriteta od asinkronog ulaza R: da bi S mogao doći do izražaja, R ne smije djelovati.

Konačno, ako niti R niti S ne djeluju, ispitivanje propada na redak 19 koji ispituje je li nastupio padajući brid signala takta, i ako je, bistabil će promijeniti stanje u skladu s ulazom D. Ulaz D stoga je sinkroni ulaz: on stanje bistabila može promijeniti samo ako mu signal takta to dopusti.

Sumirajmo: prethodni model je model bistabila D okidanog padajućim bridom signala takta koji ima dva asinkrona ulaza, ulaz S koji postavlja stanje bistabila i ulaz R koji ga resetira. Oba asinkrona ulaza djeluju na nisku razinu pri čemu je ulaz R većeg prioriteta od ulaza S, pa ako oba djeluju, bistabil će se ponašati kao da djeluje samo ulaz R. Uočite i listu osjetljivosti bloka process (redak 12): u skladu s prethodno danim pravilom za bridom okidane sekvencijske sklopove, u listi osjetljivosti bloka process nalazi se signal takta i svi asinkroni ulazi, i ništa više od toga.

Pogledajmo još jedan primjer (kôd u nastavku).

Listing 4.5 : Model bistabila D za analizu

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY DFlipFlop IS PORT(
5    D, S, R, CP: IN STD_LOGIC;
6    Q, nQ: OUT STD_LOGIC);
7  END DFlipFlop;
8
9  ARCHITECTURE arch OF DFlipFlop IS
10 BEGIN
11
12   PROCESS(CP,S)
13     VARIABLE stanje: std_logic := '0';
14   BEGIN
15     IF S='1' THEN
16       stanje := '1';
17     ELSIF falling_edge(CP) THEN
18       IF R='1' THEN
19         stanje := '0';
20       ELSE
21         stanje := D;
22       END IF;
23     END IF;
24     Q <= stanje;
25     nQ <= not stanje;
26   END PROCESS;
27
28 END arch;
```

Vidite li razlike? Dodatni ulazi za postavljanje i brisanje sada djeluju na visoku razinu. U ovom modelu ulaz S je najprioritetniji ulaz, i on je jedini asinkroni ulaz. Ulaz R postao je sinkron (ispituje ga se samo kad to signal takta dozvoli, pa je ujedno uklonjen i iz liste osjetljivosti bloka process). Tek ako niti S niti R ne djeluju, provjerava se ulaz D.

Ovakvih različitih kôdova moguće je generirati mnoštvo i nije ideja da ih ovdje sve analiziramo. Umjesto toga, proučite način kako su ovi kôdovi napisani i kako smo ih analizirali. Pokušajte sami napisati modele rastućim bridom okidanog bistabila D sa sinkronim ulazima za postavljanje i brisanje koji djeluju visokom razinom i brisanje je prioritetnije. Pokušajte napisati sličan model ali bistabila

T. Napišite model padajućim bridom okidanog bistabila T koji ima dodatni asinkroni ulaz za brisanje koji djeluje visokom razinom.

Pogledajmo još kako bismo ponašajno modelirali bistabil JK s asinkronim ulazom za reset koji djeluje niskom razinom.

Listing 4.6 : Ponašajni model bistabila JK s asinkronim ulazom za reset

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY JKFlipFlop IS PORT(
5    J, K, R, CP: IN STD_LOGIC;
6    Q, nQ: OUT STD_LOGIC);
7  END JKFlipFlop;
8
9  ARCHITECTURE arch OF JKFlipFlop IS
10 BEGIN
11
12   PROCESS(CP,R)
13     VARIABLE stanje: std_logic := '0';
14   BEGIN
15     IF R='0' THEN
16       stanje := '0';
17     ELSIF falling_edge(CP) THEN
18       stanje := (stanje AND NOT K) OR (NOT stanje AND J);
19     END IF;
20     Q <= stanje;
21     nQ <= not stanje;
22   END PROCESS;
23
24 END arch;
```

Umjesto jednažbe promjene stanja u retku 18 mogli smo raspisati direktno i tablicu promjene stanja. Koristeći naredbu IF i ispitujući sve slučajeve (je li J=0, K=0, ili J=0, K=1, ili ...) i temeljem toga definirati kako se mijenja stanje. Međutim, jednadžba promjene stanja definira isto ponašanje ali uz bitno manje pisanja kôda, pa smo je zato iskoristili.

4.2.1 Pojam sinkronosti

Za kraj ovog poglavlja osvrnimo se još na nekoliko važnih definicija sinkronosti i asinkronosti.

- Kažemo da *neki signal djeluje sinkrono* ako djeluje samo kada mu to signal takta dopusti. Ako djeluje neovisno o signalu takta, kažemo da djeluje *asinkrono*.
- Kažemo da je *digitalni sklop sinkron* ako ima ulaz za takt koji mu određuje u kojem trenutku sluša svoje "glavne" ulaze. Takav sklop može imati dodatne asinkrone ulaze; sam sklop ćemo i dalje zvati sinkronim (uz pretpostavku da se asinkroni ulazi koriste u kontroliranim specifičnim uvjetima gdje ne stvaraju probleme). Primjerice, možemo imati sinkroni bistabil tipa D koji ima dodatni asinkroni ulaz za resetiranje koji se koristi samo po uključenju sklopa na napajanje kako bi se bistabil doveo u poznato početno stanje. Ako digitalni sklop nije sinkron, on je asinkron.
- Kažemo da je *digitalni sustav izgrađen od više sinkronih sklopova i sam sinkron* ako i samo ako svi sinkroni sklopovi dobivaju isti signal takta (koji je paralelno razveden do svih njih), tako da

svi takvi sklopovi u istim trenutcima reagiraju na pobudu i mijenjaju stanje. *Digitalni sustav izgrađen od više sinkronih sklopova je asinkron* ako nije sinkron.

Što možemo reći o samom signalu takta? Je li on sinkron ili asinkron? Ako ste pažljivo proučili prethodne definicije, onda bi trebalo biti jasno da je to pitanje besmisleno: sinkronost odnosno asinkronost signala, digitalnih sklopova i digitalnih sustava definira se temeljem razmatranja je li djelovanje istih uvjetovano (istim) signalom takta ili nije – signal takta ne djeluje sam na sebe niti u skladu sa sobom pa je pitanje besmisleno.

Kod bridom okidanih sinkronih sklopova važno je ponoviti da postoje uvjeti na vremenske odnose između trenutaka kada se smiju mijenjati sinkroni signali i između djelotvornog brida signala takta. Ta vremena za primjer bridom okidanih bistabila navedena su u nastavku i morate ih razumjeti.

- *Vrijeme postavljanja bistabila* (engl. *setup time*, t_{setup}) je minimalno vrijeme prije nailaska djelotvornog brida signala takta nakon kojeg se sve do nailaska djelotvornog brida signala takta vrijednost sinkronog ulaza više ne smije mijenjati (kako bi u trenutku nailaska djelotvornog brida sklop pouzdano očitao tu vrijednost).
- *Vrijeme zadržavanja bistabila* (engl. *hold time*, t_{hold}) je minimalno vrijeme nakon nailaska djelotvornog brida signala takta unutar kojeg se vrijednost sinkronog ulaza i dalje ne smije mijenjati (kako bi sklop pouzdano stigao odreagirati na tu pobudu).
- Uz pretpostavku da su oko djelotvornog brida svi sinkroni signali zadovoljili *vrijeme postavljanja bistabila* i *vrijeme zadržavanja bistabila*, sklop će odreagirati i promijeniti izlaze najkasnije nakon *vremena kašnjenja bistabila* (engl. *delay time*, t_{db}) mjereno od trenutka nailaska djelotvornog brida signala takta. Kod bistabila ovo se vrijeme prikladno još označava i $t_{\mathrm{cp-to-q}}$.

Poglavlje 5

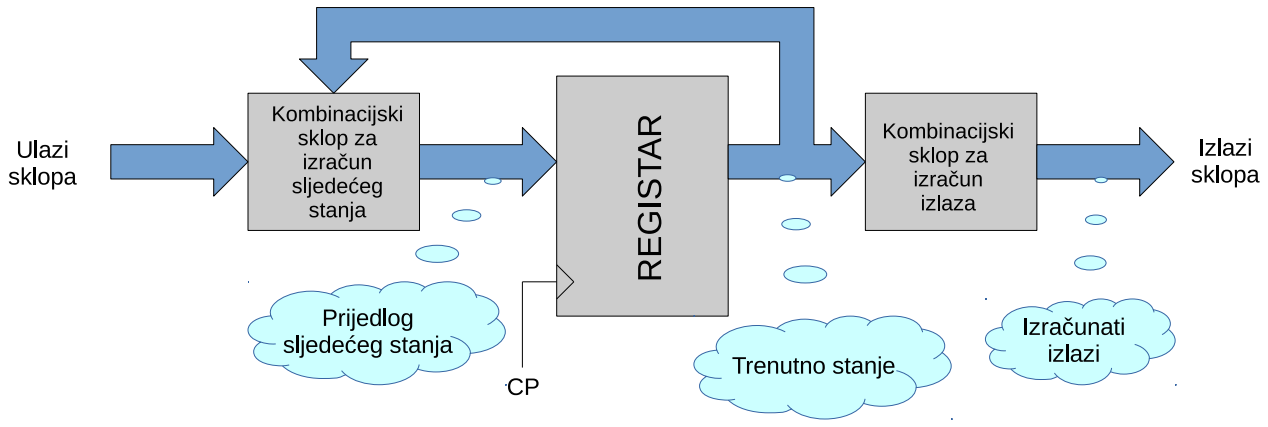
Modeliranje strojeva s konačnim brojem stanja

U prethodnom poglavlju osvnuli smo se na pojam sekvencijskog sklopa i neke njegove karakteristike: to je sklop koji za razliku od kombinacijskog sklopa ima *stanje* i kod njega izlaz nije samo funkcija ulaza (kažemo još i pobude) već i trenutnog stanja. Tako je moguće da uz istu pobudu (iste vrijednosti na ulazima) sklop u različitim trenutcima ima različite izlaze (jer će biti u različitim stanjima). U sklopovskom smislu, stanje sekvencijskog sklopa čine izlazi svih memorijskih elemenata. Za potrebe općenitog razmatranja građe ovakvih sklopova uvijek možemo zamisliti da se sklop sastoji od registra koji pamti trenutno stanje te dva kombinacijska sklopa: jednog koji određuje što sljedeće treba biti upisano u registar (drugim riječima: koje će biti sljedeće stanje) i drugog koji određuje što treba postaviti na izlaze sklopa.

U nastavku ćemo najprije pogledati Mooreov a potom i Mealyjev model stroja s konačnim brojem stanja (odnosno automata).

5.1 Mooreov stroj s konačnim brojem stanja

Općeniti oblik (shema) Mooreovog automata prikazan je na slici u nastavku.



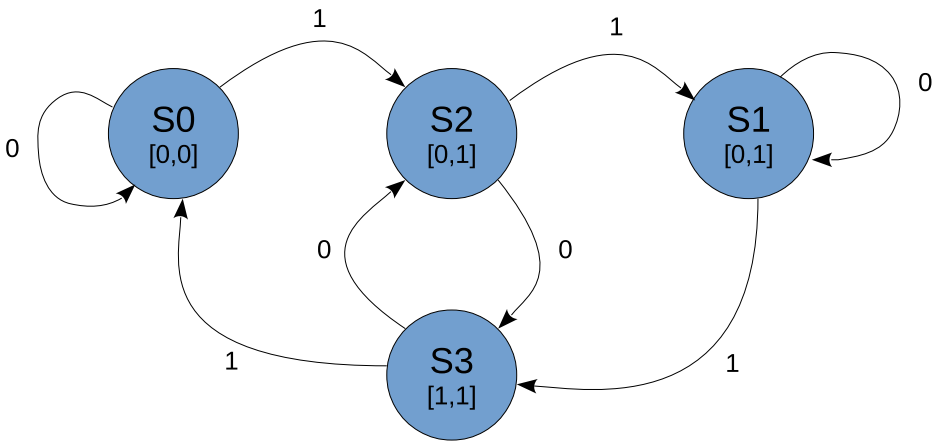
Središnji dio sklopa čini registar (memorija) koja dobiva prijedlog sljedećeg stanja i kada to signal takta dopusti (na slici kako je prikazano radilo bi se o rastućem bridu signala takta), taj prijedlog

upisuje kao trenutno stanje. Konceptualno, registar nije ništa drugo već potreban broj bistabila tipa D koji su svi spojeni na isti signal takta, pa kada im takt to dopusti, vrijednosti dovedene na ulaze zapamte kao novo stanje i prebace ga na izlaze.

Lijevi kombinacijski sklop na svoje ulaze dobiva ulaze automata koje postavlja korisnik te trenutno stanje automata. Temeljem tih podataka računa što treba biti sljedeće stanje i to generira na svojem izlazu. Taj izlaz dovodi se na ulaz registra koji će, kada mu to signal takta dopusti, taj prijedlog upisati kao novo "trenutno" stanje. Uočite, ako se ulazi automata mijenjaju asinkrono, prijedlog sljedećeg stanja također se može mijenjati ali neće imati nikakvog utjecaja na trenutno stanje sve do sljedećeg impulsa signala takta.

Desni kombinacijski sklop kao jedini ulaz dobiva trenutno stanje automata. Temeljem tog stanja računa vrijednosti koje je potrebno postaviti na izlaze automata. Kako kod ovog sklopa ne postoji direktna veza između ulaza automata i ovog izlaznog kombinacijskog sklopa, promjene ulaza ne utječu na izlaz: izlaz je isključivo funkcija od trenutnog stanja i može se promijeniti samo promjenom stanja automata.

Pogledajmo ovo na konkretnom primjeru. Neka je zadan automat dijagramom promjene stanja prikazanim na slici u nastavku.



Prikazani automat ima jedan ulaz (nazovimo ga A) te dva izlaza (nazovimo ih O1 i O0).

Zadaća prvog kombinacijskog sklopa jest na temelju trenutnog stanja i ulaza automata definirati što treba biti sljedeće stanje. Gledajući dani dijagram promjene stanja, ovaj bi kombinacijski sklop trebao sadržavati kôd poput sljedećeg:

Ako je trenutno stanje S0 i na ulazu je 0, sljedeće stanje bi moralo biti S0.
Ako je trenutno stanje S0 i na ulazu je 1, sljedeće stanje bi moralo biti S2.
...
Ako je trenutno stanje S3 i na ulazu je 0, sljedeće stanje bi moralo biti S2.
Ako je trenutno stanje S3 i na ulazu je 1, sljedeće stanje bi moralo biti S0.

Uočite da je ovo po karakteru doista čisti kombinacijski sklop: on uvijek za određenu kombinaciju stanja i ulaza generira isti prijedlog sljedećeg stanja.

Zadaća drugog kombinacijskog sklopa je generiranje izlaza. Kod Mooreovog automata izlaz je funkcija samo trenutnog stanja, pa ih zato i na slici pišemo direktno u kružićima (koji predstavljaju stanja).

Gledajući zadani dijagram promjene stanja, ovaj bi kombinacijski sklop trebao sadržavati kôd poput sljedećeg:

Ako je trenutno stanje S0, postavi na izlaze O1=0, O0=0.
Ako je trenutno stanje S1, postavi na izlaze O1=0, O0=1.
Ako je trenutno stanje S2, postavi na izlaze O1=0, O0=1.
Ako je trenutno stanje S3, postavi na izlaze O1=1, O0=1.

Konačno, zadaća registra je zapamtiti predloženo stanje kao trenutno, u trenutku kada mu to signal takta dozvoli. Za potrebe primjera pretpostavit ćemo da modeliramo Mooreov automat koji svoju pobudu sluša na rastući brid signala takta. Kako imamo 4 stanja, definirat ćemo sljedeću tablicu kodiranja stanja:

Stanje	Q3	Q2	Q1	Q0
S0	0	0	0	1
S1	0	0	1	0
S2	0	1	0	0
S3	1	0	0	0

Naš registar imat će četiri bistabila koja čuvaju stanje automata. Za kodiranje stanja iskoristili smo jednojedinični kôd: u stanju S_i samo je izlaz i-tog bistabila postavljen na vrijednost 1 dok su svi ostali bistabili resetirani.

VHDL kôd odnosno ponašajni model zadanog Mooreovog automata prikazan je u nastavku. Uočite tri bloka process: dva modeliraju kombinacijske sklopove a jedan modelira registar. Obratite pažnju i na njihove liste osjetljivosti koje bi sada morale biti jasne. Kako nam je informacija o trenutnom stanju potrebna i izvan bloka process koji generira memorijski element, ovom prilikom nismo za pamćenje stanja koristili varijablu (kao što je to bio slučaj kod modeliranja bistabila) već interni signal.

Listing 5.1 : Ponašajni model Mooreovog automata

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  ENTITY AutomatMoore IS PORT(
5    A, CP: IN STD_LOGIC;
6    O1, O0: OUT STD_LOGIC);
7  END AutomatMoore;
8
9  ARCHITECTURE arch OF AutomatMoore IS
10   -- tablica kodiranja stanja:
11   CONSTANT S0: std_logic_vector(3 downto 0) := "0001";
12   CONSTANT S1: std_logic_vector(3 downto 0) := "0010";
13   CONSTANT S2: std_logic_vector(3 downto 0) := "0100";
14   CONSTANT S3: std_logic_vector(3 downto 0) := "1000";
15   -- interni signali za pamćenje stanja i prijedloga sljedećeg stanja
16   SIGNAL stanje: std_logic_vector(3 downto 0) := S0;
17   SIGNAL sljed_stanje: std_logic_vector(3 downto 0);
18 BEGIN
19
20   -- kombinacijski sklop koji predlaže sljedeće stanje:
21   PROCESS(stanje, A)
22   BEGIN
23     CASE stanje IS
24       WHEN S0 => IF A=0 THEN sljed_stanje <= S0; ELSE sljed_stanje <= S2; END IF;
25       WHEN S1 => IF A=0 THEN sljed_stanje <= S1; ELSE sljed_stanje <= S3; END IF;
26       WHEN S2 => IF A=0 THEN sljed_stanje <= S3; ELSE sljed_stanje <= S1; END IF;
27       WHEN S3 => IF A=0 THEN sljed_stanje <= S2; ELSE sljed_stanje <= S0; END IF;
28       WHEN OTHERS => sljed_stanje <= S0;
29     END CASE;
30   END PROCESS;
31
```

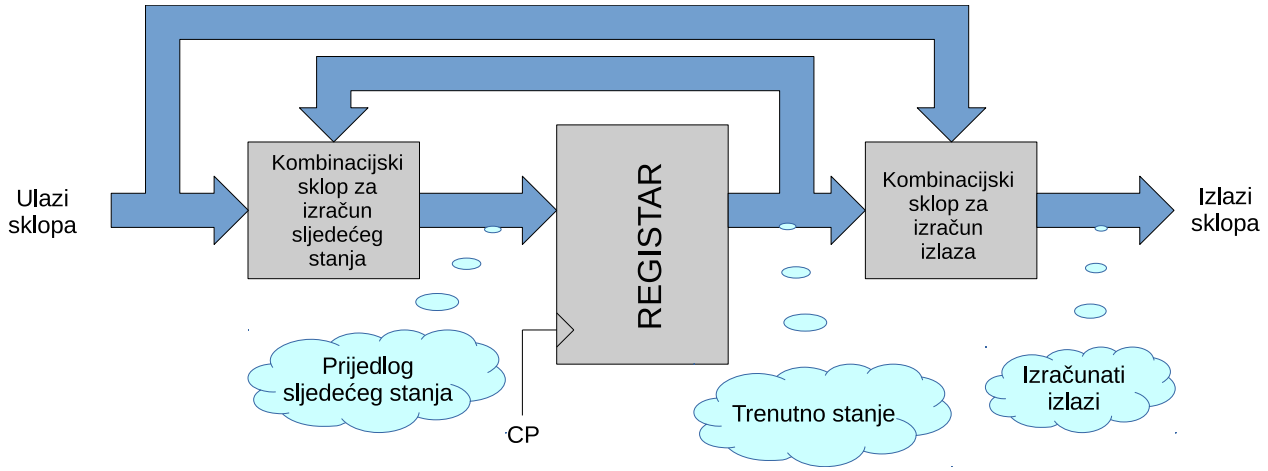
```
32   -- kombinacijski sklop koji odreduje izlaz automata:
33   PROCESS(stanje)
34   BEGIN
35      CASE stanje IS
36        WHEN S0 => O1 <= '0'; O0 <= '0';
37        WHEN S1 => O1 <= '0'; O0 <= '1';
38        WHEN S2 => O1 <= '0'; O0 <= '1';
39        WHEN S3 => O1 <= '1'; O0 <= '1';
40        WHEN OTHERS => O1 <= '0'; O0 <= '0';
41      END CASE;
42   END PROCESS;
43
44   -- model 4-bitnog registra:
45   PROCESS(CP)
46   BEGIN
47      IF rising_edge(CP) THEN
48         stanje <= sljed_stanje;
49      END IF;
50   END PROCESS;
51
52 END arch;
```

Blokove process koji modeliraju kombinacijske sklopove riješili smo uporabom naredbe case. Uočite da smo morali dodati stavku when others jer signal stanje može poprimiti puno više vrijednosti od pokrivene 4 (sjetite se, radi se o 4-bitnom signalu tipa std_logic gdje svaki bit može poprimiti jednu od 9 mogućih vrijednosti). Da bismo dobili kombinacijski sklop, za svaki mogući prolaz kroz ovaj blok moramo nešto zapisati u signal koji definiramo; u suprotnom ćemo i na ovim mjestima dobiti memorijske elemente, a to ne želimo. Stoga smo u kombinacijskom sklopu koji određuje sljedeće stanje definirali da je ono S0 ako se nađemo u nekom nepoznatom stanju odnosno da na izlaze u tom slučaju treba upisati 0.

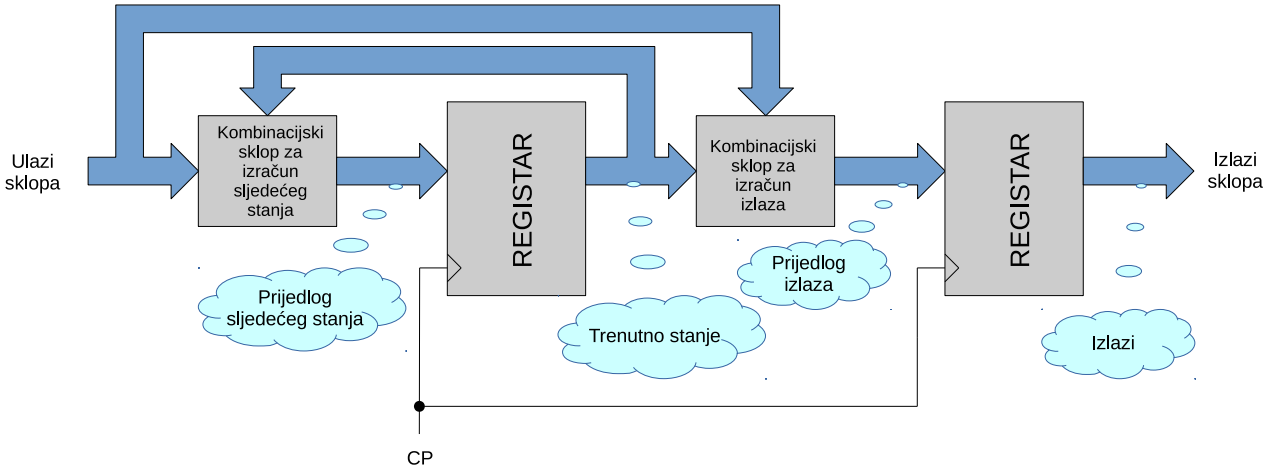
Za potrebe simulacije definirali smo da bi početna vrijednost signala stanje trebala biti S0 (redak 16).

5.2 Mealyjev stroj s konačnim brojem stanja

Opći oblik (shema) Mealyjevog automata prikazana je na slici u nastavku. Kod ove vrste automata trenutni ulazi dostupni su i u izlaznom kombinacijskom sklopu pa izlazi kod ove vrste automata mogu ovisiti i o trenutnom stanju i o trenutnim vrijednostima dovedenim na ulaze automata.

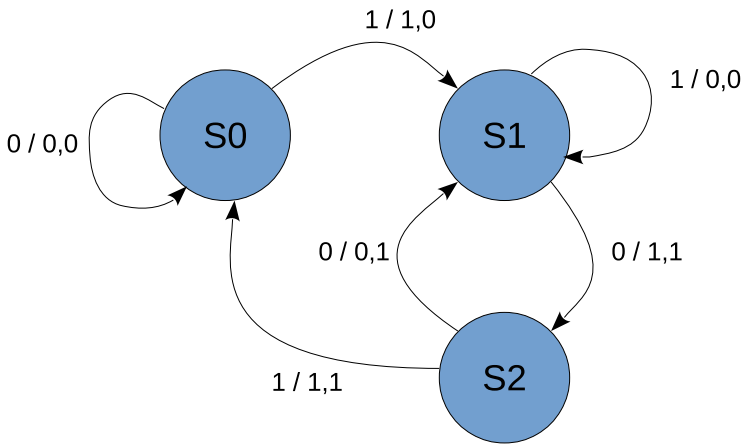


Zahvaljujući toj slobodi, ponekad nam je jednostavnije modelirati ponašanje određenih sklopova na ovaj način. Međutim, negativna strana ovog modela je da asinkrona promjena ulaza postaje direktno vidljiva i na izlazima: mijenjamo li ulaze dok je stanje fiksno i signal takta miruje, izlazi automata će se također mijenjati što može nepovoljno utjecati na rad sklopova kojima automat (preko svojih izlaza) upravlja. Stoga se umjesto ovakve direktne realizacije Mealyjev automat najčešće realizira uz uporabu dodatnog registra koji stvarne izlaze automata odvaja (i sinkronizira s taktom) od vrijednosti koje generira izlazni kombinacijski sklop. Modificirana shema koja prikazuje ovakvu izvedbu (koju ćemo i mi sami dalje koristiti) prikazana je na novoj slici u nastavku.



Ovakav model podrazumijeva postojanje dvaju registara (jedan koji pamti stanje, drugi koji pamti izlaze) te dvaju kombinacijskih sklopova (jedan koji predlaže sljedeće stanje a drugi koji predlaže vrijednost izlaza), što bi značilo pisanje četiriju blokova process. Međutim, u praksi se oba kombinacijska sklopa kod Mealyjevog automata modeliraju u jednom bloku process (jer imaju zajedničku listu osjetljivosti) te se oba registra modeliraju u jednom bloku process. To pak znači da model Mooreovog automata u praksi ima tri bloka process a model Mealyjevog automata dva bloka process.

Ponašajni opis Mealyjevog automata dat ćemo za automat čiji je dijagram promjene stanja prikazan na slici u nastavku.



Za kodiranje stanja opet ćemo koristiti jednojedinični kôd: kako imamo 3 stanja, kôdne riječi imat će tri bita. Tablica kodiranja stanja prikazana je u nastavku.

Stanje	Q2	Q1	Q0
S0	0	0	1
S1	0	1	0
S2	1	0	0

Ponašajni VHDL model za zadani automat dan je u nastavku.

Listing 5.2 : Ponašajni model Mealyjevog automata

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

ENTITY AutomatMealy IS PORT(
  A, CP: IN STD_LOGIC;
  O1, O0: OUT STD_LOGIC);
END AutomatMealy;

ARCHITECTURE arch OF AutomatMealy IS
  -- tablica kodiranja stanja:
  CONSTANT S0: std_logic_vector(2 downto 0) := "001";
  CONSTANT S1: std_logic_vector(2 downto 0) := "010";
  CONSTANT S2: std_logic_vector(2 downto 0) := "100";
  -- interni signali za pamćenje stanja, prijedloga sljedećeg stanja
  -- te prijedloga izlaza
  SIGNAL stanje: std_logic_vector(2 downto 0) := S0;
  SIGNAL sljed_stanje: std_logic_vector(2 downto 0);
  SIGNAL sljed_O1, sljed_O0: std_logic;
BEGIN

  -- kombinacijski sklop koji predlaže sljedeće stanje i izlaze:
  PROCESS(stanje, A)
  BEGIN
    CASE stanje IS
      WHEN S0 =>
        IF A=0 THEN sljed_stanje <= S0; sljed_O1 <= '0'; sljed_O0 <= '0';
               ELSE sljed_stanje <= S1; sljed_O1 <= '1'; sljed_O0 <= '0'; END IF;
      WHEN S1 =>
        IF A=0 THEN sljed_stanje <= S2; sljed_O1 <= '1'; sljed_O0 <= '1';
               ELSE sljed_stanje <= S1; sljed_O1 <= '0'; sljed_O0 <= '0'; END IF;
      WHEN S2 =>
        IF A=0 THEN sljed_stanje <= S1; sljed_O1 <= '0'; sljed_O0 <= '1';
               ELSE sljed_stanje <= S0; sljed_O1 <= '1'; sljed_O0 <= '1'; END IF;
      WHEN OTHERS => sljed_stanje <= S0; sljed_O1 <= '0'; sljed_O0 <= '0';
    END CASE;
  END PROCESS;

  -- model registra stanja i registra izlaza:
  PROCESS(CP)
  BEGIN
    IF rising_edge(CP) THEN
      stanje <= sljed_stanje;
      O1 <= sljed_O1;
      O0 <= sljed_O0;
    END IF;
  END PROCESS;

END arch;
```

Kada bismo čitavom automatu trebali dodati asinkroni ulaz za reset (čijim bi se djelovanjem automat prebacio u početno stanje S0), trebalo bi napraviti dvije korekcije (i kod Mooreovog automata, i kod Mealyjevog automata) koje su opisane u nastavku.

1. Promijeniti sučelje automata tako da se doda još jedan ulaz (primjerice, reset).

2. Modificirati blok process koji modelira registar tako da se omogući asinkrono brisanje (na isti način kao što smo to napravili kod bistabila). Konkretno, umjesto:

```
1     IF rising_edge(CP) THEN
2       stanje <= sljed_stanje;
3     END IF;
```

trebali bismo napisati nešto poput:

```
1     IF reset='1' THEN
2       stanje <= S0;
3     ELSIF rising_edge(CP) THEN
4       stanje <= sljed_stanje;
5     END IF;
```

Također, primjetimo da prilikom modeliranja automata nije nužno definirati tablicu kodiranja stanja. Moguće je umjesto toga definirati novi podatkovni tip i potom njega koristiti:

```
1 TYPE stanja IS (S0, S1, S2);
2 SIGNAL stanje: stanja;
```

Ovo pri simulaciji rada sklopa ne čini nikakvu razliku. Međutim, prilikom sinteze sklopa ovaj drugi pristup omogućava sintetizatoru da sam pokuša pronaći optimalnu tablicu kodiranja stanja uz koju će potrošiti minimalnu količinu kombinacijskog i sekvencijskog sklopovlja. Za potrebe laboratorijskih vježbi ovog kolegija, molim studente koji koriste VHDLLab okolinu za simuliranje sklopova da koriste prvi pristup (s eksplicitnom tablicom kodiranja stanja). Naime, u slučaju uporabe korisnički definiranih tipova podataka ti se signali neće vidjeti u rezultatu simulacije što je jedno od ograničenja simulacijskog alata koji koristimo i koji u ovom trenutku ne možemo jednostavno razriješiti.

Opisujete li automat u VHDL kako biste ga mogli sintetizirati, treba napomenuti da postavljanje inicijalnih (početnih) vrijednosti na način kako smo prethodno pokazali:

```
1 SIGNAL stanje: std_logic_vector(2 downto 0) := S0;
```

može uzrokovati nesintetizabilnost takvog opisa. Naime, ovisno o karakteristikama programirljivog sklopa takva inicijalizacija može i ne mora biti podržana. Stoga se u slučaju modeliranja automata koji će biti sintetizirani u sklopovlje preporuča da se automat izvede s eksplicitnim ulazom za resetiranje (kako je prethodno komentirano) te da se na taj ulaz po uključenju sustava dovede kratkotrajni impuls koji će obaviti reset i koji će nakon toga do kraja rada sklopa ostati neaktivan.

Konačno, ako automat ima mnoštvo ulaza, da bi bio korektno specificiran, nužno je da na dijagramu promjene stanja za svako stanje naznačimo što će automat napraviti za svaku moguću kombinaciju ulaza. Za automat koji ima jedan binarni ulaz to ne predstavlja nikakav problem: imat ćemo iz svakog stanja maksimalno po dva luka (jedan koji govori što napraviti ako je taj ulaz u nuli a drugi koji govori što napraviti ako je taj ulaz u jedinici). Ako automat ima veći broj ulaza (primjerice, 6 binarnih ulaza), tada ovo postaje nepraktično (i nepregledno); za 6 binarnih ulaza morali bismo se izjasniti što će automat napraviti za $2^6 = 64$ različite kombinacije ulaznih vrijednosti. Najčešće, pojedina stanja će predstavljati situacije u kojima automat čeka na neku interesantnu pobudu, a za sve ostalo radi nešto zajedničko. Kako bismo imali pregledniju sliku, za sve "interesantne" pobude (ma što to u pojedinom slučaju značilo) nacrtat ćemo lukove i navesti te pobude. Ono *za sve ostalo što nije eksplicitno pokriveno pobudama napisanim na lukovima* koristit ćemo luk na kojem ćemo napisati samo zvjezdicu na mjestu pobude. Pretpostavit ćemo da automat taj luk slijedi samo i isključivo ako je pobuda takva da nije pokrivena niti jednim drugim prijelazom iz tog stanja.