Fakultet elektrotehnike i računarstva
Zavod za elektroniku, mikroelektroniku,
računalne i inteligentne sustave

DIGITALNA LOGIKA

UPUTA ZA TREĆU LABORATORIJSKU VJEŽBU

dr. sc. Marko Čupić

Zagreb, 2013.

1. Uvod

Ova laboratorijska vježba prva je u kojoj ćete sklopove opisivati direktno uporabom jezika za specifikaciju sklopova VHDL. S jezikom VHDL već smo se upoznali na pripremnoj laboratorijskoj vježbi. U nastavku ove upute dano je još nekoliko napomena o jeziku VHDL a potom su dana i dva zadatka koja je potrebno riješiti prije dolaska u termin predaje laboratorijske vježbe.

Projektiranje nekog sklopa općenito se sastoji od niza koraka od kojih je svaki idući korak na nižoj razini apstrakcije, tj. bliži je samom sklopovlju. U nastavku je dan redoslijed po kojemu se obavlja projektiranje, a masnim slovima označeni su pojmovi specifični za modeliranje sklopova.

1. *Opis funkcionalnosti sklopa*

   U ovom koraku opisuje se funkcionalnost sklopa koju želimo postići. Sklop je moguće opisati na dva načina. Prvi način – sličan programiranju, tj. izradi programa – opisivanje je ponašanja sklopa. U tom slučaju govorimo o **funkcijskom (**odnosno **ponašajnom) modelu**. Druga mogućnost je da opišemo sastavne elemente od kojih se sklop sastoji i način međusobnog povezivanja elemenata (što smo radili na prve dvije laboratorijske vježbe). U ovom slučaju govorimo o **strukturnom modelu**. Elementi koje koristimo prilikom strukturnog opisivanja sklopa nazivaju se *komponente*. Komponente su tipično sklopovi nešto jednostavnije funkcionalnosti u odnosu na sklop koji želimo modelirati i čijim se povezivanjem postiže tražena funkcionalnost sklopa. Komponente su također opisane ili ponašajno ili strukturno. U slučaju strukturnog modela bitno je uočiti razliku između **komponente** i **primjerka** (**instance) komponente**. U nekom strukturnom modelu koristimo komponente (procesore, ALU, itd.), a kada upotrijebimo komponentu i povežemo je s ostatkom dizajna tada govorimo o **primjerku komponente**. Evo jednostavne ilustracije na primjeru računala. Ako govorimo o procesoru AMD Athlon 64 X2 6000+, 3000 MHz, govorimo o komponenti. No, na matičnoj ploči tih procesora može biti jedan ili više. Ako imamo više takvih procesora, svaki je spojen na druge vodiče na matičnoj ploči, pa govorimo o više primjeraka te komponente.

2. *Simulacija sklopa*

   Nakon što je sklop opisan potrebno je provjeriti ispravnost opisa što se obavlja simulacijom. Ulaz simulatora je niz ispitnih uzoraka koje simulator postavlja na ulaze sklopa koji se simulira te se računa odziv odnosno izlaz. Na osnovu izlaza može se zaključiti je li sklop ispravno opisan. Često je osim ispitnih uzoraka potrebno napraviti i cjelokupnu ispitnu okolinu. Npr. ako želimo testirati procesor potrebno mu je dodati RAM i ROM kako bi se on mogao testirati, te u tom slučaju govorimo o ispitnoj okolini koja se sastoji od RAM-a i ROM-a i ispitivane komponente – procesora. Okoline za razvoj sklopova uvijek omogućavaju izvođenje više vrsta simulacija. Najjednostavnija je *simulacija ponašajnog modela* (engl. *Behavioral model simulation*), kod koje se simulira isključivo logički rad sklopa i gdje se pretpostavlja da svi korišteni elementi rade beskonačno brzo (kašnjenja se zanemaruju). Najsloženija je *simulacija nakon postavljanja i povezivanja* u ciljnu tehnologiju (engl. *Post place and route simulation*) gdje se u obzir uzimaju svojstva realnih sklopova koji se koriste kako bi se ostvario opisani sklop (primjerice, koliko iznosi kašnjenje logičkih sklopova i sl.).

3. *Implementacija sklopa u ciljnoj tehnologiji*

   U ovom koraku opisani sklop prevodi se u željenu tehnologiju (FPGA, CPLD, ASIC, itd.). To se često izvodi automatizirano budući da se radi o dosta kompleksnom koraku. Podaci ovog koraka mogu se koristiti u prethodnom koraku kako bi se poboljšala točnost simulacije. Recimo, u ovom koraku saznajemo kašnjenja na sklopu koja se u koraku 2. nisu uzimala u obzir, ali koja mogu utjecati na dizajn.

4. *Izrada prototipa*

Usprkos simulaciji još nije sigurno hoće li sklop nakon proizvodnje ispravno raditi. Naime, prilikom simulacije nikada nije moguće uzeti baš sve u obzir. Stoga je potrebno proizvesti probne primjerke (prototipove) i testirati ih kako bi se utvrdila njihova ispravnost. U slučaju FPGA i CPLD uređaja (vrste programirljivih uređaja koje se obrađuju na kolegiju Digitalna logika) izrada probnih primjeraka svodi se na programiranje sklopova, dok se u slučaju ASIC-a tvornici šalju podaci dobiveni u koraku 3 na osnovi kojih tvornica izrađuje čipove.

Iako su prethodni koraci predstavljeni slijedno, u stvarnosti se radi o *iterativnom* procesu. Ove laboratorijske vježbe pokrivaju prva dva koraka, modeliranje sklopa i simulaciju. Za modeliranje se upotrebljava jezik VHDL (engl. *VHSIC Hardware Description Language*, *VHSIC – Very High Speed Integrated Circuit*).

2. Dodatno o jeziku VHDL

U ovoj laboratorijskoj vježbi po prvi se puta susrećemo s jezikom VHDL. Uvod u ovaj jezik napravljen je kroz predavanja i pripremnu laboratorijsku vježbu. Stoga se u nastavku pažnja posvećuje nekim specifičnostima. U jeziku VHDL uz signale vrijednosti se mogu pohranjivati i u varijable koje se koriste unutar bloka `process`.

2.1. Signali, varijable, blok process

U jeziku VHDL signali se definiraju ključnom riječi `SIGNAL` a varijable ključnom riječi `VARIABLE`. Razlika između signala i varijabli je u načinu uporabe: signale koristimo kao općenitu reprezentaciju vodiča; signalima se vrijednost može dodjeljivati uz definiranje kašnjenja. Primjerice, imamo li definiran signal `s1`, sljedeći izraz je legalan:

```
s1 <= a and not b after 25 ns;
```

i njime definiramo da će vrijednost tog signala uvijek odgovarati vrijednosti koju računa izraz `a and not b`, ali uz 25 nanosekundi kašnjenja; kad god se promijene bilo `a`, bilo `b`, signal `s1` poprimit će vrijednost `a and not b` tek 25 nanosekundi nakon te promjene.

Signali se ključnom riječi `SIGNAL` definiraju u okviru definicije arhitekture sklopa, ali prije navođenja ključne riječi `BEGIN`. Primjer koji ćemo proanalizirati dan je u nastavku.

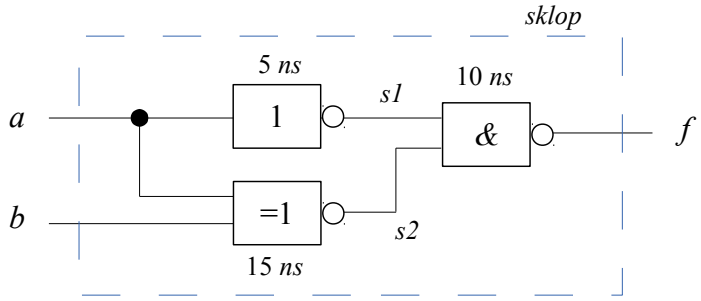
```
ENTITY sklop IS PORT (
  a, b: IN std_logic;
  f: OUT std_logic
);
END sklop;

ARCHITECTURE arch OF sklop IS
  SIGNAL s1: std_logic;
  SIGNAL s2: std_logic;
BEGIN

  s1 <= not a AFTER 5 ns;
  s2 <= not (a xor b) AFTER 15 ns;
  f <= s1 nand s2 AFTER 10 ns;

END arch;
```

Prikazani opis definira sklop koji je shematski prikazan u nastavku.



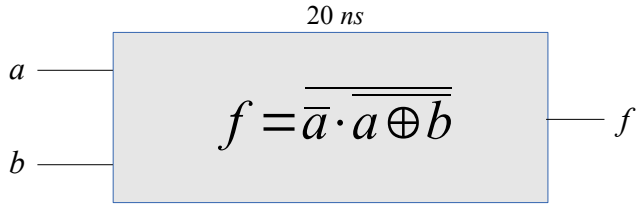
Pisanjem modela bez uporabe internih signala ne bismo mogli prikazati vremensku dinamiku koja je vezana uz različite puteve kojima ulazni signali prolaze kroz sklop. Primjerice, ako bismo arhitekturu sklopa modelirali samo jednim izrazom:

```
ARCHITECTURE arch OF sklop IS
BEGIN

  f <= not a nand not (a xor b) AFTER 20 ns;

END arch;
```

radili bismo zapravo model sklopa koji je shematski prikazan u nastavku:



i za koji definiramo jedno vrijeme kašnjenja.

Modeliramo li arhitekturu sklopa uporabom više izraza, kao što je bio prethodni primjer gdje smo najprije definirali vrijednost za `s1`, potom vrijednost za `s2` i konačno vrijednost za `f`, treba napomenuti da se u jeziku VHDL ti izrazi ne izvode redoslijedom kojim su napisani, odnosno ne smijemo ih promatrati kao program koji se sastoji od tri naredbe koje se izvode od prve do zadnje. Te naredbe predstavljaju komponente i način njihova povezivanja te ih (programski) moramo promatrati kao da se izvode paralelno. Stoga će prilikom simulacije simulator sva tri izraza promatrati kao tri naredbe koje se izvode paralelno odnosno efekt njihovog izvođenja bit će kao da se sve tri naredbe izvršavaju istovremeno i beskonačno često.

Ako nam je modeliranje sklopa lakše obaviti uporabom slijednog algoritma, možemo posegnuti za naredbom process koja omogućava definiranje isječka koda koji treba izvesti slijedno. Pri tome valja napomenuti da će simulator čitav navedeni isječak promatrati kao jednu veliku naredbu koju će izvoditi beskonačno brzo i paralelno sa svim ostalim naredbama. Primjerice, ako je arhitektura nekog sklopa opisana na sljedeći način:

```
ARCHITECTURE arch OF sklop IS
  SIGNAL s1, s2, s3, s4: std_logic;
BEGIN

  p1: process(...) begin ... end; -- definira vrijednost za s3
  p2: process(...) begin ... end; -- definira vrijednost za s4
  s1 <= not a AFTER 5 ns;
  s2 <= not (a xor b) AFTER 15 ns;
  f <= (s1 nand s2) or (s3 and s4) AFTER 10 ns;

END arch;
```

VHDL će je tretirati kao arhitekturu koja se sastoji od pet naredbi koje se izvode istovremeno; dvije od njih su isječci koda dani u okviru bloka `process` a tri su jednostavni izrazi pridruživanja.

Pogledajmo sada jedan blok process. Opći oblik bloka process prikazan je u nastavku:

```
naziv: process is
  deklaracije varijabli;
begin

  naredba1;
  naredba2;
  ...
  naredban;

end process;
```

Ovakav blok process VHDL tumači kao beskonačnu petlju te je stoga nužno u njegovu tijelu imati naredbu koja će zaustaviti izvođenje do nekog trenutka. Primjerice, jednostavnu naredbu pridruživanja koju smo već imali:

```
f <= not a nand not (a xor b) AFTER 20 ns;
```

možemo zamijeniti ekvivalentnom izvedbom opisanom sljedećim blokom process:

```
p1: process is
    begin

      f <= not a nand not (a xor b) AFTER 20 ns;
      wait on a, b;

    end process;
```

Prilikom pokretanja simulacije, automatski se pokreće izvođenje svih blokova process. Pokretanjem prethodnog bloka process temeljem trenutnih vrijednosti signala `a` i `b` izračunat će se vrijednost za `f` i potom će program stati na naredbi:

```
wait on a, b;
```

koja će izvođenje bloka process zamrznuti tako dugo dok se ne dogodi promjena bilo na signalu `a`, bilo na signalu `b`. Čim se dogodi bilo kakva promjena vrijednosti na jednom od ta dva signala, izvođenje naredbi tog bloka process se nastavlja: izvođenje doseže kraj bloka process i automatski se ponovno skače na njegovu prvu liniju i nastavlja izvođenje: računa se nova vrijednost za izlaz `f` nakon čega se izvođenje ponovno zamrzava na naredbi `wait`. Blok process napisan na prethodni način zapravo je beskonačna petlja pa ga u pseudokodu možemo zapisati ovako:

ponavljaj
  naredba1;
  naredba2;
  ...
  naredban;
zauvjek;

Bez da je jedna od naredbi u ovom bloku process naredba čekanja, blok process postao bi beskonačna petlja koja bi zaglavila rad simulatora. **Nemojte pokretati ovako napisane opise u VHDLLabu – poslužitelj će ih detektirati i završit ćete kod nas na razgovoru i s velikom količinom negativnih bodova.**

Eksperiment:

Otvorite VHDLLab i pronađite u vašim projektima bilo koji ispitni sklop (*testbench*). Napravite desni klik i iz iskočnog izbornika odaberite stavku "View VHDL". U VHDL opisu ispitnog sklopa arhitektura će se sastojati od dvije naredbe. Prva naredba bit će naredba kojom se stvara primjerak ispitivanog sklopa. Druga naredba će biti blok `process` koji strukturom direktno odgovara prethodno danom opisu. U tom bloku `process` neće se međutim koristiti oblik naredbe *wait on* već oblik *wait for* koji izvođenje bloka `process` zamrzava u trajanju koje se navodi u nastavku te oblik *wait* bez ikakvih argumenata koji trajno blokira izvođenje tog bloka `process`. Na taj način u bloku `process` se postiže generiranje "naklikanih" valnih oblika kojima se ispituje rad sklopa i nakon toga zamrzavanje bloka `process`.

Da ne biste morali razmišljati o pisanju naredbe `wait on`, VHDL nudi alternativnu sintaksu naredbe kojom se definira blok `process`: umjesto da kao zadnju naredbu bloka `process` napišete `wait on` i date popis signala, možete tu naredbu izostaviti a popis signala možete navesti u zagradama odmah nakon ključne riječi `process`. Blok `process` koji je napisan na ovaj način i koji je ekvivalentan prethodno danom bloku `process` prikazan je u nastavku.

```
p1: process(a, b) is
    begin

      f <= not a nand not (a xor b) AFTER 20 ns;

    end process;
```

Popis signala koji je naveden u zagradama nakon ključne riječi `process` nazivamo ***lista osjetljivosti***. Razlog za takvo ime sada bi trebao biti jasan: nakon prvog izvođenja svih napisanih naredbi u bloku `process`, isti će se zamrznuti i čekati na promjenu bilo kojeg od signala iz liste osjetljivosti nakon čega će napraviti novi prolaz kroz naredbe bloka `process` i potom se opet zamrznuti do sljedeće promjene bilo kojeg signala iz liste osjetljivosti.

Kao pomoć za izračun međurezultata u bloku process možete koristiti lokalne varijable. Evo primjera.

```
p1: process(a,b) is
      variable v1, v2: std_logic;
    begin

      v1 := not a;          -- ne mogu dodati AFTER 5 ns
      v2 := not (a xor b); -- ne mogu dodati AFTER 15 ns;
      f <= v1 nand v2 AFTER 20 ns;

    end process;
```

Varijablama se vrijednost dodjeljuje operatorom `:=` a ne operatorom <=. Također, varijablama se vrijednost ne može pridijeliti uz navođenje kašnjenja: desna strana izraza pridruživanja se računa i odmah upisuje u varijablu čije je ime navedeno s lijeve strane izraza pridruživanja. Stoga je upravo dani opis ekvivalentan onome koji smo prethodno dali jednom naredbom direktno u arhitekturi sklopa i koju ponavljamo u nastavku:

```
f <= not a nand not (a xor b) AFTER 20 ns;
```

Koristite li u bloku `process` naredbe koje dodjeljuju vrijednosti signalima i koje potom čitaju vrijednosti tih signala, moguće je napisati opis koji se neće simulirati onako kako biste od prve pomislili. Evo istog primjera u kojem umjesto lokalnih varijabli koristimo interne signale `s1` i `s2`.

```
p1: process(a,b) is
    begin

      s1 <= not a;          -- namjerno nemamo AFTER 5 ns
      s2 <= not (a xor b); -- namjerno nemamo AFTER 15 ns
      f <= s1 nand s2 AFTER 20 ns;

    end process;
```

Problem s ovim opisom jest tretman signala unutar bloka `process`: u bloku `process` vrijednosti se signalima ne dodjeljuju u trenutku izvođenja naredbe pridruživanja već u trenutku kada blok `process` po prvi puta zapne na nekoj naredbi čekanja. Pogledajmo konkretan primjer. Pretpostavimo da su u nekom trenutku vrijednosti signala `s1` i `s2` jednake `1`. Pretpostavimo da u tom trenutku dolazi do promjene na signalima `a` i `b`: `a` postaje `1` i `b` postaje `1`. Zbog te promjene izvođenje bloka `process` se "budi" iz naredbe `wait on` i simulator kreće s izvođenjem naredbi u bloku `process`. Najprije izvodi naredbu:

```
s1 <= not a;
```

Računa desnu stranu: `not a = not 1 = 0` i pamti da u `s1` mora upisati izračunati vrijednost `0`, ali to još ne radi. Potom prelazi na sljedeću naredbu:

```
s2 <= not (a xor b);
```

i računa desnu stranu: `not (a xor b) = not (1 xor 1) = not 0 = 1` i pamti da u `s2` mora upisati izračunatu vrijednost `1` ali to još ne radi. Konačno, prelazi se i na treću naredbu:

```
f <= s1 nand s2 AFTER 20 ns;
```

računa se desna strana: `s1 nand s2 = 1 nand 1 = 0` i pamti da uz kašnjenje od 20 nanosekundi mora kao `f` upisati vrijednost `0`. Uočite da, s obzirom da se u `s1` i `s2` još nisu upisale korektne vrijednosti, desna strana za izraz koji definira `f` izračunata je sa starim vrijednostima i stoga je pogrešna. Tek sada, kada bi blok `process` zapeo na naredbi `wait on`, obavljaju se zakazana pridruživanja: u `s1` se upisuje `0`, u `s2` se upisuje `1` i u `f` će se uz kašnjenje od 20 ns upisati `0`; blok `process` zamrzava se do sljedeće promjene bilo signala `a` bilo signala `b` a na `f` ostaje pogrešan rezultat (gledamo li samo logičke izraze, uz `a=1` i `b=1` izlaz `f` je trebao postati `1`). Ovaj se problem može riješiti tako da se proširi lista osjetljivosti bloka `process` svim internim signalima kojima se u tom bloku vrijednost najprije dodjeljuje a potom čita. Puno je čišće rješenje naprosto ne pisati takav kod odnosno za pohranu privremenih međurezultata koristiti varijable.

Imajući u vidu sve izloženo, vratite se sada na zadatke 16.1 do 16.4 iz zbirke i prođite kroz njih. Ako ćete imati bilo kakvih pitanja, dođite na konzultacije.

Deklaracija višebitnih signala slična je deklaraciji jednobitnih, ali se umjesto `std_logic` mora upotrijebiti `std_logic_vector` te se također mora definirati raspon. U općem slučaju deklaracija višebitnih signala obavlja se na sljedeći način:

```
SIGNAL imesignala: std_logic_vector (donja_granica TO gornja_granica);
```

odnosno sa:

```
SIGNAL imesignala: std_logic_vector (gornja_granica DOWNTO donja_granica);
```

Jednom kada se odluči za način deklaracije (`TO` ili `DOWNTO`), taj način dalje treba poštivati pri radu s tim signalom. Umjesto riječi `imesignala` treba staviti naziv signala koji se deklarira. U slučaju da je istovremeno potrebno deklarirati više signala imena se razdvajaju zarezom. Primjeri deklariranja višebitnih signala su:

```
-- Deklaracija internog 10-bitnog signala
SIGNAL bus10: std_logic_vector (9 downto 0);
```

```
-- Deklaracija tri trobitna
SIGNAL sigA, sigB, sigC: std_logic_vector (2 downto 0);

-- Deklaracija ulaznog 8-bitnog signala
inbus: IN std_logic_vector (0 to 8);
```

Dodjeljivanje vrijednosti višebitnim signalima obavlja se upotrebom standardnog operatora za dodjeljivanje vrijednosti (<=), a konstantne – višebitne – vrijednosti se moraju upisivati unutar dvostrukih navodnika ("). U dodjeljivaju je također moguće korištenje raspona. Slijedi niz primjera koji ilustrira sve do sada rečeno:

```
-- Inicijalizacija 3-bitnog signala kojemu je bit 2 bit najveće težine
sigA <= "011";

-- Prethodno dodjeljivanje je ekvivalentno sljedećem:
sigA (2 downto 0) <= "011";

-- Dodjeljivanje 3-bitnog signala sigB signalu sigA širine 6-bita
-- (sigA), pri čemu se ta 3 bita smještaju u bitove, 5, 4 i 3
sigA (5 downto 3) <= sigB;
```

U slučaju da je potrebno spajanje dva ili više signala u šire signale koristi se operator konkatenacije (`&`). Primjer:

```
-- Spajanje dva 3-bitna signala (sigB i sigC) u jedan 6-bitni signal
-- te dodjeljivanje te vrijednosti signalu sigA. sigB se smješta u
-- više bitove signala sigA, a sigC u niže bitove signala sigA.
sigA <= sigB & sigC;
```

Osim operatora konkatenacije za spajanje signala moguće je koristiti i zagradu, npr.:

```
-- Spajanje tri 3-bitna signala (sigB, sigC i sigD) u jedan 9-bitni signal
-- te dodjeljivanje te vrijednosti signalu sigA. sigB se smješta u
-- više bitove signala sigA, a sigD u niže bitove signala sigA.
sigA <= (sigB, sigC, sigD);
```

2.2. Izrazi IF i CASE

U ovom poglavlju opisani su izrazi `IF` i `CASE`. Ovi se izrazi smiju pisati isključivo unutar bloka `process`. Budući da su to izrazi koji se nalaze u gotovo svakom programskom jeziku danas u upotrebi nije puno prostora posvećeno pojašnjavanju njihova rada, ali su naglašene specifičnosti vezane uz VHDL.

Sintaksa IF izraza je:

```
IF uvjet THEN
    blok VHDL izraza
{ ELSIF uvjet THEN
    blok VHDL izraza }
[ ELSE
    blok VHDL izraza ]
END IF;
```

U prethodnom kodu `uvjet` je potrebno zamijeniti odgovarajućim VHDL izrazima provjere, dok je `blok VHDL izraza` niz izraza koji se simulira ako je odgovarajući uvjet točan, odnosno – u slučaju `ELSE` izraza – ako ni jedan uvjet nije točan. Unutar vitičastih zagrada označeni su izrazi koji se ne moraju pojaviti, mogu se pojaviti jednom ili proizvoljno puno puta. U uglatim zagradama naznačeni su izrazi koji se mogu pojaviti maksimalno jednom, tj. moguće ih je izostaviti. Prilikom konstruiranja uvjeta mogu se koristiti operatori prikazani u nastavku.

Operator	*Značenje*	*Primjer*
=	Provjera jednakosti	sigA = "1010", sigB = '1'
/=	Provjera nejednakosti	sigA /= sigB
and	Logička I operacija	sigA = '1' AND sigB = '0'
or	Logička ILI operacija	
not	Negacija	

Primjer izraza IF:

```
-- Ako je signal e aktivan (u stanju logičke jedinice) tada u
-- ovisnosti o vrijednosti signala sel propusti na izlaz signale
-- a odnosno b, u suprotnom na izlaz postavi logičku nulu.
IF (e = '1') THEN
    IF (sel = '1' AND e /= '0') THEN
        c <= a;
    ELSE
        c <= b;
    END IF;
ELSE
    c <= '0';
END IF;
```

U slučaju višestrukih provjera izraz IF može postati nepregledan. U tom slučaju koristi se izraz CASE. Sintaksa tog izraza je:

```
CASE izraz IS
    WHEN uvjet =>
        blok VHDL izraza
  { WHEN uvjet =>
        blok VHDL izraza }
  [ WHEN OTHERS =>
        blok VHDL izraza ]
END CASE;
```

Opet, kao i u slučaju IF naredbe, su sa uglatim i vitičastim zagradama označeni dijelovi koji se mogu ponavljati proizvoljan broj puta (vitičaste zagrade), odnosno maksimalno jednom (uglate zagrade). Umjesto izraza može stajati varijabla ili signal čiju vrijednost provjeravamo, dok se umjesto uvjeta postavljaju određene kombinacije vrijednosti koje želimo obraditi. Izraz WHEN OTHERS je dio koji se izvršava u slučaju da niti jedan od prethodnih uvjeta nije zadovoljio, slično kao ELSE u slučaju IF naredbe.

Kod IF naredbe potrebno je pripaziti da se **prilikom modeliranja kombinacijskih sklopova uključe sve moguće kombinacije**, jer u suprotnom se nakon sinteze može dobiti sekvencijski sklop, tj. sklop koji sadrži memorijske elemente. To konkretno znači da je prilikom pisanja modela kombinacijskog sklopa obavezno navesti ELSE izraz s odgovarajućim blokom naredbi, bez obzira što je taj dio u sintaksi naveden kao neobavezan.

Kod **CASE** naredbe je **obavezno obuhvatiti sve** moguće slučajeve koje izraz može poprimiti. To znači da se mora ili navesti potreban broj WHEN *uvjeta* (pa WHEN OTHERS ne pisati), ili navesti nedovoljan broj WHEN *uvjeta* i zatim kroz WHEN OTHERS pokriti preostale slučajeve. Evo primjera za razmisliti: definiran je dvobitni signal: SIGNAL sig: std_logic_vector(1 downto 0). Ukoliko želimo poduzeti odgovarajuću akciju za svaku moguću kombinaciju vrijednosti tog dvobitnog signala (dakle, pokriti je jednim WHEN *uvjet*), koliko ukupno WHEN *uvjet* izraza trebamo? (Hint: puno više od 4!)

2.3. Stukturni VHDL

Strukturni VHDL također smo pojasnili na pripremnoj vježbi. Ovdje su, kompletnosti radi, uz nove ponovljeni i neki već poznati elementi. Prilikom korištenja strukturnog VHDL-a potrebno je razumjeti razliku između komponente, tj. njene deklaracije, te stvaranja primjerka komponente – što će biti pojašnjeno u nastavku.

Strukturni VHDL za razliku od ponašajnog opisuje od kojih jednostavnijih komponenti se sklop sastoji i na koji način su te komponente međusobno povezane. Opisi pojedinih komponenti mogu biti ponašajni, ali također mogu biti i strukturni.

Nakon ključne riječi `ARCHITECTURE`, a prije ključne riječi `BEGIN`, moraju se deklarirati i svi interni signali. Interni signali služe za međusobno povezivanje komponenti. Kao pravilo može se uzeti da sve linije na nekoj shemi koje povezuju izlaze jedne komponente sa ulazom neke druge komponente treba deklarirati kao interni signal.

Komponente se upotrebljavaju u strukturnom opisu sklopa stvaranjem njihovih primjeraka (kažemo još *instanciranjem*). Stvaranje primjerka znači konkretnu upotrebu neke unaprijed deklarirane komponente. Prema tome, kada se govori o primjerku komponente govori se o konkretnom elementu u dizajnu, a kada se govori o komponenti misli se na sve elemente u dizajnu s istim sučeljem i ponašanjem. Stvaranje primjeraka i povezivanje komponenata obavlja se u bloku `ARCHITECTURE` između odgovarajućih rezerviranih riječi `BEGIN` i `END`. Sintaksa za stvaranje primjerka je sljedeća:

```
ime_primjerka: ENTITY work.naziv_sklopa
               PORT MAP (povezivanje ulaza i izlaza);
```

Umjesto `ime_primjerka` potrebno je navesti neko proizvoljno i jedinstveno ime koje će dobiti primjerak komponente. Umjesto `naziv_sklopa` potrebno je staviti ime komponente koja se upotrebljava. Konačno, u zagradama nakon rezerviranih riječi `PORT MAP` obavlja se povezivanje internih signala i ulaznih i izlaznih signala sklopa kojeg modeliramo s ulazima i izlazima komponente. Povezivanje se može provesti po poziciji ili može biti po imenu.

Detaljnije se to može pokazati na primjeru. Ako koristimo komponentu čije je sučelje:

```
ENTITY sklopI IS
    PORT (a, b: IN std_logic; z: OUT std_logic);
END ENTITY;
```

Tada primjerak te komponente možemo stvoriti na sljedeći način:

```
i1: ENTITY work.sklopI PORT MAP (s1, s2, s3);
```

Primjerak je nazvan `i1`. Signal `s1` povezuje se na `a` ulaz komponente, signal `s2` na `b` ulaz komponente, te signal `s3` na `z` izlaz komponente. U tom primjeru upotrebljen je implicitan način povezivanja, odnosno povezivanje putem pozicije. Drugim riječima, kako su navođeni interni signali tako su uparivani sa ulazima i izlazima komponente navedenima u deklaraciji komponente. Osim implicitnog, moguće je i povezivanje putem imena. Primjer ovakvog povezivanja dan je u nastavku.

```
i2: ENTITY work.sklopI PORT MAP (z => s3, b => s2, a => s1);
```

U tom primjeru postignut je isti efekt kao i kod pozicijskog povezivanja, tj. signal `s3` spojen je na izlaz `z` sklopa, signal `s2` spojen je na ulaz `b` sklopa a signal `s1` spojen je na ulaz `a` sklopa `sklopI`. U ovom slučaju nije bilo nužno navoditi signale po redoslijedu kako su navođeni prilikom deklaracije komponente.

Tijekom spajanja komponenti ponekad je potrebno kombiniranje signala i dodavanje invertora. Pri tome je potrebno obratiti pažnju na nekoliko pravila koja treba poštivati. Unutar izraza `PORT MAP` na mjestu gdje se očekuje ulazni jednobitni signal, dopušteno je staviti konstantu (`'0'` ili `'1'`), signal

(`s1`) ili komplement signala (`not s1`). Nije dopušteno obavljati složene Booleove funkcije (`and`, `or`, ...). Time je dopušteno npr:

```
i1: ENTITY work.sklopI PORT MAP ('0', NOT s2, s3);
```

ali sljedeći izraz nije dopušten:

```
i2: ENTITY work.sklopI PORT MAP (NOT s1 OR s2, s2, s3);
```

Na mjesta na kojima se očekuje višebitni signal, može se dovesti odgovarajući višebitni interni signal, njegov dio ili pak agregacija signala, uz ograničenja koja će biti navedena u nastavku. Evo primjera: neka je definiran sklop `sklopA` čije je sučelje sljedeće:

```
ENTITY sklopA IS PORT (
  ulaz1: std_logic_vector(3 downto 0);
  ulaz2: std_logic_vector(0 to 1);
  izlaz: OUT std_logic);
END sklopA;
```

Prilikom stvaranja primjerka komponente, na mjestu višebitnih ulaznih signala mogu doći agregacije konstanti, na oba načina (koristeći zagrade ili operator `&`). Npr. ispravno je:

```
i3: ENTITY work.sklopA PORT MAP ( ('1','0','1','1'), '0'&'1', y);
```

Međutim, agregacija unutar `PORT MAP` izraza ne smije sadržavati signale (niti operacije sa signalima). Dakle, sljedeće je pogrešno (ako je `a` jednobitni signal):

```
i3: ENTITY work.sklopI PORT MAP ( ('1',a,'1','1'), '0'&'1', y);
```

Ovakav izraz može (a i ne mora) preživjeti prevođenje, međutim, pokretanje simulacije neće uspjeti. Da bismo riješili ovaj problem, mogu se definirati dva pomoćna interna signala (4-bitni `int1` i 2-bitni `int2`), pa možemo pisati:

```
int1 <= (a, not a, not a or b, '1');
int2 <= not a & (not b or c);

i3: ENTITY work.sklopI PORT MAP ( int1, int2, y);
```

Uočite da je kod izvedbe agregacije operatorom `&` desni izraz stavljen u zagrade – u suprotnom se događa problem s prioritetima i prevođenje puca. Kada se agregacije pridjeljuju internim signalima (dakle nisu dio `PORT MAP` izraza), one mogu sadržavati i signale i složene izraze nad signalima.

Alternativno ovom pristupu, možemo definirati jedan veliki interni signal (6-bitni `interni`), i zatim njegove dijelove koristiti za potrebne višebitne ulaze:

```
interni <= (a, not a, not a or b, '1', not a, not b or c);

i3: sklopI PORT MAP ( interni(0 to 3), interni(4 to 5), y);
```

U tom slučaju na svaki od ulaza postavljamo jedan dio tog velikog internog signala: na prvi ulaz postavljamo prva četiri bita internog signala (odnosno bitove 0, 1, 2 i 3) a na drugi ulaz preostala dva bita (tj. bitove 4 i 5).

Važno: prilikom rada u sustavu VHDLLab2 postoji dodatno ograničenje koje morate poštivati. Iako jezik VHDL jednako tretira velika i mala slova (odnosno ne razlikuje ih), u sustavu VHDLLab2 **nužno je naziv sklopa uvijek pisati na isti način**, Primjerice, ako ste sklop nazvali SklopAnd (veliko početno S i A), u strukturnom VHDL opisu taj sklop morate koristiti uz Identično napisano ime: napišete li sklopAnd ili SKLOPand, to neće raditi!

3. Što je potrebno napraviti

Ova laboratorijska vježba sastoji se od dva zadatka i oba je potrebno riješiti u sustavu VHDLLab. Prvi zadatak je zajednički za sve studente dok se drugi sastoji od dvije inačice. Svaki student rješava samo jednu inačicu drugog zadatka, u skladu s tablicom koja je dana u nastavku.

Odabir inačice

Studenti čija je suma *predzadnje dvije* znamenke JMBAG-a neparna rade inačicu 1 drugog zadatka; studenti čija je suma *predzadnje dvije* znamenka JMBAG-a parna (uključuje i 0) rade inačicu 2 drugog zadatka. Dakle, ako je JMBAG=ABCDEFG**HI**J, razmatra se parnost sume H+I. Primjer:

JMBAG	*Inačica*
0036123**45**6	Inačica 1
0036234**46**7	Inačica 2
0036345**67**8	Inačica 1
0036678**00**1	Inačica 2

Ponovimo još jednom: svi studenti rade zadatak 1.

Sve dodijeljene zadatke trebate riješiti prije dolaska u Vaš termin predaje vježbe. Konkretno, u trenutku dolaska na vježbu morate imate:

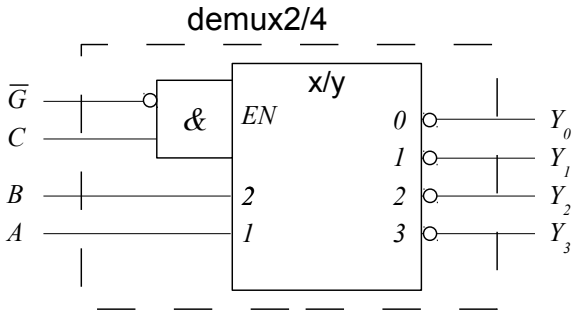
1. Popunjene tablice iz zadatka 1 te zapisane odgovore na postavljena pitanja;
2. traženi VHDL model te shemu iz zadatka 1 i pripadne ispitne sklopove;
3. VHDL modele tri tražena sklopa iz zadatka 2 i pripadne ispitne sklopove;
4. nacrtanu shemu multipleksorskog odnosno dekoderskog stabla (ovisno o inačici koju radite) na kojoj su jasno naznačene vrste sklopova i nazivi primjeraka sklopova (kako smo označavali na pripremnom satu) te gdje svaki simbol ima s unutrašnje strane napisane nazive ulaza i izlaza (opet, sjetite se kako smo to docrtali za pripremnom satu);
5. nacrtan vremenski dijagram koji prikazuje očekivani rezultat simulacije sva tri sklopa (pri čemu ne treba uzimati u obzir kašnjenja); te
6. tablicu kombinacija zadane funkcije f.

4. Zadatak 1

U okviru prvog zadatka modelirat ćemo dekoder/demultipeksor 3/8 na sličan način na koji je ostvaren sklop TTL 74155 opisan na predavanjima. Detaljnija specifikacija ovog sklopa u izvedbi tvrtke Texas Instruments dostupna je na adresi:

http://pdf1.alldatasheet.com/datasheet-pdf/view/27418/TI/74155.html

Sve oznake koje ćemo koristiti u nastavku bit će usklađene s oznakama u toj specifikaciji. Krenut ćemo s modeliranjem jedne polovice tog sklopa koju možemo simbolički prikazati kao što je to napravljeno na slici u nastavku.



Sklop se ponaša kao dekoder 2/4 na čiji je ulaz za omogućavanje (EN) dovedeno $\overline{G} \cdot C$: rad sklopa bit će omogućen samo ako je taj umnožak jednak 1. Oznaka $\overline{G}$ u sučelju sklopa govori nam da taj ulaz djeluje s logičkom nulom dok oznaka C simbolizira uobičajeno djelovanje logičkom jedinicom. Stoga će sklop biti omogućen samo kada je istovremeno na ulaz $\overline{G}$ dovedena logička 0 i na ulaz C dovedena logička 1. Ulazi B i A predstavljaju adresne ulaze pri čemu je ulaz B adresni ulaz veće težine. Konačno, izlazi dekodera su Y_0 do Y_3 i kada su aktivni, na njima je niska razina (to je vidljivo iz kružića koji simboliziraju invertore s desne strane simbola sklopa). Rad sklopa stoga možemo opisati sljedećom sažetom tablicom.

Ulazi				Izlazi			
Adresa		Aktivno	Podatak	Y_0	Y_1	Y_2	Y_3
B	A	$\overline{G}$	C				
x	x	1	x	1	1	1	1
0	0	0	1	0	1	1	1
0	1	0	1	1	0	1	1
1	0	0	1	1	1	0	1
1	1	0	1	1	1	1	0
x	x	x	0	1	1	1	1

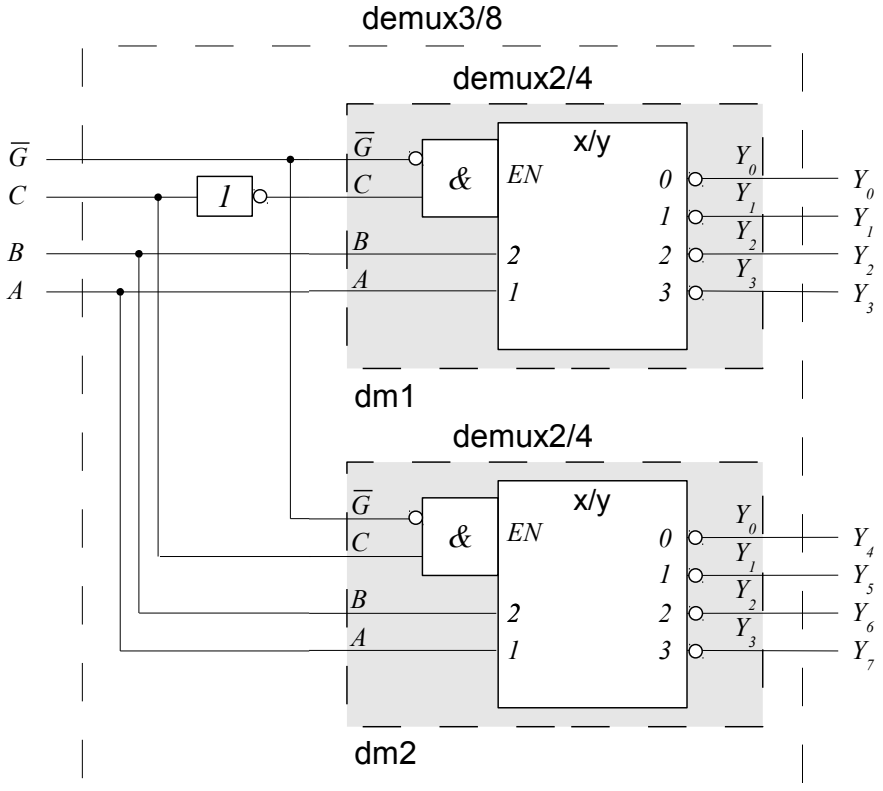
Ovaj sklop možemo istovremeno gledati i kao dekoder 2/4 i kao demultipleksor 2/4. Spojimo li fiksno na ulaz $\overline{G}$ logičku 0 i na ulaz C logičku 1, imamo dekoder 2/4 s niskoaktivnim izlazima. Takav dekoder bit će onemogućen bilo kada je na ulazu $\overline{G}$ logička 1 ili kada je na ulaz C logička 0. Spojimo li fiksno samo na ulaz $\overline{G}$ logičku 0, pažljivijom analizom prethodne tablice uočit ćemo da ovaj sklop na izlaz koji je odabran adresom BA propušta upravo $\overline{C}$: drugim

riječima, na odabranom izlazu sklop rekonstruira komplement podatka dovedenog na ulaz C -- stoga se uz takav način spajanja sklop ponaša kao demultipleksor 2/4 gdje je $\overline{G}$ njegov ulaz za omogućavanje. Stoga prethodnu tablicu možemo prikazati i na sljedeći način.

Ulazi				Izlazi			
Adresa		Aktivno	Podatak	Y_0	Y_1	Y_2	Y_3
B	A	$\overline{G}$	C				
x	x	1	x	1	1	1	1
0	0	0	x	$\overline{C}$	1	1	1
0	1	0	x	1	$\overline{C}$	1	1
1	0	0	x	1	1	$\overline{C}$	1
1	1	0	x	1	1	1	$\overline{C}$

Ovaj sklop nazovite `demux24` i za njega napišite ponašajni VHDL opis. Potom ispitajte rad tog sklopa prikladnim ispitnim sklopom.

Vaš sljedeći zadatak je upotrijebiti dva primjerka ovog sklopa kako biste izgradili dekoder/demultipleksor 3/8. Shema ovog sklopa prikazana je u nastavku.



Uz svaki primjerak sklopa naziv vrste sklopa napisan je iznad primjerka a naziv primjerka sklopa napisan je ispod primjerka (kao što smo radili na pripremnoj laboratorijskoj vježbi). Uočite kako su kod ovog sklopa ulazi $\overline{G}$ oba primjerka spojeni zajedno. S ulazom C postupa se drugačije: vanjski C dovodi se preko invertora na ulaz C prvog primjerka a direktno na ulaz C drugog primjerka. Razumijete li zašto je to napravljeno tako? Za ovaj sklop popunite njegovu tablicu istinitosti koja je dana u nastavku.

Ulazi				Izlazi							
Adresa			Aktivno	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
C	B	A	$\overline{G}$								
x	x	x	1								
0	0	0	0								
0	0	1	0								
0	1	0	0								
0	1	1	0								
1	0	0	0								
1	0	1	0								
1	1	0	0								
1	1	1	0								

Popunite i sažetu varijantu ove tablice:

Ulazi			Izlazi							
Adresa			Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
C	B	A								
0	0	0								
0	0	1								
0	1	0								
0	1	1								
1	0	0								
1	0	1								
1	1	0								
1	1	1								

Ovaj sklop nemojte modelirati jezikom VHDL – umjesto toga, postupite kako ste radili u 1. i 2. laboratorijskoj vježbi: **sklop definirajte crtanjem njegove sheme**. Sklop nazovite `demux38`. U uređivaču sheme ćete s desne strane u popisu komponenti pronaći prethodno definirani `demux24` koji ste opisali jezikom VHDL – dodajte dva primjerka, dodajte jedan invertor i nacrtajte ostatak sheme. Spremite sklop, provjerite može li se prevesti bez pogrešaka i potom napravite novi ispitni sklop kojim ćete ispitati njegov rad.

Jednom kada ste se uvjerili da sklop radi ispravno, otiđite mišem do njegovog naziva u projektu, otvorite iskočni izbornik (engl. *popup menu*) i pozovite stavku "View VHDL". Proučite dobiveni VHDL. Razumijete li sve što je pred Vama prikazano? Je li prikazani opis sklopa strukturni ili ponašajni? Po čemu to možete zaključiti?

Prilikom modeliranja digitalnih sklopova sada bi trebalo biti jasno da način modeliranja možete prilagoditi potrebama: gdje je prikladno, možete pisati VHDL modele, gdje je prikladno, možete crtati sheme – i sve uvijek možete kombinirati jedno s drugim; primjerice, mogli smo u projekt još dodati VHDL opis sklopa `sklop7` koji će koristiti `demux38` koji ste upravo definirali crtanjem sheme.

5. Zadatak 2

U nastavku su dane dvije inačice ovog zadatka. Vi morate napraviti samo jednu od te dvije inačice, u skladu s prethodno danim naputkom.

5.1. Inačica 1

U [2] počev od stranice 167 nalazi se nekoliko VHDL modela multipleksora 2/1. Jedan od tih modela je uz malu modifikaciju prepisan u nastavku (dodan je ulaz za omogućavanje te kašnjenje izlaza).

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

ENTITY mux21 IS PORT (
       E: IN Std_Logic;
       S: IN Std_Logic;
       D0: IN Std_Logic;
       D1: IN Std_Logic;
       Y: OUT Std_Logic);
end mux21;

ARCHITECTURE arch OF mux21 IS

BEGIN
  PROCESS (E, S, D0, D1)
  BEGIN
    IF E='1' THEN
      CASE S IS
        WHEN '0' => Y <= D0 AFTER 10 ns;
        WHEN '1' => Y <= D1 AFTER 10 ns;
        WHEN OTHERS => Y <= '0' AFTER 10 ns;
      END CASE;
    ELSE
      Y <= '0' AFTER 10 ns;
    END IF;
  END PROCESS;

END arch;
```

Prepišite ovaj model, napravite ispitnu okolinu i ispitajte njegov rad izvođenjem *simulacije ponašajnog modela*. Kako ovaj sklop ima 4 ulaza (E, S, D0 te D1), obavite ispitivanje za sve kombinacije ulaza (0000, 0001, ..., 1111 upravo navedenim radosljedom varijabli; svaku kombinaciju držite na ulazu 100 ns). Utvrdite koliko iznosi kašnjenje ulaza D0. Kako ćete to utvrditi?

Uporabom opisanog multipleksora 2/1 napišite strukturni model multipleksora 16/1, realiziranog kao multipleksorsko stablo. Sučelje tog sklopa treba sadržavati signale *D* (std_logic_vector), *S* (std_logic_vector), *E* (std_logic) te *f* (std_logic), tim poretkom. Svi vektori neka budu oblika (0 TO

n), gdje je *n* gornja granica. Ispitajte rad tog sklopa na nekoliko ispitnih uzoraka. Prilikom provjere rezultata simulacije obratite pažnju na način imenovanja signala! Možete li objasniti od kojih se dijelova sastoji ime signala?

Konačno, uporabom opisanog multipleksora 16/1 napišite strukturni model sklopa koji ostvaruje funkciju f(A,B,C,D)=f_n, te ispitajte rad tog sklopa za sve kombinacije ulaza (od 0000 do 1111; svaku kombinaciju zadržite 100 ns). Sučelje ovog sklopa treba sadržavati samo 5 signala: *A*, *B*, *C*, *D* te *f*, tim poretkom. Točna definicija funkcije f_n zadana je na kraju ovog dokumenta i ovisi o Vašem JMBAG-u.

5.2. Inačica 2

U [2] na stranicama 169-172 nalazi se nekoliko VHDL modela dekodera. Model dekodera 1/2 s ulazom za omogućavanje je uz malu modifikaciju prepisan u nastavku (dodano je kašnjenje izlaza).

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

ENTITY dek12e IS PORT (
       e : IN Std_Logic;
       a : IN Std_Logic;
       y0 : OUT Std_Logic;
       y1 : OUT Std_Logic);
end dek12e;

ARCHITECTURE arch OF dek12e IS

BEGIN

  y0 <= E AND NOT A AFTER 10 ns;
  y1 <= E AND A AFTER 10 ns;

END arch;
```

Prepišite ovaj model, napravite ispitnu okolinu i ispitajte njegov rad izvođenjem *simulacije ponašajnog modela*. Kako ovaj sklop ima 2 ulaza (E, A), obavite ispitivanje za sve kombinacije ulaza (00, 01, ..., 11 upravo navedenim redosljedom varijabli; svaku kombinaciju držite na ulazu 100 ns). Utvrdite koliko iznosi kašnjenje ulaza A. Kako ćete to utvrditi?

Uporabom opisanog dekodera 1/2 napišite strukturni model dekodera 4/16 s ulazom za omogućavanje, realiziranog kao dekodersko stablo. Sučelje tog sklopa treba sadržavati signale *E* (std_logic), *A* (std_logic_vector) te *Y* (std_logic_vector), tim poretkom. Svi vektori neka budu oblika (0 TO *n*), gdje je *n* gornja granica. Ispitajte rad tog sklopa na nekoliko ispitnih uzoraka. Prilikom provjere rezultata simulacije obratite pažnju na način imenovanja signala! Možete li objasniti od kojih se dijelova sastoji ime signala?

Konačno, uporabom opisanog dekodera 4/16 napišite strukturni model sklopa koji ostvaruje funkciju f(A,B,C,D)=f_n, te ispitajte rad tog sklopa za sve kombinacije ulaza (od 0000 do 1111; svaku kombinaciju držite na ulazu 100 ns). Sučelje ovog sklopa treba sadržavati samo 5 signala: *A*, *B*, *C*, *D* te *f*, tim poretkom. Točna definicija funkcije f_n zadana je na kraju ovog dokumenta i ovisi o Vašem JMBAG-u.

6. Primjer pitanja

U nastavku je dan primjer pitanja koja se mogu pojaviti na izlaznom testu. Navedena su samo pitanja bez ponuđenih odgovora:

- Koju funkciju obavlja demultipleksor?
- Koji su ulazi a koji izlazi sklopa `demux24` opisanog u ovoj vježbi?
- Koji su ulazi a koji izlazi sklopa `demux38` opisanog u ovoj vježbi?
- Koja je razlika između demultipleksora i dekodera?
- Koja je razlika između ponašajnog i strukturnog modela?
- Koja je razlika između varijabli i signala u VHDL-u?
- Kakav blok `process` se ne smije pisati?
- Koja su dva ekvivalentna oblika bloka `process`?
- Koji su sve oblici naredbe `WAIT`?
- Što označava pojam lista osjetljivosti?
- Gdje u VHDL kodu možemo pronaći listu osjetljivosti?
- Pojasnite vezu između odgovarajućeg oblika naredbe `WAIT` i liste osjetljivosti.
- Koju funkciju obavlja multipleksor?
- Koju funkciju obavlja dekoder?
- Kako se gradi multipleksorsko stablo?
- Kako se gradi dekodersko stablo?
- Kako se u VHDL-u označavaju komentari?
- Postoji li kašnjenje u simulaciji ponašajnog modela?
- Koje je puno ime nekog signala prilikom izvođenja simulacije?
- Koliko iznosi vrijeme kašnjenja sklopa?
- Koliko podatkovnih ulaza ima multipleksor koji ima 3 adresna ulaza?
- Kako se definiraju višebitni signali?
- Ispravna uporaba `CASE` naredbe prilikom modeliranja kombinacijskih sklopova.
- Način stvaranja primjeraka komponenti u strukturnom modeliranju?
- Agregacija signala.

- Povezivanje po imenu / povezivanje po poziciji.

- Kako operatori I, ILI i NE djeluju nad argumentima čije vrijednosti nisu samo `0` i `1`, već uključuju i vrijednost `U` (neinicijalizirano)? Primjerice, znamo da je `NOT 1 = 0`; a koliko iznosi `NOT U`? Prikažite djelovanja svih navedenih operatora tablično (također i za *Ex-ILI*)!

- Prilikom strukturnog opisa nekog sklopa, taj sklop opisujemo koristeći komponente. Svaka od tih komponenti može biti opisana opet strukturno ili ponašajno. Ako se tako spuštamo do dna, kako je opisana ona najjednostavnija komponenta – strukturno ili ponašajno? Objasnite!

7. Zadane funkcije za inačicu 1

Funkciju odabirete tako da uzmete predzadnje dvije znamenke JMBAG-a (ne sumu!) i potom primijenite operator modulo 20. Znači, ako je JMBAG=ABCDEFG**HI**J, birate N = (HI) % 20. Primjerice, student čiji je JMBAG 0012345678 računa 67%20=7.

N	***f=suma_minterma(...)***
0	5, 6, 8, 9, 10, 11
1	4, 6, 7, 8, 9, 14, 15
2	4, 5, 6, 7, 11, 13
3	3, 6, 7, 9, 12, 14, 15
4	3, 5, 7, 10, 13, 15
5	3, 4, 6, 8, 9, 14
6	3, 4, 5, 6, 7, 8, 11, 12, 14
7	2, 8, 9, 11, 12, 13, 15
8	2, 5, 7, 11, 12, 13, 15
9	2, 4, 6, 7, 11, 12, 13, 15
10	2, 4, 5, 7, 10, 11, 13, 14, 15
11	2, 3, 6, 7, 8, 10, 12, 13
12	2, 3, 5, 7, 8, 11, 12
13	2, 3, 4, 9, 12, 13, 14
14	2, 3, 4, 5, 7, 8, 9, 10, 13, 14
15	1, 6, 8, 9, 11, 12, 13
16	1, 5, 6, 7, 9, 10, 14, 15
17	1, 4, 6, 9, 11, 13
18	1, 4, 5, 7, 10, 14, 15
19	8, 10, 13, 14

8. Zadane funkcije za inačicu 2

Funkciju odabirete tako da uzmete predzadnje dvije znamenke JMBAG-a (ne sumu!) i potom primijenite operator modulo 20. Znači, ako je JMBAG=ABCDEFG**HI**J, birate N = (HI) % 20. Primjerice, student čiji je JMBAG 0012345678 računa 67%20=7.

N	*f=suma_minterma(...)*
0	0, 5, 8, 9, 12, 13
1	0, 4, 6, 7, 8, 9, 11, 14
2	0, 4, 5, 6, 9, 10, 11
3	0, 3, 7, 10, 11, 12, 13, 14
4	0, 3, 5, 6, 10, 11, 15
5	0, 3, 4, 7, 9, 11, 12, 13
6	0, 3, 4, 5, 7, 8, 10, 11, 13, 14
7	0, 2, 10, 11, 12, 13, 15
8	0, 2, 5, 7, 10, 11, 13, 14, 15
9	0, 2, 4, 8, 10, 11, 12, 15
10	0, 2, 4, 5, 7, 8, 10, 11, 13, 15
11	0, 2, 3, 7, 8, 9, 12
12	0, 2, 3, 5, 6, 7, 9, 10, 11, 12
13	0, 2, 3, 4, 7, 8, 9, 11, 12, 13
14	0, 2, 3, 4, 5, 6, 7, 8, 9, 13
15	0, 1, 6, 7, 8, 9, 12, 14
16	0, 1, 5, 6, 8, 9, 12, 13, 14
17	0, 1, 4, 6, 7, 8, 10, 14
18	0, 1, 4, 5, 8, 9, 13, 15
19	0, 6, 7, 8, 9, 12, 14

Literatura:

[1] Peruško, Glavinić: *Digitalni sustavi*. Školska knjiga, 2005.

[2] Čupić, *Digitalna elektronika i Digitalna logika. Zbirka riješenih zadataka*. Kigen, 2006.