Универзитет у Новом Саду

Факултет техничких наука

Рачунарске вежбе из предмета

Формалне методе пројектовања и верификације хардвера

Станиша Даутовић

Вук Врањковић

Станиша Даутовић

Вук Врањковић

Рачунарске вежбе из предмета

Формалне методе пројектовања и верификације хардвера

Факултет техничких наука
Катедра за електронику
Нови Сад, октобар 2014.

Садржај

Предговор

Овај помоћни уџбеник је настао током школске 2012/13 године, као пратећи материјал намењен студентима мастер студија на усмерењу Микропроцесорски системи и алгоритми, на Департману за енергетику, електронику и телекомуникације Факултета техничких наука у Новом Саду.

Предмет Формалне методе пројектовања и верификације хардвера, заједно са предметом Пројектовање сложених дигиталних система, у чијим оквирима студенти стичу знања о функционалној верификацији хардвера, треба да студентима пруже потребна знања у раду једног верификационог инжењера.

Иако је млађа и мање распрострањена од функционалне верификације дигиталног хардвера, формална верификација је чврсто утемељена и стандардизована област у дизајану и верификацији дигиталног хардвера. Данас постоје индустријски софтверски алати у којима је истовремено могуће спровести поступке формалне и функционалне верификације (попут IBM-овог RuleBase-а, Cadence®-овог Incisive®-а итд.). Процес потпуног прихватања формалне верификације у индустријским оквирима је комплетиран усвајањем IEEE стандарда бр. P1850, у којем је стандардизован језик за спецификацију особина PSL (енг. Property Specification Language). Језик PSL са једне стране израста над временским логикама CTL (енг. Computational Tree Logic) и LTL (енг. Linear Time Logic), и у том смислу поседује потребну математичку формалност. Са друге стране, PSL је намењен инжењерима и инжењерским применама, што се огледа кроз његову компатибилност са различитим језицима за опис хардвера, укључујући IEEE 1076 VHDL, IEEE 1364 Verilog, IEEE 1800 SystemVerilog и IEEE 1666 SystemC стандардизоване језике.

Претходна верзија помоћног уџбеника је објављена школске 2009/10, када је по први пут уведен предмет Формалне методе пројектовања и верификације хардвера. Главна разлика између две верзије помоћног уџбеника је промена софтверског алата у ком су развијене рачунарске вежбе на предметном курсу. У првој верзији, то је био програмски пакет NuSMV ver. 2.4.3, који је 2009, као и данас, слободно доступан за академско и комерцијално коришћење под условима LGPL ver.2.1 лиценце. Слободна доступност NuSMV-а је представљала главни разлог за избор овог симболичког проверивача модела. Стара верзија помоћног уџбеника ће и даље наставити да живи на веб-страници курса, за заинтересоване

читаоце који желе да се упуте у основе формалне верификације, али немају приступ комерцијалним лиценцама индустријских CAD EDA алата.

Потреба за другом верзијом овог помоћног уџбеника је настала са преласком са слободно доступног NuSMV-а, на индустријски EDA CAD алат *Incisive® Formal Verifier* (IFV) фирме *Cadence®*. Главни разлог за прелазак на нови софтверски алат је могућност верификације дизајна који су описани у неком од стандардних језика попут VHDL-а, Verilog-а и SystemVerilog-а, за разлику од NuSMV-а, који користи свој интерни језик. IFV је индустријски софтверски алат који се користи и ван академске заједнице. Оспособљавање студената мастер студија за коришћење *Cadence®* IFV-а им пружа додатна практична знања и вештине потребне у раду једног савременог верификационог инжењера.

Са друге стране, NuSMV у потпуности подржава све формалне тврдње које је могуће исказати у темпоралним логикама CTL и LTL, док у тренутној (октобар 2014) верзији *Cadence®* IFV-а, неки оператори логика CTL и LTL још увек нису подржани, што је утицало на делимичну реорганизацију текста. Да би се у најкраћим цртама набројале главне разлике између два софтверска алата, потребно је напоменути да се IEEE стандардизовани језик PSL састоји из три дела, тзв. PSL FL дела (скраћено од PSL Foundation Language), дела који подржава тзв. секвенцијално проширене регуларне изразе (енг. SEREs, од Sequential Extended Regular Expressions) и дела који подржава гранање карактеристично за CTL логику (у терминима PSL-а, тзв. OBE, од Opional Branching Extensions). NuSMV подржава у потпуности CTL и LTL логике, не подржава SEREs и подржава подскуп PSL-а. IFV не подржава CTL и LTL, подржава SEREs и подржава подскуп PSL-а (који није идентичан са оним који подржава NuSMV).

Аутори су посветили посебну пажњу конципирању примера који су ближи опису и верификацији дигиталних система на „нижим“ нивоима апстракције. Постојећи примери развијени у оквирима курсева из области формалне спецификације и верификације на различитим академским институцијама у свету, су често на вишим нивоима апстракције, у виду различитих бихевиоралних модела или протокола.

На крају, аутори изражавају захвалност колегама са Катедре за електронику на Департману за енергетику, електронику и телекомуникације, који су учествовали у стварању потребних предуслова за рађање једног оваквог курса, или који су својим корисним сугестијама унапредили квалитет његовог садржаја.

У Новом Саду, октобра 2014. Станиша Даутовић и Вук Врањковић

Кратко упутство за коришћење програма IFV

IFV (*Incisive*® *Formal Verifier*) је један од **EDA** (енг. *Electronic Design Automation*) алата компаније *Cadence*® из *Incisive* пакета програма намењених за функционалну и формалну верификацију дигиталних система. **IFV** је програм који служи само за формалну верификацију система, базирану на упитима који могу бити формулисани у два језика за спецификацију особина, **PSL** (енг. *Property Specification Language*) и **SVA** (енг. *System Verilog Assertions*).

У решеним задацима из ове збирке, коришћен је језик **PSL**. Хардверски модел система који је предмет верификације (енг. **DUT** или **DUV**, скраћено од *Design under Test/Verification*) је најчешће дат на **RTL** нивоу (енг. *Register Transfer Level*). Хардверски модел и спецификација особина које се верификују се у том случају могу сместити у исту датотеку. Ови модели су описани у једном од два језика за опис хардвера који су данас стандардно у употреби, *Verilog*-у или **VHDL**-у. Особине које се верификују се смештају у виду коментара у датотеке које садрже моделе система.

Алат **IFV** се покреће из терминала следећом командом:

```
>> ifv -f cmd_file.f +top+top_module +gui
```

Опција **`-f`** служи да се проследи име датотеке у којој се, између осталог, налази листа имена свих осталих датотека које се користе приликом моделовања и верификације. Опцијом **`+top+`** се прослеђује име модула од кога хијерархијски почиње процес верификације. Ова опција је неопходна уколико је тај модул моделован у језику **VHDL**, а опциона у језику *Verilog*. На крају, помоћу **`+gui`** опције, **IFV** програм може да се покрене са графичким корисничким окружењем.

Илустративни пример

Покретање алата **IFV** ће бити илустровано на следећем једноставном примеру. Да би се стартовао програм, неопходно је да постоје три обичне (енг. *plain*) текстуалне датотеке, са конвенционалним екстензијама *vhd* (или *v*), *f* и *tcl*, док су имена датотека произвољна:

1. simple_example.vhd

2. files.f

3. do.tcl

Садржаји три датотеке су приказани ниже.

Датотека simple_example.vhd:

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

entity simple_example is
    port (
        clk : in  std_logic;
        rst : in  std_logic;
        a   : in  std_logic;
        b   : in  std_logic;
        r0  : out std_logic;
        r1  : out std_logic;
        r2  : out std_logic);
end entity simple_example;

architecture rtl of simple_example is
    signal r0_d  : std_logic;
    signal r1_d  : std_logic;
    signal r2_d  : std_logic;
    signal sel_w : std_logic_vector(1 downto 0);
begin

    sel_w <= a & b;

    reg_p : process (clk) is
```

```
begin
    if rising_edge(clk) then
        if rst = '1' then
            r0 <= '0';
            r1 <= '0';
            r2 <= '0';
        else
            r0 <= r0_d;
            r1 <= r1_d;
            r2 <= r2_d;
        end if;
    end if;
end process reg_p;

r0_d <= a and b;

r1_p : process (sel_w) is
begin
    case sel_w is
        when "00"   => r1_d <= '0';
        when "01"   => r1_d <= '0';
        when "10"   => r1_d <= '0';
        when others => r1_d <= '1';
    end case;
end process r1_p;

r2_p : process (sel_w) is
begin
```

```
        case sel_w is
            when "00"   => r2_d <= '0';
            when "01"   => r2_d <= '1';
            when "10"   => r2_d <= '0';
            when others => r2_d <= '1';
        end case;
    end process r2_p;

    -- psl EQ_01 : assert always r0 = r1;
    -- psl EQ_02 : assert always r0 = r2;
end architecture rtl;
```

Датотека files.f:

```
simple_example.vhd
+tcl+do.tcl
```

Датотека do.tcl:

```
clock -add clk -initial 0
force rst 1
run 2
init -load -current
init -show
constraint -add -pin rst 0
prove
```

VHDL модел коришћен у примеру садржи три различите имплементације редне везе двоулазне логичке AND капије и флип-флопа. Један од три модела има грешку (погрешна имплементирана AND капија). У коментарима на крају *vhd* датотеке су специфициране и две особине које верификују логичку еквивалентност ових модела.

У датотеци *files.f* су прво излистана имена датотека које садрже описе свих хардверских модела који чине укупни дизајн система. У овом примеру, то је само дадотека под именом *simple_example.vhd*. Потом је у последњој линији

специфицирана **TCL** (енг. *Tool Command Language*) датотека коју ће **IVF** употребити чим се покрене, чија је функционалност описана ниже.

Датотека *do.tcl* садржи листу команди написаних у језику **TCL**. Прва команда одређује који је подразумевани синхронизациони сигнал система. У овом примеру то је сигнал **`clk`**. Наредне две команде служе да се систем доведе у иницијално стање. За овај пример довољно је поставити **`rst`** на '1' и сачекати један циклус. Команда **`run`** као параметар очекује број полупериода синхронизационог сигнала. Будући да је за ресетовање система у овом примеру потребан један циклус, команди run је прослеђен број 2. Линије 4. и 5. учитавају добијено иницијално стање након ресета и приказују га на терминалу. Линија 6. поставља ограничење да је током верификације сигнал **`rst`** постављен на '0'. Последња линија покреће верификацију система. Ова датотека се може користити као шаблон за даље примере. По потреби се само могу променити имена синхронизационог и ресет сигнала.

Након што су написане све датотеке, верификација се покреће из директоријума где се оне налазе. Команда за стартовање је следећа:

```
>> ifv -f files.f +top+simple_example
```

По завршетку верификације добија се следећи извештај из ког се може видети да је једна особина прошла тест, док друга није.

```
FormalVerifier> clock -add clk -initial 0

1 clock added.

FormalVerifier> force rst 1

FormalVerifier> run 2

Ran until 2 crank(s)

FormalVerifier> init -load -current

formalverifier: Current value of state variables loaded for
formal verification.

FormalVerifier> init -show


       r0                                              : b"0"
       r1                                              : b"0"
       r2                                              : b"0"
```

```
FormalVerifier> constraint -add -pin rst 0
1 pin constraint added.
FormalVerifier> prove
Modeling check mode:
  Vacuity check finished
Verification mode:
  :EQ_02 : Fail (1)
  :EQ_01 : Pass
Assertion Summary:
  Total                  :   2
  Pass                   :   1
  Fail                   :   1
  Not_Run                :   0
```

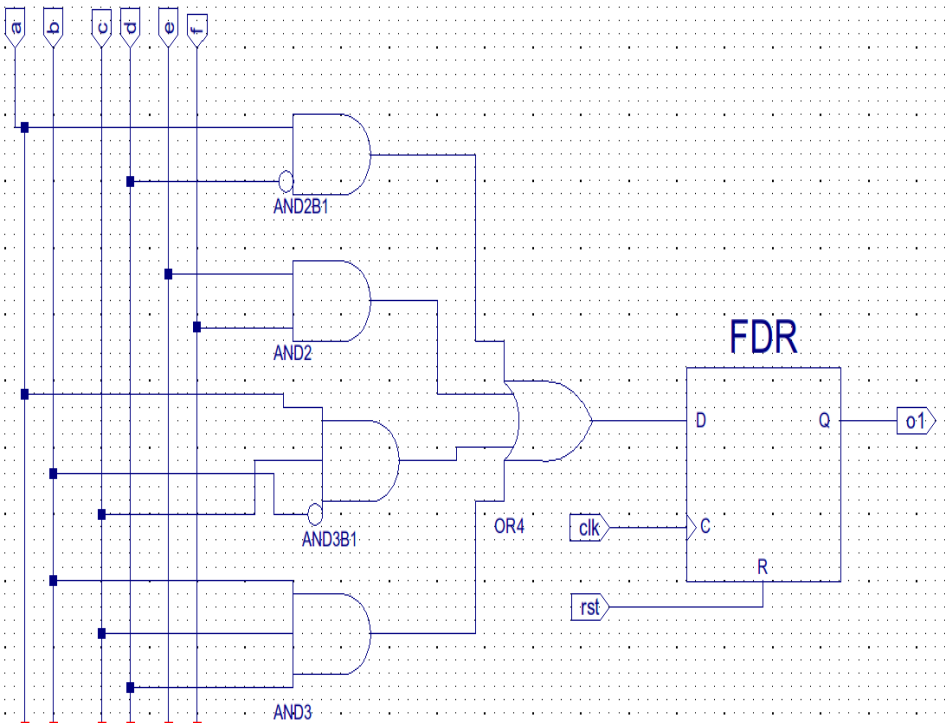
Из добијеног извештаја се може видети да је особина под именом **EQ_01** „прошла“ доказ (енг. *Pass*, тј. логички тачна), док је особина под именом **EQ_02** „пала“ (енг. *Fail*, тј. логички нетачна).

1. Провера логичке еквивалентности (LEC) комбинационих мрежа у standard-cell технологији

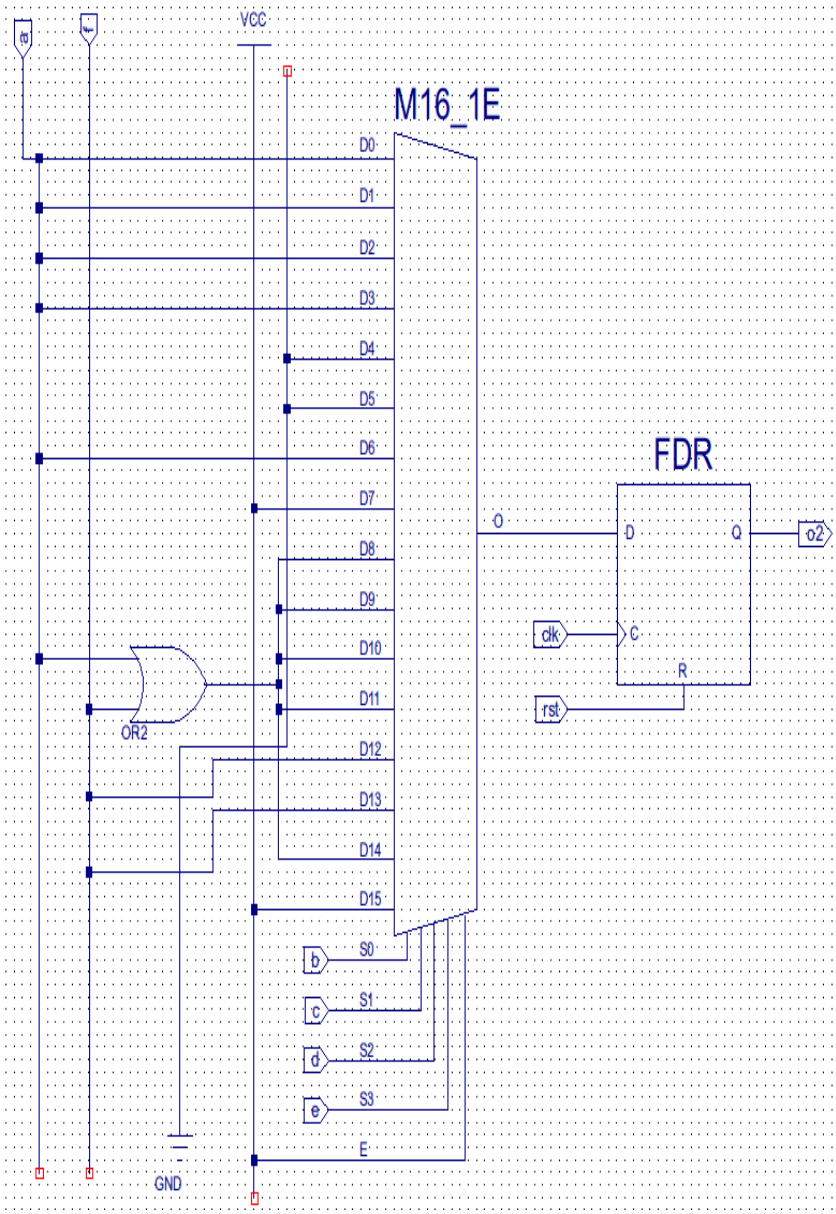
Ова вежба илуструје могућност употребе **IFV**-а у тзв. логичкој провери еквивалентности (енг. *Logical Equivalence Checking*, скраћено LEC) комбинационих кола. Потреба за LEC-ом се јавља у тзв. поступку мапирања технологије (енг. *technology mapping*), када је потребно верификовати еквивалентност технолошки независног „златног дизајна“ (енг. *golden design*) и његове технолошки зависне имплементације. Нпр. уколико се дизајн имплементира у *semi-custom standard cell* технологији, на располагању су тзв. библиотеке стандардних ћелија (енг. *standard cell libraries*). У зависности од расположивих елементарних логичких кола, златни дизајн се мора имплементирати коришћењем само оних кола која су на располагању у стандардној библиотеци (нпр. NOT, NAND2, NAND3 и NAND4 коло). Након имплементације, потребно је проверити логичку еквивалентност полазног златног кола и имплементираног кола.

Задатак 1. (решен)

На слици 1.1 је дат златни модел Буловог кола, реализован коришћењем NOT, AND и OR кола. На слици 1.2 је приказана имплементација Буловог кола коришћењем MUX и OR кола. Моделовати системе у **VHDL**-у, а затим коришћењем **IFV**-а испитати да ли Булова кола са слика 1.1 и 1.2 имају исту функционалност, тј. да ли су логички еквивалентна са становишта понашања улаз-излаз.



Слика 1.1


VCC
M16_1E
D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
D14
D15
S0
S1
S2
S3
E
O
FDR
D
Q
C
R
clk
rst
o2
OR2
GND

Слика 1.2

Решење:

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

entity example1 is
    port(
        rst : in  std_logic;
        clk   : in  std_logic;
        a     : in  std_logic;
        b     : in  std_logic;
        c     : in  std_logic;
        d     : in  std_logic;
        e     : in  std_logic;
        f     : in  std_logic;
        o1    : out std_logic;
        o2    : out std_logic
        );
end example1;

architecture arch of example1 is

    signal sel             : std_logic_vector(3 downto 0);
    signal o1_reg, o1_next : std_logic;
    signal o2_reg, o2_next : std_logic;

begin
    -- psl default clock is rising_edge(clk);
```

```
    -- psl P1 : assert always (o1 = o2);

    reg_proc : process(clk, rst)
    begin
        if (rst = '1') then
            o1_reg <= '0';
            o2_reg <= '0';
        elsif rising_edge(clk) then
            o1_reg <= o1_next;
            o2_reg <= o2_next;
        end if;
    end process reg_proc;

    o1 <= o1_reg;
    o2 <= o2_reg;

    o1_next <= (a and (not b) and c) or (a and (not d)) or
(b and c and d) or (e and f);

    with sel select
        o2_next <= a when "0000",
        a when "0001",
        a when "0010",
        a when "0011",
        '0'          when "0100",
        '0'          when "0101",
        a when "0110",
        '1'          when "0111",
        (a or f)     when "1000",
```

```
        (a or f)     when "1001",
        (a or f)     when "1010",
        (a or f)     when "1011",
        f when "1100",
        f when "1101",
        (a or f)     when "1110",
        '1'          when others;

    sel <= e & d & c & b;

end arch;
```

У **VHDL** датотеци, која садржи поставку овог проблема (опис система и спецификацију особине логичке еквивалентности), су имплементирана оба дигитална система. Ово је урађено да би поједноставио опис система. У духу добре дизајнерске праксе и у случајевима верификације сложених/великих дизајна, системе треба партиционисати на функционалне модуле. У овом примеру, уместо једне *vhd* датотеке било је могуће употребити три датотеке. У две датотеке би се имплементирали системи од интереса, док би се у хијерархијски највишој трећој датотеци инстанцирала оба система.

Оба Булова кола су моделована стандардним **VHDL** конкурентним изразима и процесима. Системи деле исте улазе: **`rst, clk, a, b, c, d, e`** и **`f`**. Излаз из првог система је **`o1`**, док је излаз из другог система **`o2`**.

Линије кода уз помоћ којих се реализује формална верификација се налазе у **VHDL** коментарима. Алат **IFV** узима у обзир само коментаре који почињу са **`-- psl`**.

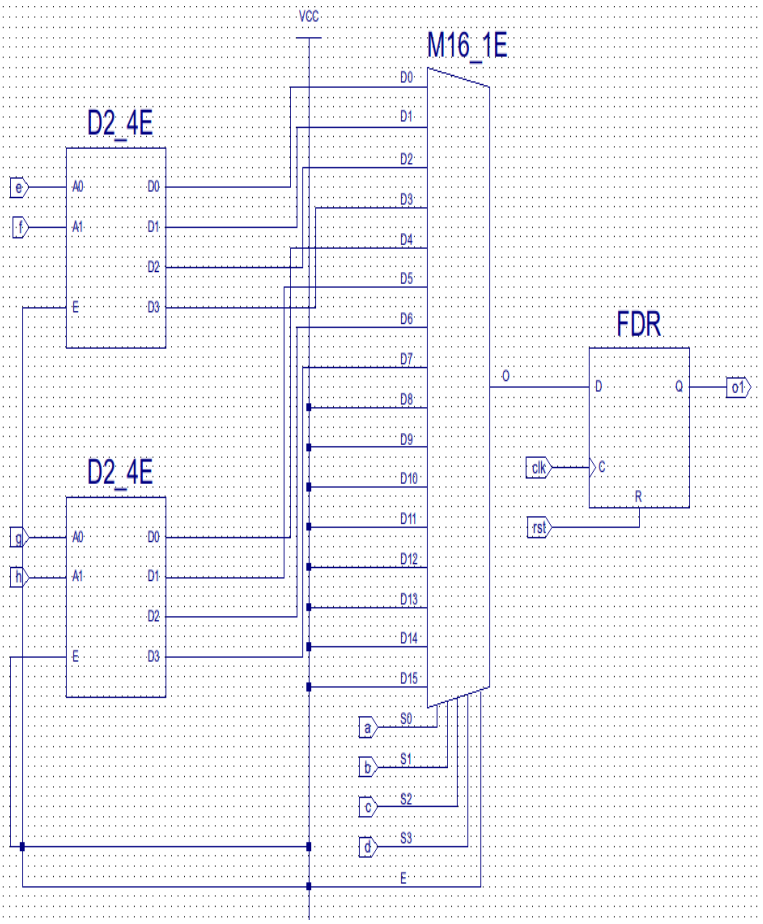
Прва верификациона наредба, **`-- psl default clock is rising_edge(clk);`** није типа тврдње, већ је припремна наредба, која прослеђује **IFV**-у подразумевани синхронизациони механизам. У овом случају, то је појава растуће ивице сигнала **clk**.

Друга верификациона наредба, **`-- psl P1 : assert always (o1 = o2);`** представља **PSL** тврдњу којом се проверава да ли су два сигнала логички еквивалентна. У овом примеру се проверава да ли су излази два система еквивалентни. Ова верификациона наредба има више елемената. Име особине коју

проверавамо је **P1**. Кључна реч **assert** означава да желимо да проверимо да ли особина у наставку израза важи. Изрази у **PSL**-у могу да се користе и за ограничавање допуштених улаза, и у том случају се користи реч **assume**. Сви улази су допуштени за овај пример, тако да се кључна реч **assume** не користи. Кључна реч **always** говори да особина који следи треба да важи увек. Уколико би ова реч била изостављена, особина која следи би се проверила само у првом циклусу. На крају следи и Булов исказ. Пошто се проверава логичка еквивалентност, употребљен је **VHDL** оператор једнакости.

Задатак 2.

Користећи **IFV** одредити да ли Булово коло са слике 1.3 имплементира Булову функцију задату формулом **out = (!A!B!C!E!F + A!B!CE!F + !AB!C!EF + AB!CEF + !A!BC!G!H + A!BCG!H + !ABC!GH + ABCGH)!D + D**.



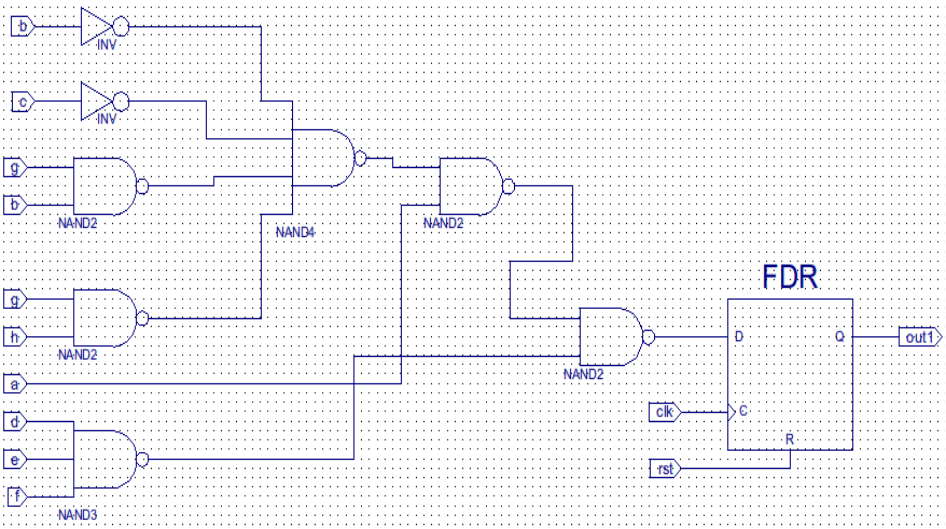
Слика 1.3

2. Провера логичке еквивалентности (LEC) комбинационих мрежа у FPGA технологији

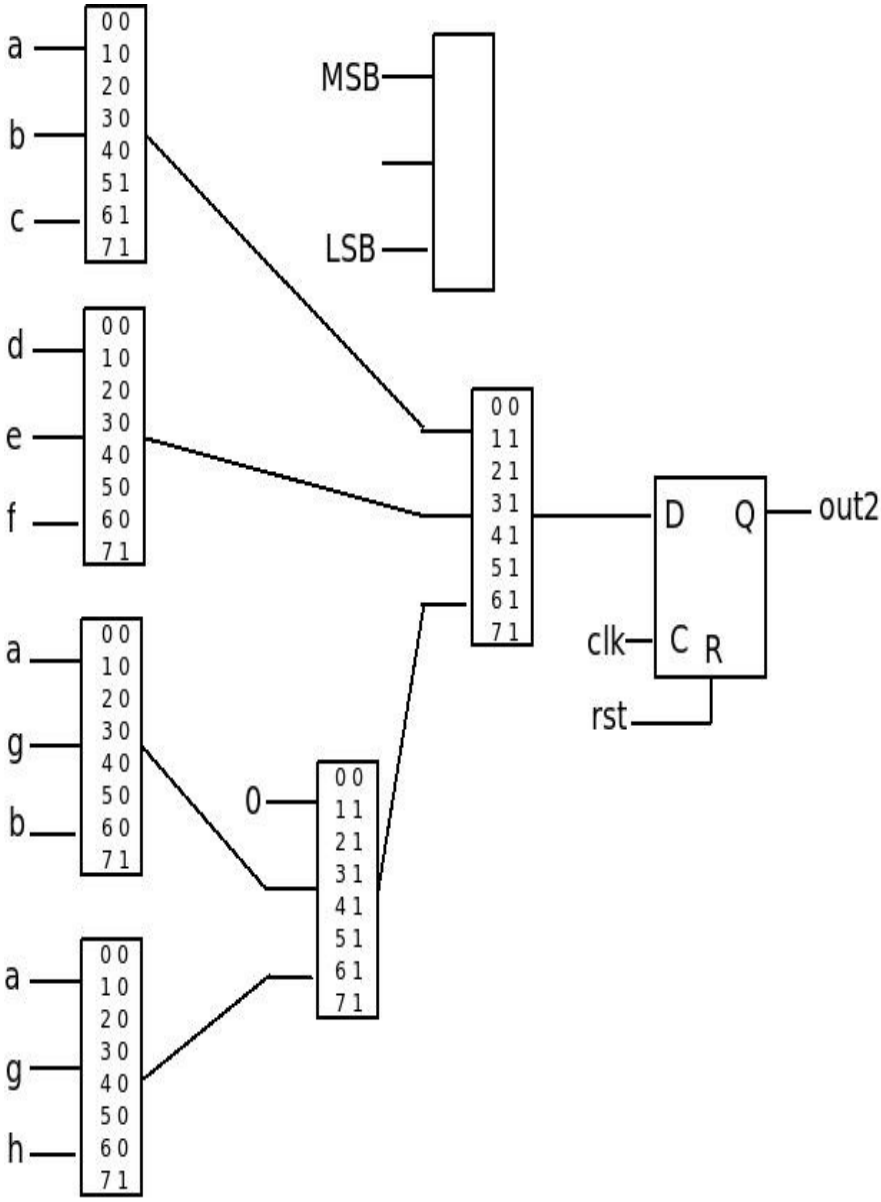
Логичка провера еквивалентности комбинационих кола је од интереса како код имплементација дизајна у *standard-cell* технологији, тако и у њиховом мапирању у **FPGA** технологији. За разлику од претходне вежбе, која је илустровала проверу логичке еквивалентности након мапирања кола на елементарна кола из неке стандардне библиотеке ћелија, код **FPGA** технологије се Булова кола имплементирају преко тзв. LUT-ова (скраћено од енг. *Look-up table*). Нпр. у Xilinx Virtex фамилији чипова, LUT-ови имају 64 места, што је довољно за упис истинитосне таблице шесто-улазне Булове функције. Након мапирања кола на LUT-ове, такође је могуће испитати логичку еквивалентност полазног златног кола и његове имплементације преко LUT-ова, што илуструје ова вежба.

Задатак 1. (решен)

На слици 2.1 је дат златни модел Буловог кола, реализован коришћењем NOT и NAND кола. На слици 2.2 је приказана имплементација Буловог кола коришћењем тробитних LUT-ова. Моделовати системе у **VHDL**-у, а затим коришћењем **IFV**-а испитати да ли су Булова кола са слика 2.1 и 2.2 логички еквивалентна.



Слика 2.1


a
b
c
00
10
20
30
40
51
61
71
MSB
LSB
d
e
f
00
10
20
30
40
50
60
71
00
11
21
31
41
51
61
71
D
Q
out2
clk
C
R
rst
a
g
b
00
10
20
30
40
50
60
71
0
00
11
21
31
41
51
61
71
a
g
h
00
10
20
30
40
50
60
71

Слика 2.2

Решење:

Модел тробитног LUT-а је имплементиран у посебној датотеци.

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

entity lut3 is
   generic(INIT : std_logic_vector(7 downto 0));
   port(
      I2 : in  std_logic;
      I1 : in  std_logic;
      I0 : in  std_logic;
      O  : out  std_logic
   );
end lut3;

architecture rtl of lut3 is

   signal input : std_logic_vector(2 downto 0);

begin

   LUT3_proc : process(input)
   begin
      case input is
         when "000" => O <= INIT(0);
         when "001" => O <= INIT(1);
         when "010" => O <= INIT(2);
         when "011" => O <= INIT(3);
```

```
        when "100" => O <= INIT(4);
        when "101" => O <= INIT(5);
        when "110" => O <= INIT(6);
        when others => O <= INIT(7);
      end case;
   end process LUT3_proc;

   input <= I2 & I1 & I0;

end rtl;
```

Оба Булова кола од интереса су имплементирана у једној датотеци. У овом моделу се инстацирају раније описане тробитне LUT компоненте.

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

entity lec_fpga is
    port(
        rst  : in  std_logic;
        clk  : in  std_logic;
        a    : in  std_logic;
        b    : in  std_logic;
        c    : in  std_logic;
        d    : in  std_logic;
        e    : in  std_logic;
        f    : in  std_logic;
        g    : in  std_logic;
        h    : in  std_logic;
        out1 : out std_logic;
        out2 : out std_logic
```

```
        );
end lec_fpga;

architecture rtl of lec_fpga is

    signal lut1_6, lut2_6, lut3_5, lut4_5, lut5_6 :
std_logic;
    signal out1_reg, out1_next                    :
std_logic;
    signal out2_reg, out2_next                    :
std_logic;

begin

    -- psl default clock is rising_edge(clk);
    -- psl P1 : assert always (out1 = out2);

    reg_proc : process(clk, rst)
    begin
        if (rst = '1') then
            out1_reg <= '0';
            out2_reg <= '0';
        elsif rising_edge(clk) then
            out1_reg <= out1_next;
            out2_reg <= out2_next;
        end if;
    end process reg_proc;

    out1 <= out1_reg;
```

```
    out2 <= out2_reg;

    out1_next <= not (not (not (not (b) and not (c) and not
(g and b) and
                                    (not (g and h))) and a) and
not (d and e and f));


    look_up_table_1 : entity work.lut3
        generic map(INIT => "11100000")
        port map(
            I0 => c,
            I1 => b,
            I2 => a,
            O  => lut1_6);

    look_up_table_2 : entity work.lut3
        generic map(INIT => "10000000")
        port map(
            I0 => f,
            I1 => e,
            I2 => d,
            O  => lut2_6);

    look_up_table_3 : entity work.lut3
        generic map(INIT => "10000000")
        port map(
            I0 => b,
            I1 => g,
```

```
            I2 => a,
            O  => lut3_5);

    look_up_table_4 : entity work.lut3
        generic map(INIT => "10000000")
        port map(
            I0 => h,
            I1 => g,
            I2 => a,
            O  => lut4_5);

    look_up_table_5 : entity work.lut3
        generic map(INIT => "11111110")
        port map(
            I0 => lut4_5,
            I1 => lut3_5,
            I2 => '0',
            O  => lut5_6);

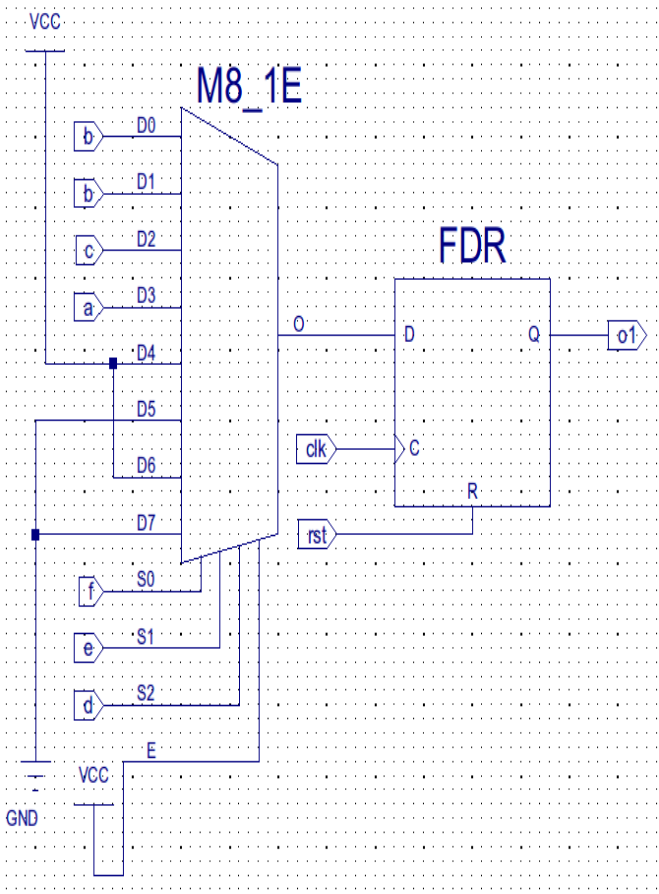
    look_up_table_6 : entity work.lut3
        generic map(INIT => "11111110")
        port map(
            I0 => lut5_6,
            I1 => lut2_6,
            I2 => lut1_6,
            O  => out2_next);

end rtl;
```

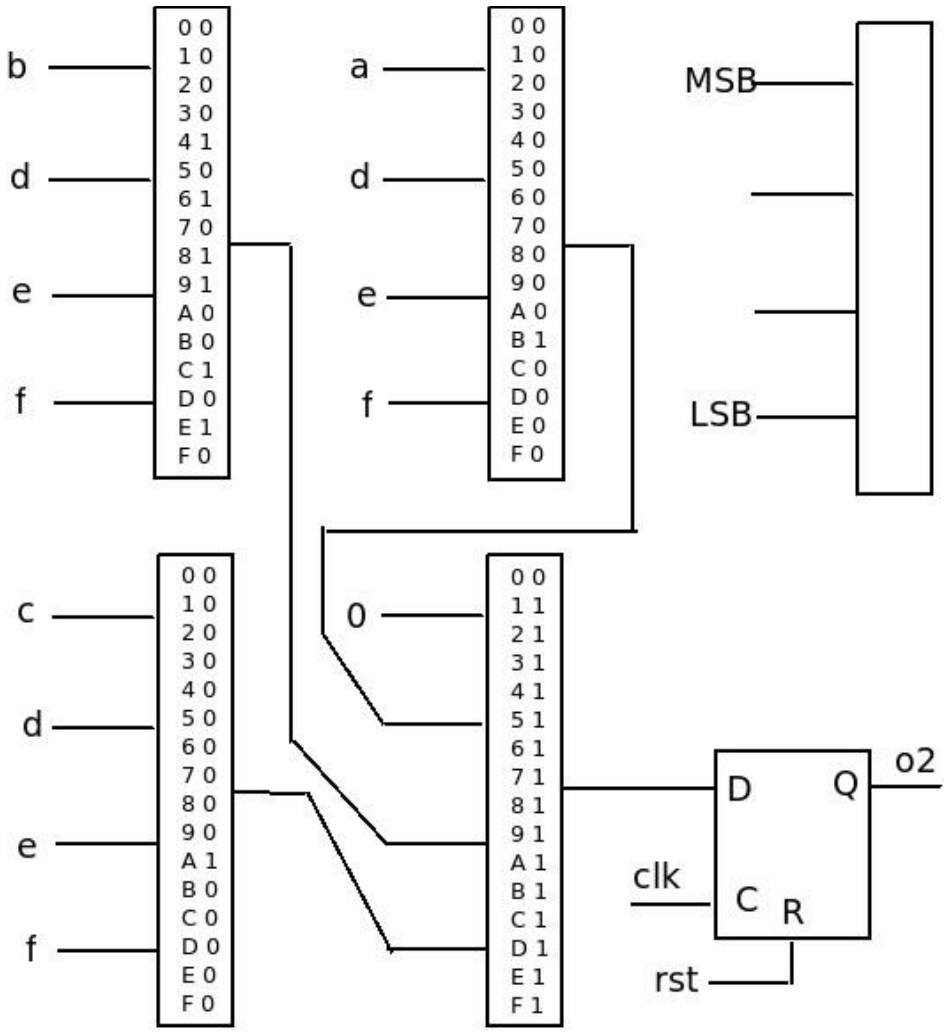
У овом примеру модел тробитног LUT-а је реализован као посебан **VHDL** **`entity`**. Потом је у другом моделу инстанциран LUT модел, да би се кола реализовала у FPGA технологији. Прво Булово коло је имплементирано на уобичајеном RTL нивоу апстракције.

Задатак 2.

На слици 2.3 је дат златни модел Буловог кола, реализован коришћењем MUX кола. На слици 2.4 је приказана имплементација Буловог кола коришћењем четворобитних LUT-ова. Моделовати системе у **VHDL**-у, а затим коришћењем **IFV**-а испитати да ли су Булова кола са слика 2.3 и 2.4 логички еквивалентна.



Слика 2.3


b
d
e
f
0 0
1 0
2 0
3 0
4 1
5 0
6 1
7 0
8 1
9 1
A 0
B 0
C 1
D 0
E 1
F 0
a
d
e
f
0 0
1 0
2 0
3 0
4 0
5 0
6 0
7 0
8 0
9 0
A 0
B 1
C 0
D 0
E 0
F 0
MSB
LSB
c
d
e
f
0 0
1 0
2 0
3 0
4 0
5 0
6 0
7 0
8 0
9 0
A 1
B 0
C 0
D 0
E 0
F 0
0
0 0
1 1
2 1
3 1
4 1
5 1
6 1
7 1
8 1
9 1
A 1
B 1
C 1
D 1
E 1
F 1
D
Q
o2
clk
C
R
rst

Слика 2.4

Задатак 3.

Дата су два **VHDL** модела кола.

Модел 1:

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


entity n1 is
    Port ( a : in  std_logic_vector (3 downto 0);
           y : out  std_logic;
           z : out  std_logic);
end n1;

architecture RTL of n1 is

    signal t: std_logic_vector (1 downto 0);

begin

    t(1) <= a(1) xor a(0);
    t(0) <= a(3) and a(2);
    with t select
        y <= '0' when "01",
             '1' when others;
    with t select
        z <= '1' when "00",
             '1' when "11",
             '0' when others;

end RTL;
```

Модел 2:

```
library ieee;
use ieee.std_logic_1164.ALL;

entity n1_lec is
  port (
    y : out std_logic;
    z : out std_logic;
```

```
    a : in std_logic_vector ( 3 downto 0 )
  );
end n1_lec;

architecture GLVL of n1 is

begin
  y1 : GEN_LUT4
    generic map(
      INIT => X"??FF"
    )
    port map (
      I0 => a(1),
      I1 => a(0),
      I2 => a(3),
      I3 => a(2),
      O => y
    );

  Mrom_z11 : GEN_LUT4
    generic map(
      INIT => X"??87"
    )
    port map (
      I0 => a(2),
      I1 => a(3),
      I2 => a(0),
      I3 => a(1),
      O => z
    );

end GLVL;
```

Користећи **IFV**, у моделу 2 одредити непознате цифре (означене са ?) унутар хексадецималних бројева који служе за иницијализацију два четвороулазна LUT-а, тако да модели 1 и 2 буду логички еквивалентни.

3. Превођење неформалне спецификације особина у формалну

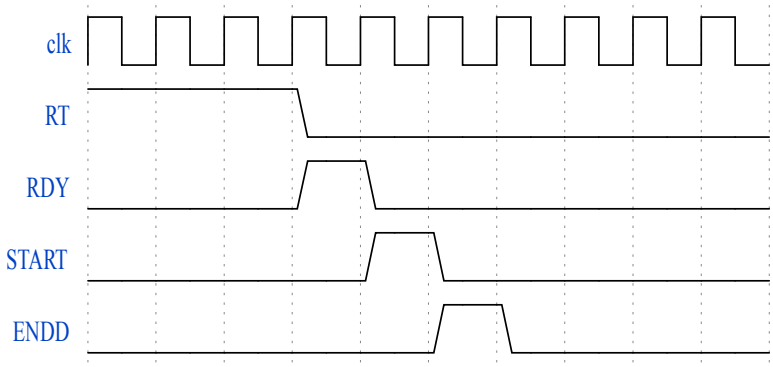
Приликом развоја неког дигиталног хардверског уређаја, уобичајено је да се читав процес раздвоји на више корака, међу којима су најважнији дизајн архитектуре система, имплементација система и његова верификација. Захтеване особине система се у фази дизајна исказују неформално, (нпр. списак особина које систем мора да задовољи, шта не сме да се деси итд.). Ова спецификација се са једне стране користи за имплементацију система, најчешће његовим описом у неком од **HDL** језика (језика за опис хардвера, од енг. *Hardware Description Language*). Са друге стране, листа неформалних специфицираних особина се може превести у формалну, у циљу верификације имплементираног дизајна. Ова вежба је посвећана превођењу неформалне у формалну спецификацију особина, као и њиховој верификацији у **IFV-**у. Вежба истовремено илуструје експресивност **PSL**-а .

Вежбу чине два дела. У првом делу је дато десет примера неформалних особина неког хипотетичког уређаја, које су све преведене у одговарајуће формуле **PSL**-а. За сваку од десет особина је задат одговарајући скуп сигнала тог уређаја, за које треба утврдити да ли задовољавају или не захтевану особину. Сигнали и захтеване особине су моделовани у **VHDL-**у.

У другом делу вежбе је задато нових десет неформалних особина, које прате сигнали са одговарајућих слика. Сваку особину треба формално исказати у **PSL**-у, а затим треба верификовати да ли се сигнали са одговарајућих слика понашају у складу са специфицираном формулом. Десет задатака треба решити коришћењем **IFV**-а. Напомена: изразе „сигнал је активан“, „сигнал је на 1“, „сигнал је на високом нивоу“, „сигнал је постављен“, „сигнал је сетован“. ћемо користити као синониме. Аналогно се као синоними могу користити изрази „сигнал је неактиван“, „сигнал је на 0“, „сигнал је на ниском нивоу“, „сигнал није постављен“ или „сигнал је ресетован“.

Пример 1.

Док год је RT активан, START, ENDD и RDY су неактивни.



Слика 3.1

PSL-формула:

```
assert always (RT -> not (RDY or START or ENDD))
```

Решење:

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

entity prop01 is
    port(
        clk   : in  std_logic;
        RT    : out std_logic;
        RDY   : out std_logic;
        START : out std_logic;
        ENDD  : out std_logic
        );
end prop01;
```

```
architecture rtl of prop01 is

    signal cnt : unsigned(3 downto 0) := "0000";

begin

    -- psl default clock is rising_edge(clk);
    -- psl P1 : assert always (RT -> not (RDY or START or
ENDD));

    process(clk)
    begin
        if rising_edge(clk) then
            cnt <= cnt + "0001";
        end if;
    end process;

    with cnt select
        RT <= '1' when "0000"|"0001"|"0010",
        '0'       when others;

    with cnt select
        RDY <= '1' when "0011",
        '0'        when others;

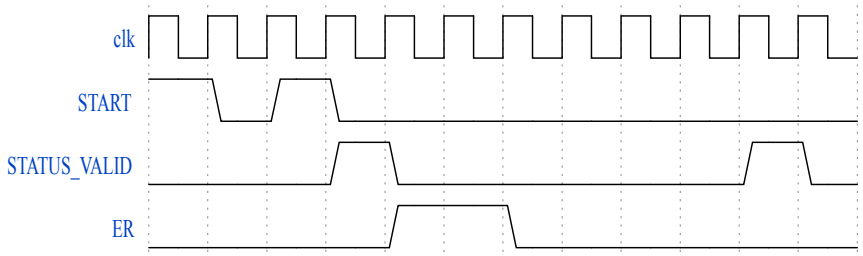
    with cnt select
        START <= '1' when "0100",
        '0'          when others;
```

```
    with cnt select
        ENDD <= '1' when "0101",
        '0'         when others;

end rtl;
```

Пример 2.

Сигнали START, STATUS_VALID и ERROR смеју да буду активни само један циклус.



Слика 3.2

PSL-формуле:

```
assert always (START -> next (not START))
assert always (STATUS_VALID -> next (not STATUS_VALID))
assert always (ER -> next (not ER));
```

Решење:

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

entity prop02 is
    port(
```

```
        clk          : in  std_logic;
        START        : out std_logic;
        STATUS_VALID : out std_logic;
        ER           : out std_logic
        );
end prop02;

architecture rtl of prop02 is

    signal clk : unsigned(7 downto 0) := X"00";

begin

    -- psl default clock is rising_edge(clk);

    -- psl P1 : assert always (START -> next (not START));
    -- psl P2 : assert always (STATUS_VALID -> next (not
STATUS_VALID));
    -- psl P3 : assert always (ER -> next (not ER));

    process(clk)
    begin
        if rising_edge(clk) then
            clk <= clk + X"01";
        end if;
    end process;

    with clk select
        START <= '1' when X"00"|X"02",
```

```
        '0'          when others;

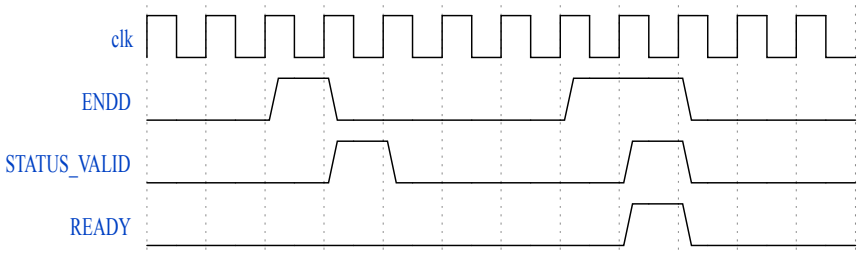
    with clk select
        STATUS_VALID <= '1' when X"03"|X"09",
        '0'                 when others;

    with clk select
        ER <= '1' when X"04"|X"05",
        '0'       when others;

end rtl;
```

Пример 3.

END је активан само један циклус, осим ако и START и RDY нису активни.



Слика 3.3

PSL-формула:

```
assert always (ENDD -> next ((not ENDD) or (ENDD and READY
and STATUS_VALID)))
```

Решење:

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

```
entity prop03 is
    port(
        clk          : in  std_logic;
        ENDD         : out std_logic;
        STATUS_VALID : out std_logic;
        READY        : out std_logic
        );
end prop03;

architecture rtl of prop03 is

    signal cnt : unsigned(4 downto 0) := "00000";

begin

    -- psl default clock is rising_edge(clk);

    -- psl P1 : assert always (ENDD -> next ((not ENDD) or
(ENDD and READY and STATUS_VALID)));

    process(clk)
    begin
        if rising_edge(clk) then
            cnt <= cnt + "00001";
        end if;
    end process;

    with cnt select
```

```
        ENDD <= '1' when "00010"|"00111"|"01000",
        '0'         when others;

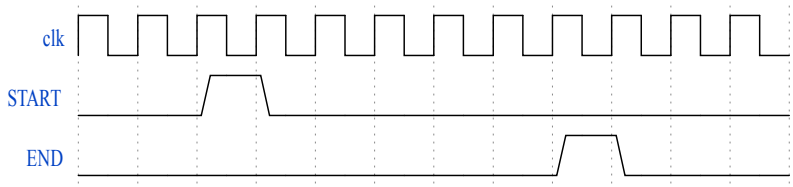
    with cnt select
        STATUS_VALID <= '1' when "00011"|"01000",
        '0'                 when others;

    with cnt select
        READY <= '1' when "01000",
        '0'          when others;

end rtl;
```

Пример 4.

Ако је END активан, пре тога мора да је START био активан.



Слика 3.4

PSL-формуле:

- `assert ((not ENDD) until START)`
- `assert always (START -> next (ENDD before START))`

Решење:

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

```
entity prop04 is
    port(
        clk   : in  std_logic;
        START : out std_logic;
        ENDD  : out std_logic
        );
end prop04;

architecture rtl of prop04 is

    signal cnt : unsigned(4 downto 0) := "00000";

begin

    -- psl default clock is rising_edge(clk);

    -- psl P1 : assert ((not ENDD) until START);
    -- psl P2 : assert always (START -> next (ENDD before
START));



    cnt_signal : process(clk)
    begin
        if rising_edge(clk) then
            cnt <= cnt + 1;
        end if;
    end process cnt_signal;
```

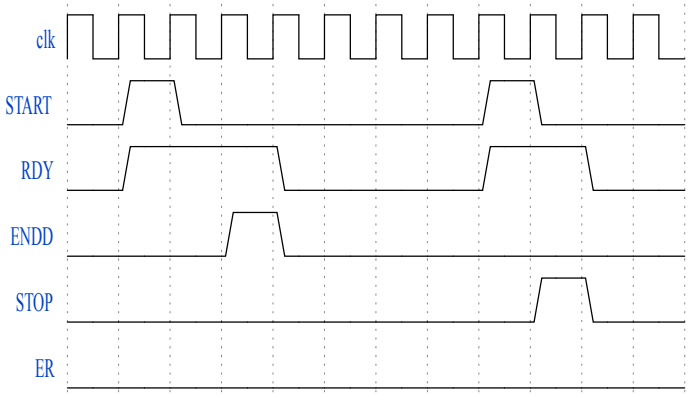
```
    with cnt select
        START <= '1' when "00010",
        '0'          when others;

    with cnt select
        ENDD <= '1' when "01000",
        '0'         when others;

end rtl;
```

Пример 5.

Да би се START активирао, RDY треба да је активан. RDY мора остати активан док год се неки од сигнала END, STOP или ERROR не активирају.



Слика 3.5

PSL-формуле:

```
assert always ({not START;  START}  |-> {RDY})
assert always ({RDY and (ENDD or STOP or ER)} |=> {not
RDY})
```

Решење:

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

entity prop05 is
   port(
      clk   : in std_logic;
      START : out std_logic;
      RDY   : out std_logic;
      ENDD  : out std_logic;
      STOP  : out std_logic;
      ER : out std_logic
   );
end prop05;

architecture beh of prop05 is

   signal cnt : unsigned(3 downto 0) := X"0";

begin

   -- psl default clock is rising_edge(clk);

   -- psl P1a : assert always ({not START;  START}  |->
{RDY});
   -- psl P2a : assert always ({RDY and (ENDD or STOP or
ER)} |=> {not RDY});
   -- psl P2b : assert always ({RDY; not RDY} |->
{prev(ENDD or STOP or ER)});
```

```
    process(clk)
    begin
       if rising_edge(clk) then
          cnt <= cnt + X"1";
       end if;
    end process;

    with cnt select
       START <= '1' when X"1"|X"8",
              '0' when others;

    with cnt select
       RDY <= '1' when X"0"|X"1"|X"2"|X"3"|X"7"|X"8"|X"9",
              '0' when others;

    with cnt select
       ENDD <= '1' when X"3",
              '0' when others;

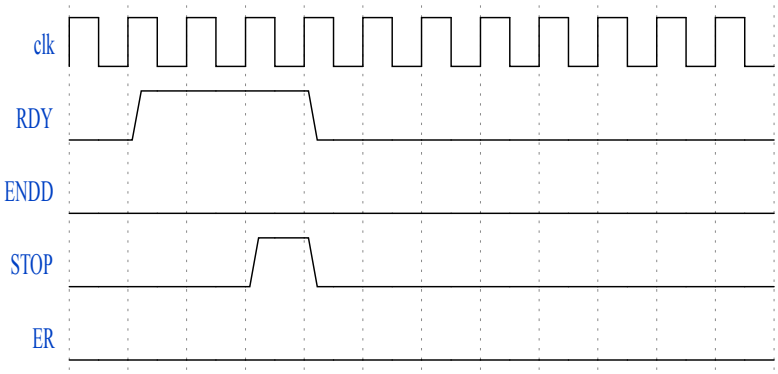
    with cnt select
       STOP <= '1' when X"9",
              '0' when others;

    ER <= '0';

end beh;
```

Пример 6.

Сигнал RDY мора постати активан, и кад-тад мора постати неактиван. RDY може постати неактиван само ако је неки од сигнала END, STOP или ERROR активан.



Слика 3.6

PSL-формуле:

- ```
  assert eventually! (RDY)
  assert always (RDY -> eventually! (not RDY))
  assert always {RDY; not RDY} |-> {prev(ENDD or STOP or
  ER)}
  ```

Решење:

```
library ieee;

use ieee.std_logic_1164.all;

use ieee.numeric_std.all;


entity prop06 is

    port(

        clk  : in  std_logic;

        RDY  : out std_logic;

        ENDD : out std_logic;
```
```

```
        STOP : out std_logic;
        ER   : out std_logic
        );
end prop06;

architecture rtl of prop06 is

    signal cnt : unsigned(3 downto 0) := X"0";

begin

    -- psl default clock is rising_edge(clk);

    -- psl P1 : assert eventually! (RDY);
    -- psl P2 : assert always (RDY -> eventually! (not
RDY));
    -- psl P3b : assert always {RDY; not RDY} |->
{prev(ENDD or STOP or ER)};

    process(clk)
    begin
        if rising_edge(clk) then
            cnt <= cnt + X"1";
        end if;
    end process;

    with cnt select
        RDY <= '1' when X"1"|X"2"|X"3",
        '0'        when others;
```

```
    ENDD <= '0';

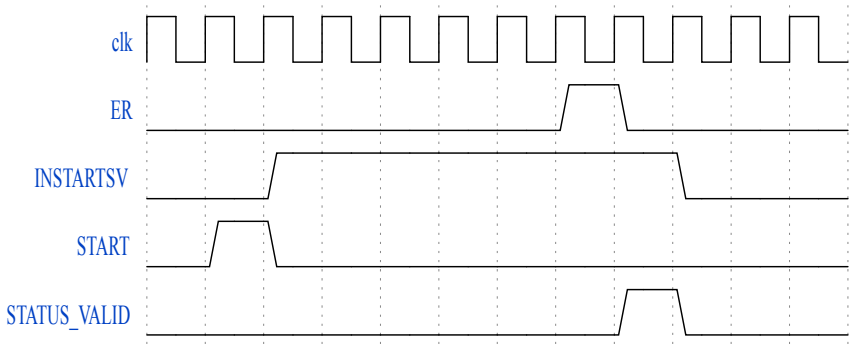
    with cnt select
        STOP <= '1' when X"3",
        '0'         when others;

    ER <= '0';

end rtl;
```

Пример 7.

ERROR сме бити активан само ако је INSTARTSV активан.



Слика 3.7

PSL-формула:

```
assert always (not INSTARTSV -> not ER)
```

Решење:

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

```
use ieee.numeric_std.all;

entity prop07 is
    port(
        clk          : in  std_logic;
        ER           : out std_logic;
        INSTARTSV    : out std_logic;
        START        : out std_logic;
        STATUS_VALID : out std_logic
        );
end prop07;

architecture rtl of prop07 is

    signal cnt : unsigned(3 downto 0) := X"0";

begin

    -- psl default clock is rising_edge(clk);

    -- psl P1a : assert always (ER -> INSTARTSV);
    -- psl P1b : assert always (not INSTARTSV -> not ER);

    cnt_signal : process(clk)
    begin
        if rising_edge(clk) then
            cnt <= cnt + X"1";
        end if;
```

```
    end process cnt_signal;

    with cnt select
        ER <= '1' when X"7",
        '0'       when others;

    with cnt select
        INSTARTSV <= '1' when X"2"|X"3"|X"4"|X"5"|X"6"|
X"7"|X"8",
        '0'              when others;

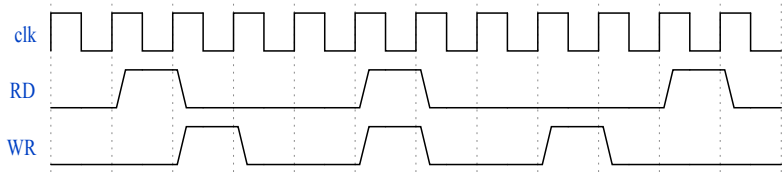
    with cnt select
        START <= '1' when X"1",
        '0'          when others;

    with cnt select
        STATUS_VALID <= '1' when X"8",
        '0'                 when others;

end rtl;
```

Пример 8.

RD и WR никада не смеју бити активни истовремено.



Слика 3.8

PSL-формула:

```
assert never (RD and WR)
```

Решење:

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

entity prop08 is
   port(
      clk : in std_logic;
      RD  : out std_logic;
      WR  : out std_logic
   );
end prop08;

architecture rtl of prop08 is

   signal cnt : unsigned(3 downto 0) := X"0";

begin

   -- psl default clock is rising_edge(clk);

   -- psl P1a : assert always (not (RD and WR));
   -- psl P1b : assert never (RD and WR);

   cnt_signal : process(clk)
   begin
```

```
      if rising_edge(clk) then
         cnt <= cnt + X"1";
      end if;
   end process cnt_signal;

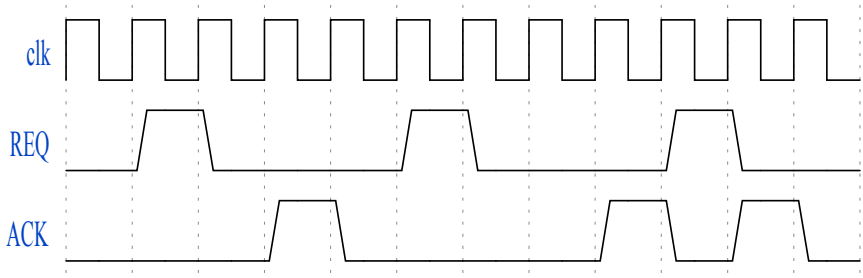
   with cnt select
      RD <= '1' when X"1"|X"5"|X"A",
            '0' when others;

   with cnt select
      WR <= '1' when X"2"|X"5"|X"8",
            '0' when others;

end rtl;
```

Пример 9.

Након сваког REQ мора наићи ACK пре него што се опет појави нови REQ. Оба сигнала су активна највише 1 циклус.



Слика 3.9

PSL-формула:

```
assert always (REQ -> next (ACK before! REQ))
```

Решење:

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

entity prop09 is
    port(
        clk : in  std_logic;
        REQ : out std_logic;
        ACK : out std_logic
        );
end prop09;

architecture rtl of prop09 is

    signal cnt : unsigned(3 downto 0) := X"0";

begin

    -- psl default clock is rising_edge(clk);

    -- psl P1a : assert always (REQ -> next (ACK before!
REQ));
    -- psl P1b : assert always (REQ -> next ((not REQ)
until ACK));

    -- psl P2 : assert always (REQ -> next (not REQ));
    -- psl P3 : assert always (ACK -> next (not ACK));
```

```
    cnt_p : process(clk)
    begin
        if rising_edge(clk) then
            cnt <= cnt + X"1";
        end if;
    end process cnt_p;

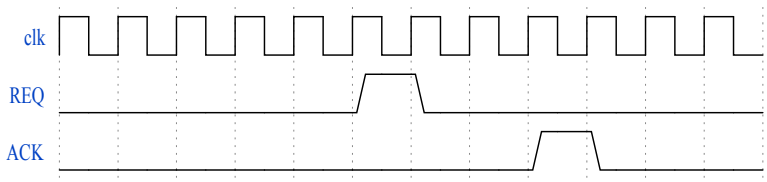
    process(cnt)
    begin
        case cnt is
            when X"1" => REQ <= '1';
                         ACK <= '0';
            when X"3" => REQ <= '0';
                         ACK <= '1';
            when X"5" => REQ <= '1';
                         ACK <= '0';
            when X"9" => REQ <= '0';
                         ACK <= '1';
            when X"A" => REQ <= '1';
                         ACK <= '0';
            when X"B" => REQ <= '0';
                         ACK <= '1';
            when others => REQ <= '0';
                           ACK <= '0';
        end case;
    end process;

end rtl;
```

Пример 10.

Ако је сигнал ACK активан, тада је REQ био активан пре 3 циклуса.

PSL-формула:



Слика 3.10

```
assert always (ACK -> prev(REQ, 3))
```

Решење:

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

entity prop10 is
   port(
      clk : in std_logic;
      REQ : out std_logic;
      ACK : out std_logic
   );
end prop10;

architecture rtl of prop10 is

   signal cnt : unsigned(3 downto 0) := X"0";
```

```
begin

   -- psl default clock is rising_edge(clk);
   -- psl P1 : assert always (ACK -> prev(REQ, 3));

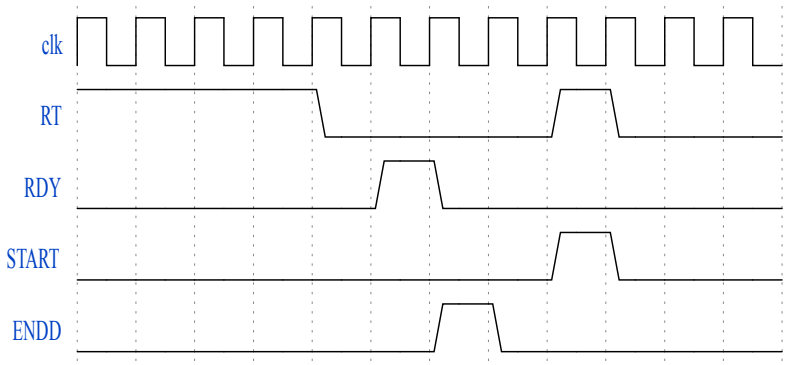
   process(clk)
   begin
      if rising_edge(clk) then
         cnt <= cnt + X"1";
      end if;
   end process;

   process(cnt)
   begin
      case cnt is
         when X"5" => REQ <= '1';
         when others => REQ <= '0';
      end case;
   end process;

   process(cnt)
   begin
      case cnt is
         when X"8" => ACK <= '1';
         when others => ACK <= '0';
      end case;
   end process;
end rtl;
```

Задатак 1.

RT је иницијално активан, и док год не постане први пут неактиван, START, ENDD и RDY не смеју бити активни.

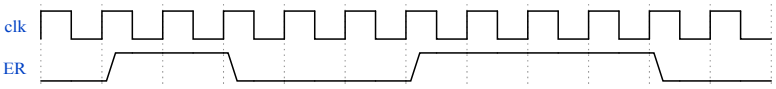


Слика 3.11

Напомена: Обратите пажњу да особина треба да важи само до прве деактивације RT сигнала. Да ли је **`always`** оператор потребан ?

Задатак 2.

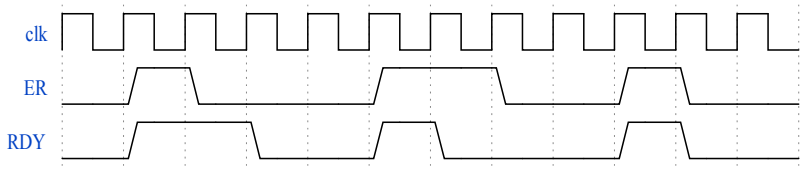
Сигнал ER не сме бити активан више од 3 циклуса узастопно.



Слика 3.12

Задатак 3.

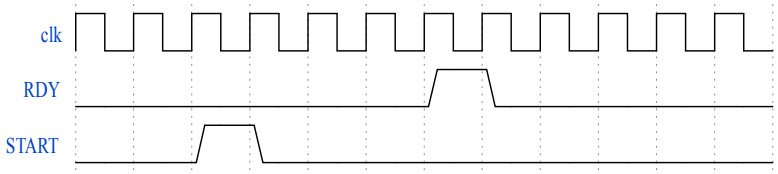
ER и RDY могу бити активни истовремено само један циклус. У наредном циклусу бар један од њих мора бити неактиван.



Слика 3.13

Задатак 4.

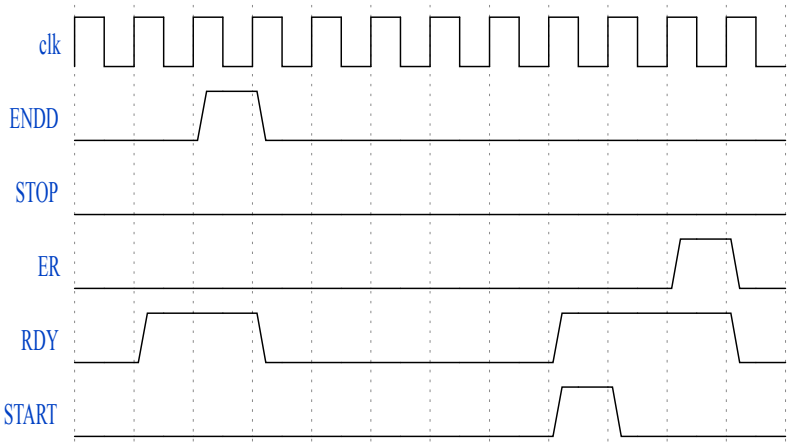
RDY се не сме активирати пре него што се START активира.



Слика 3.14

Задатак 5.

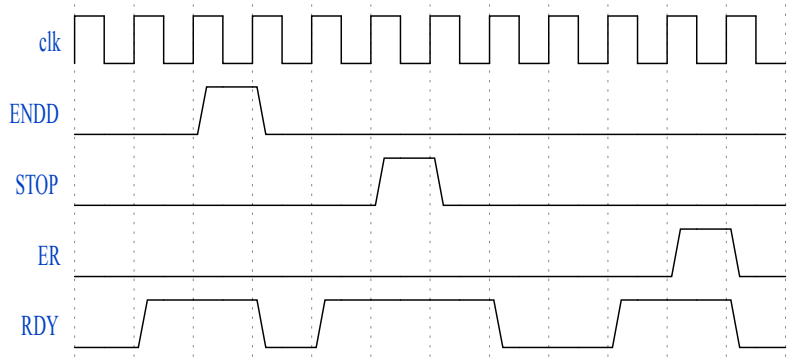
Након што неки од сигнала ENDD, STOP или ER постане активан, RDY мора постати неактиван у следећем тренутку.



Слика 3.15

Задатак 6.

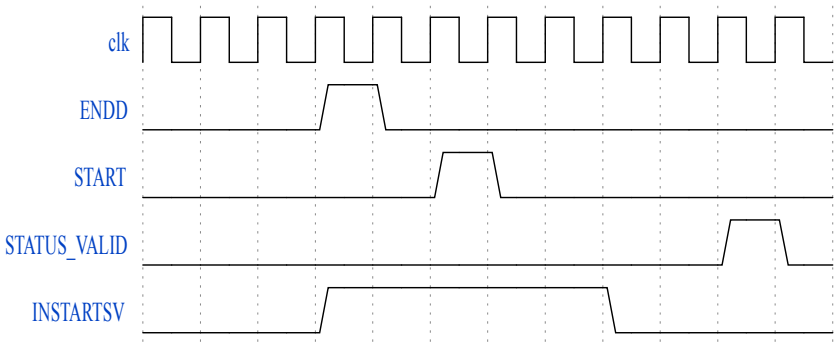
Сигнал ENDD, STOP и ER смеју бити активни само ако је RDY активан.



Слика 3.16

Задатак 7.

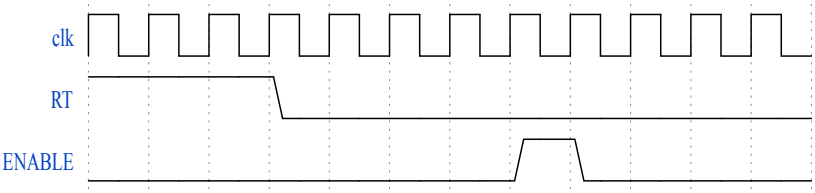
Сигнал ENDD може бити активиран само ако су START и STATUS_VALID неактивни.



Слика 3.17

Задатак 8.

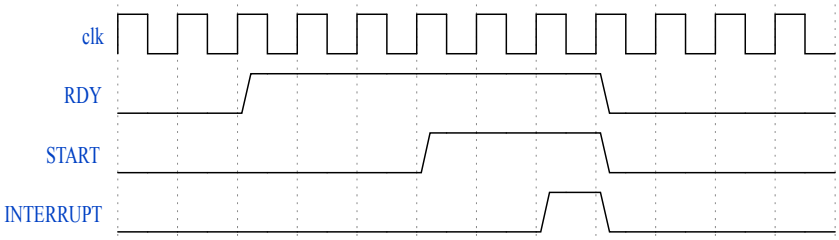
Док год је RT активан ENABLE мора бити неактиван. Сигнал ENABLE се може активирати тек 2 циклуса након што је RT постао неактиван (другим речима, потребно је да прођу бар 2 циклуса).



Слика 3.18

Задатак 9.

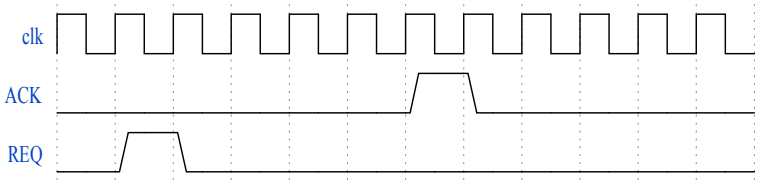
RDY и START могу да постану неактивни само ако је INTERRUPT активан.



Слика 3.19

Задатак 10.

Ако је REQ активан, тада ће ACK бити активан за 5 циклуса.



Слика 3.20

Задатак 11.

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

entity prop is
end entity;

architecture beh of prop is
    constant W : natural := 4;
    signal clk : std_logic;
    signal p11, p12 : std_logic;
    signal p21, p22 : std_logic;
    signal p31 : std_logic;
    signal p41 : std_logic;
    signal p51, p52 : std_logic;
    signal cnt : unsigned(W-1 downto 0);
begin

    process (clk)
    begin
        if rising_edge(clk) then
            cnt <= cnt + 1;
        end if;
    end process;

    process (clk)
    begin
        if rising_edge(clk) then
            if cnt = to_unsigned(6,W) then
                p11 <= '1';
            else
                p11 <= '0';
            end if;
            if cnt = to_unsigned(10,W) then
                p12 <= '1';
            else
                p12 <= '0';
            end if;
        end if;
```

```
    end process;

    process (clk)
    begin
        if rising_edge(clk) then
            if cnt < to_unsigned(8,W) then
                p21 <= '0';
                p22 <= '1';
            else
                p21 <= '1';
                p22 <= '0';
            end if;
        end if;
    end process;

    p31 <= cnt(1) xor cnt(0);

    process (clk)
    begin
        if rising_edge(clk) then
            if cnt < to_unsigned(14,W) then
                p41 <= '1';
            end if;
        end if;
    end process;

    p51 <= cnt(1);
    p52 <= cnt(2);

    --always (s1 -> next_a[1 to inf]s2);

end architecture;
```

За претходни VHDL модел, коришћењем **IFV**-а формално испитати следеће особине:

1. Након сваког импулса p11 појавиће се импулс p12.
2. Сигнали p21 и p22 никада нису једнаки.
3. Сигнал p21 је 50% времена на 0 а 50 % је на 1.
4. Колико је трајање импулса сигнала p31?
5. Сигнал p41 ће постати 0.

6. Сигнал p41 ће постајати 0.
7. На сваку опадајућу ивицу сигнала p51 сигнал p52 мења вредност.
8. Када год сигнал p52 задржава вредност, p51 мења.

4. Верификација секвенцијалних кола

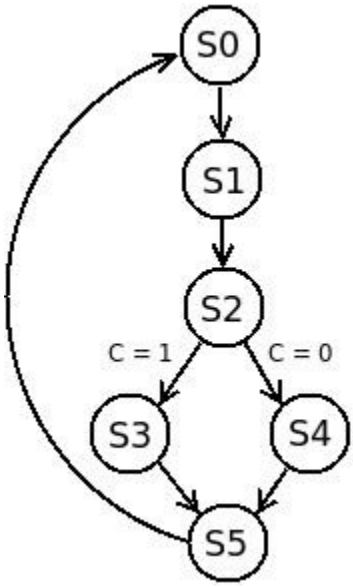
Дигитална кола уобичајено садрже меморијске елементе као што су флип-флопови, регистри, **RAM** итд. Због тога је од интереса верификовати понашање секвенцијалних кола, код којих тренутни излаз може зависити не само од тренутних улаза, већ и од њихових вредности у прошлости. За разлику од вежби посвећених верификацији логичке еквивалентности Булових кола, приступ верификацији секвенцијалних кола је базиран на моћнијем формализму коначних аутомата (енг. *Finite State Machine*, кратко **FSM**). Ова вежба је посвећена илустровању могућности верификације различитих особина **FSM**-а коришћењем **IFV-a**.

Задатак 1.

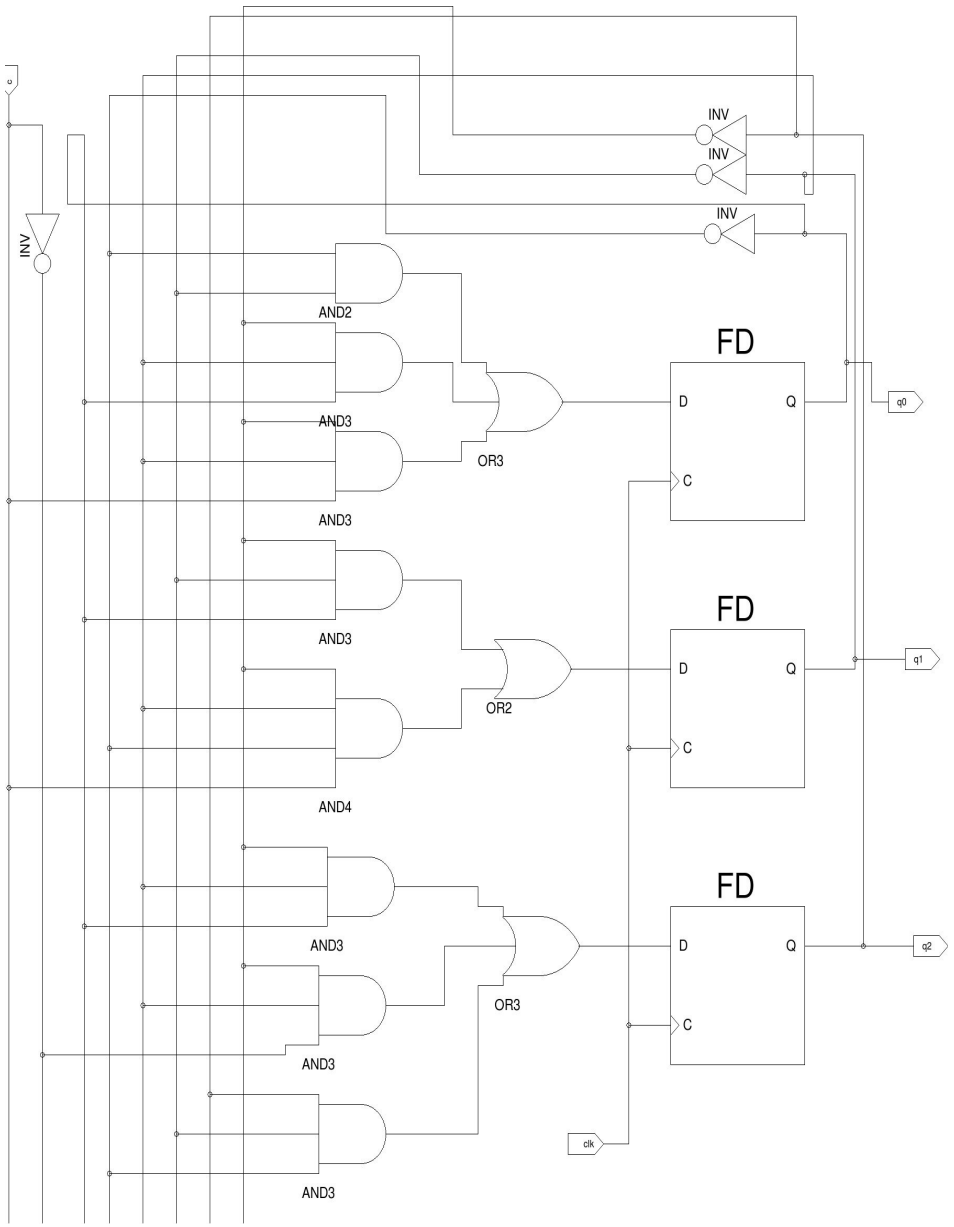
На слици 4.1 је приказан дијаграм прелаза стања Муровог аутомата, чија је шематска имплементација приказана на слици 4.2. Моделовати систем са слике 4.2 у **VHDL**-у, а затим коришћењем **IFV**-а испитати да ли за аутомат важе следеће особине:

- Два циклуса након стања s5 следи s1. *(true)*
- Два циклуса након стања s0 следи s2. *(true)*
- Стање s0 се појављује сваки пети циклус. *(true)*
- Аутомат у неком циклусу мора проћи кроз стање s2. *(true)*
- Аутомат у неком циклусу мора проћи кроз стање s3. *(false)*

Аутомат се иницијално налази у стању s0. Стања су кодирана бинарно, s0="000", s1="001", ... s5="101" .



Слика 4.1


INV
INV
INV
INV
AND2
AND3
AND3
OR3
FD
D
Q
C
q0
AND3
AND4
OR2
FD
D
Q
C
q1
AND3
AND3
OR3
FD
D
Q
C
q2
clk

Слика 4.2

Решење:

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

entity fsm is
   port(
      clk   : in std_logic;
      reset : in std_logic;
      c     : in std_logic;
      q     : out std_logic_vector(2 downto 0)
   );
end fsm;

architecture rtl of fsm is

   type state_t is (s0, s1, s2, s3, s4, s5, s_error);
   attribute enum_encoding : string;
   attribute enum_encoding of state_t :
      type is "000 001 010 011 100 101 111";

   signal s : state_t;
   signal q_reg, q_next : std_logic_vector(2 downto 0);
   signal q0, q1, q2 : std_logic;

begin

   -- psl default clock is rising_edge(clk);
```

```
   -- psl P1 : assert always (s = s5) -> (next[2] (s =
s1));
   -- psl P2 : assert always (s = s0) -> (next[2] (s =
s2));
   -- psl P3a : assert always (s = s0) -> (next[5] (s =
s0));
   -- psl P3b : assert always {s = s0} |-> next[5] {s =
s0};
   -- psl P3c : assert always {s = s0} |=> {[*4];s = s0};
   -- psl P4 : assert always eventually! (s = s2);
   -- psl P5 : assert always eventually! (s = s3);

   process(clk, reset)
   begin
      if (reset = '1') then
         q_reg <= (others => '0');
      elsif rising_edge(clk) then
         q_reg <= q_next;
      end if;
   end process;

   process(q_next)
   begin
      case q_next is
         when "000" => s <= s0;
         when "001" => s <= s1;
         when "010" => s <= s2;
         when "011" => s <= s3;
         when "100" => s <= s4;
         when "101" => s <= s5;
```

```
         when others => s <= s_error;
      end case;
   end process;

   q0 <= q_reg(0);
   q1 <= q_reg(1);
   q2 <= q_reg(2);

   q_next(0) <= ((not q1) and (not q0)) or ((not q2) and q1
and q0) or
                ((not q2) and q1 and c);
   q_next(1) <= ((not q2) and (not q1) and q0) or ((not q2)
and q1 and
               not(q0) and c);
   q_next(2) <= ((not q2) and q1 and q0) or ((not q2) and
q1 and (not c)) or
                (q2 and (not q1) and (not q0));

   q <= q_reg;

end rtl;
```

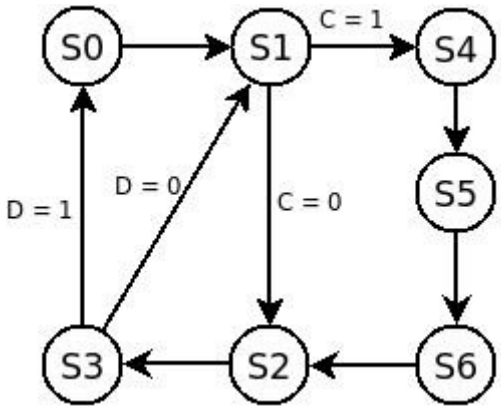
У **VHDL** датотеци, која садржи поставку овог проблема, је моделован разматрани аутомат. Обратите пажњу како се користи уведени сигнал **s** да би се у **PSL** изразима користили симболички називи стања **s0**,...,**s5**, уместо његових бинарних репрезентација.

Задатак 2.

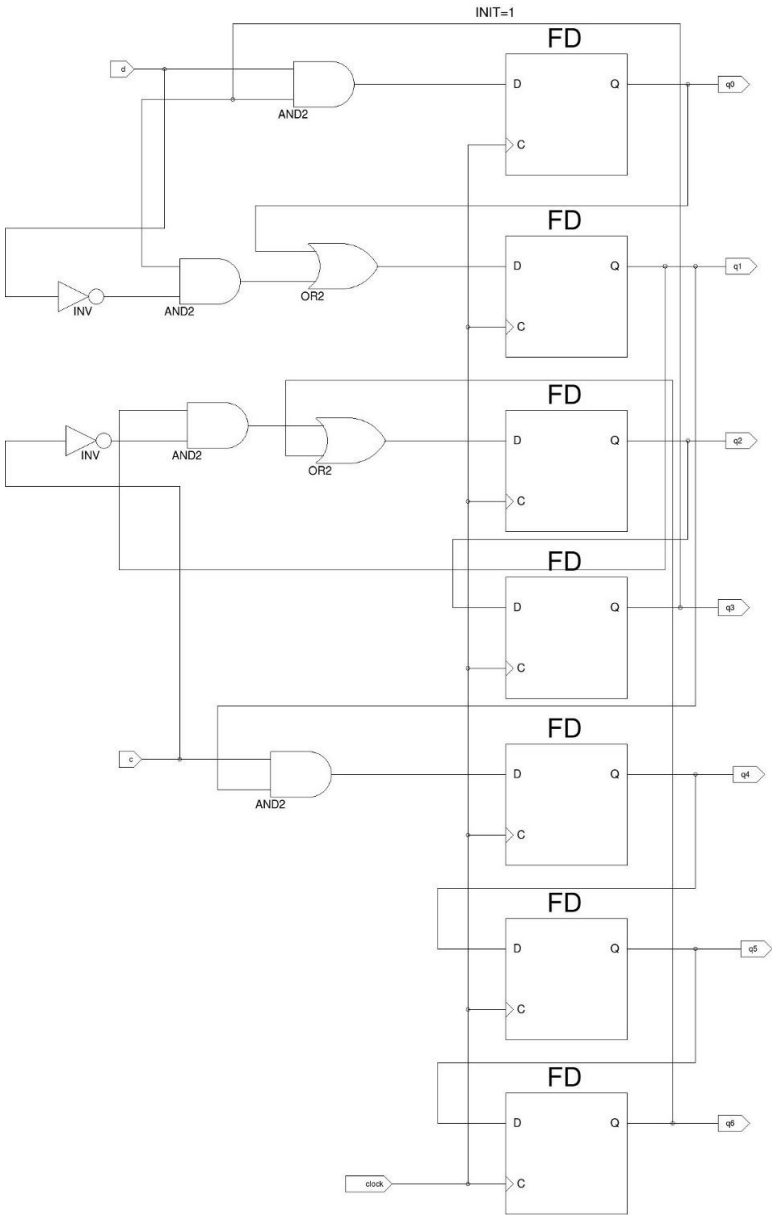
На слици 4.3 је приказан дијаграм прелаза стања Муровог аутомата, чија је шематска имплементација приказана на слици 4.4. Моделовати систем са слике 4.4 у **VHDL**-у, а затим коришћењем **IFV**-а испитати да ли за аутомат важе следеће особине:

- Аутомат у неком циклусу мора проћи кроз стање s2. *(true)*
- Аутомат у неком циклусу мора проћи кроз стање s4. *(false)*
- Два циклуса након стања s4 следи стање s6. *(true)*

Стања су кодирана "*one hot*" (s0 = 0000001, s1 = 0000010, ..., s6 = 1000000). Аутомат се иницијално налази у стању s0.



Слика 4.3



Слика 4.4

Задатак 3.

Коначни аутомат је моделован у **VHDL**-у:

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

entity fsm is
   port(
   clk : in std_logic;
   gen : in std_logic;
   reset : in std_logic;
   y : out std_logic
   );
end entity;

architecture rtl of fsm is
   type state_t is (idle, prepare, ready, high);
   signal state : state_t;
begin

   process (clk, reset)
   begin
       if reset = '1' then
           state <= idle;
           y <= '0';
       elsif rising_edge(clk) then
           y <= '0';
           case state is
               when idle => state <= prepare;
               when prepare => state <= ready;
               when ready =>
                   if gen = '1' then
                       state <= high;
                   else
                       state <= idle;
                   end if;
               when high =>
                   y <= '1';
                   state <= ready;
           end case;
       end if;
```

```
    end process;

end architecture;
```

Користећи **IFV** испитати да ли за овај аутомат важе следеће особине:

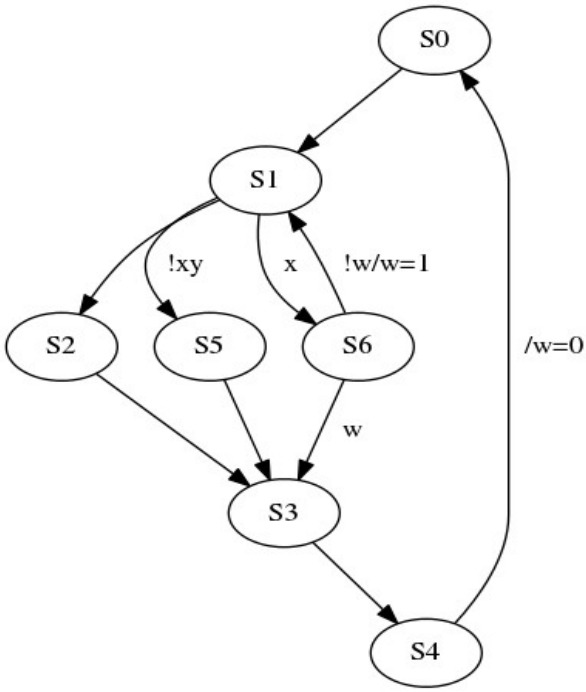
- Сигнал **y** се увек може активирати у неком будућем тренутку. *(true)*
- Из стања **idle** се након два циклуса прелази у стање **ready**. *(true)*
- Сигнал **y** може бити активан највише један циклус. *(true)*

Задатак 4.

Моделовати аутомат са слике и проверити следеће особине:

- Стање s3 је увек достижно.
- Могуће је да се исто стање понови после 2 циклуса.
- Након s1 могу следити s5, s2 и s6.
- Након s0 следи s1.
- Може се десити да w не постане 1.

Иницијално w је 0 а почетно стање је s0.



Слика 4.5

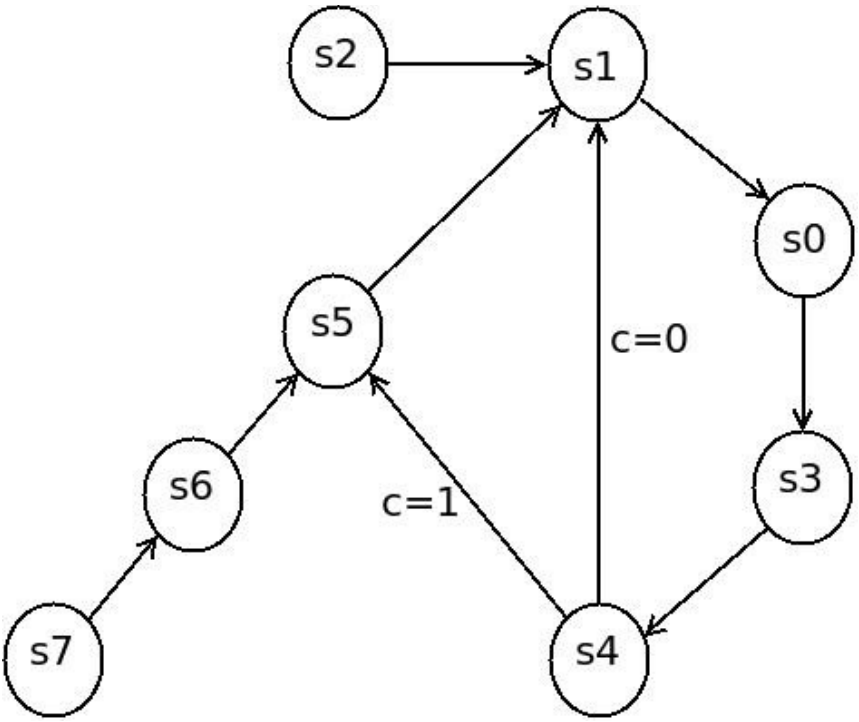
5. Верификација достижности стања аутомата

Приликом верификовања понашања секвенцијалних кола, од интереса је испитати да ли се из неког скупа полазних стања аутомата може достићи скуп неких других стања. Ова анализа се назива „анализа достижности стања **FSM**-а“ (енг. ***FSM*** *reachability analysis*), „анализа обиласка **FSM**-а“ (енг. ***FSM*** *traversal*) или „анализа покривености **FSM**-а“ (енг. ***FSM*** *coverage*). Наиме, код иоле сложенијих примера аутомата, лако се може десити да се кроз нека стања аутомата никада не прође, односно да се неке транзиције никада не десе, као и да након одређене секвенце стања и транзиција, аутомат уђе у тзв. „мртву петљу“ (енг. *dead-lock*). Ова вежба је посвећена анализи достижности стања аутомата.

Задатак 1. (решен)

На слици 5.1 је приказан дијаграм прелаза стања Муровог аутомата, чија је шематска имплементација приказана на слици 5.2. Моделовати систем са слике 5.2 у **VHDL**-у, а затим коришћењем **IFV**-а испитати да ли за аутомат важе следеће особине:

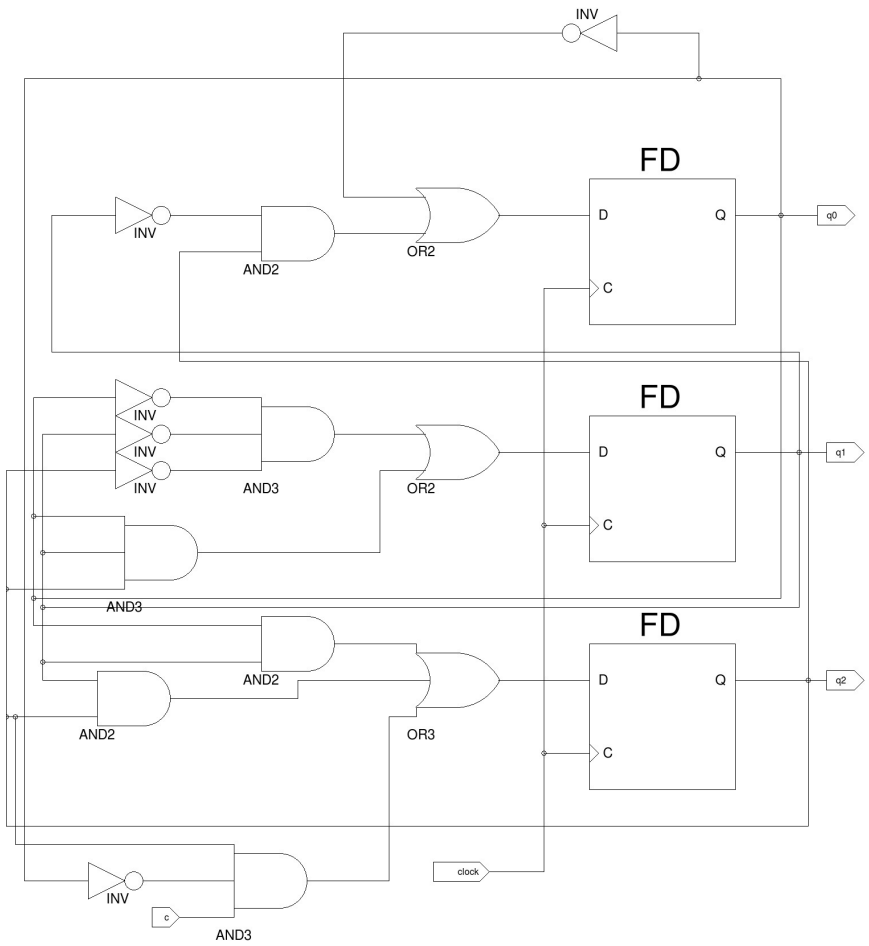
- Уколико се аутомат нађе у стању s3, у неком будућем тренутку ће се сигурно наћи и у стању s5. *(false)*
- Стање s0 је достижно из стања s2. *(true)*
- Стање s2 је достижно из стања s0. *(false)*
- Уколико се аутомат нађе у стању s7, у неком будућем тренутку ће се сигурно наћи и у стању s4. *(true)*



Слика 5.1

- Стање s5 је достижно из било ког стања. *(true)*

Стања су кодирана бинарно. Иницијално стање аутомата може бити свако стање.



Слика 5.2

Решење:

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

entity reach is
    port(
        clk   : in  std_logic;
        c     : in  std_logic;
        q     : out std_logic_vector(2 downto 0));
end reach;

architecture rtl of reach is

    constant s0 : std_logic_vector(2 downto 0) := "000";
    constant s1 : std_logic_vector(2 downto 0) := "001";
    constant s2 : std_logic_vector(2 downto 0) := "010";
    constant s3 : std_logic_vector(2 downto 0) := "011";
    constant s4 : std_logic_vector(2 downto 0) := "100";
    constant s5 : std_logic_vector(2 downto 0) := "101";
    constant s6 : std_logic_vector(2 downto 0) := "110";
    constant s7 : std_logic_vector(2 downto 0) := "111";

    signal q_reg, q_next : std_logic_vector(2 downto 0);
    signal q0, q1, q2    : std_logic;

begin

    -- psl default clock is rising_edge(clk);
```

```
    -- psl P1 : assert always (q_reg = s3) -> (eventually!
(q_reg = s5));
    -- psl P2 : cover {q_reg = s2; true[*]; q_reg = s0};
    -- psl P3 : cover {q_reg = s0; true[*]; q_reg = s2};
    -- psl P4 : assert always (q_reg = s7) -> (eventually!
(q_reg = s4));
    -- psl P5_s0 : cover {q_reg = s0; true[*]; q_reg = s5};
    -- psl P5_s1 : cover {q_reg = s1; true[*]; q_reg = s5};
    -- psl P5_s2 : cover {q_reg = s2; true[*]; q_reg = s5};
    -- psl P5_s3 : cover {q_reg = s3; true[*]; q_reg = s5};
    -- psl P5_s4 : cover {q_reg = s4; true[*]; q_reg = s5};
    -- psl P5_s6 : cover {q_reg = s6; true[*]; q_reg = s5};
    -- psl P5_s7 : cover {q_reg = s7; true[*]; q_reg = s5};


    process(clk)
    begin
        if rising_edge(clk) then
            q_reg <= q_next;
        end if;
    end process;


    q0 <= q_reg(0);
    q1 <= q_reg(1);
    q2 <= q_reg(2);


    q_next(0) <= (not q0) or (q2 and (not q1));
    q_next(1) <= ((not q2) and (not q1) and not(q0)) or (q2
and q1 and q0);
```

```
    q_next(2) <= (q1 and q0) or (q2 and q1) or (q2 and (not
q0) and c);


    q <= q_reg;


end rtl;
```

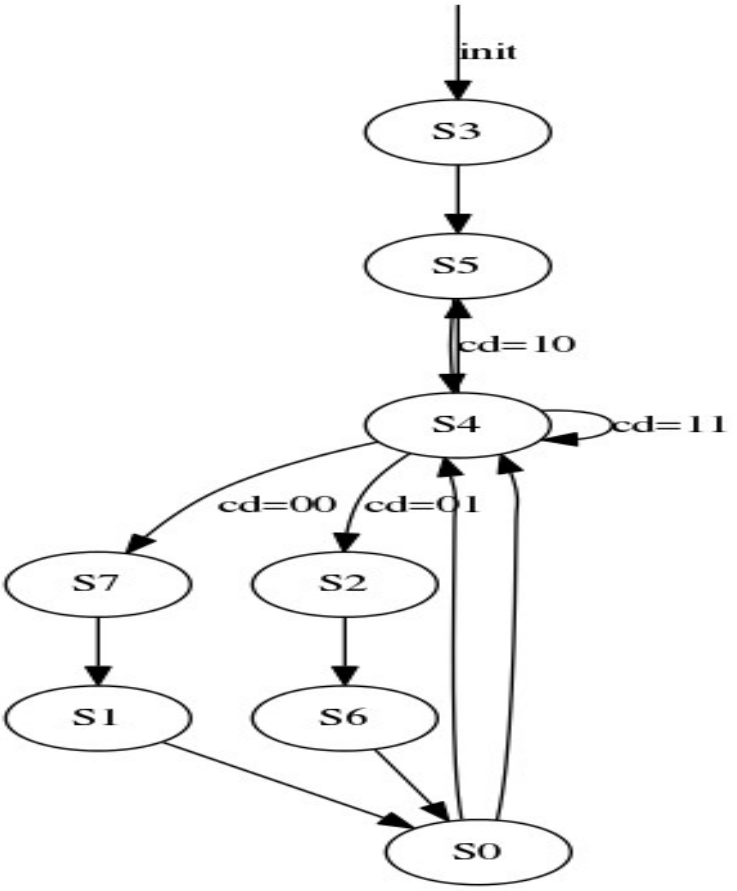
Да би се поставио упит о достижности стања, потребно је исказати ову особину у **CTL** логици, будући да се она не може исказати у **LTL** логици. Секвенца временских оператора **AG EF** коју следи жељена особина **prop** исказује особину достижности. У **PSL**-у су **CTL** искази подржани као „опциона екстензија за гранање (времена)“ - *Optional Branching Extension* (**OBE**). У тренутку изласка из штампе ове збирке задатака, **OBE** искази још увек нису подржани у **IFV**-у. Да би ипак проверили да ли се одређено стање достиже бар једном користићемо директиву **cover**. Треба нагласити да овакво решење није еквивалентно **AG EF CTL** упиту, будући да оно не исказује да ће се особина понављати изнова и изнова, већ барем једном.

У **VHDL** датотеци у којој је имплементирана инстанца задатог проблема (дизајн који се верификује), новина у односу на претходне примере је коришћење кључне речи **`cover`** уместо кључне речи **`assume`**. Када се користи кључна реч **`cover`**, тада се проверава да ли је особина која следи иза ње формално задовољена барем једном. На овај начин је могуће проверити да ли се нека секвенца сигнала уопште можу десити, што је и потребно урадити у овом задатку.

Задатак 2.

На слици 5.3 је приказан дијаграм прелаза стања Муровог аутомата, чија је шематска имплементација приказана на слици 5.4. Моделовати систем са слике 5.4 у **VHDL**-у, а затим коришћењем **IFV**-а испитати да ли за аутомат важе следеће особине:

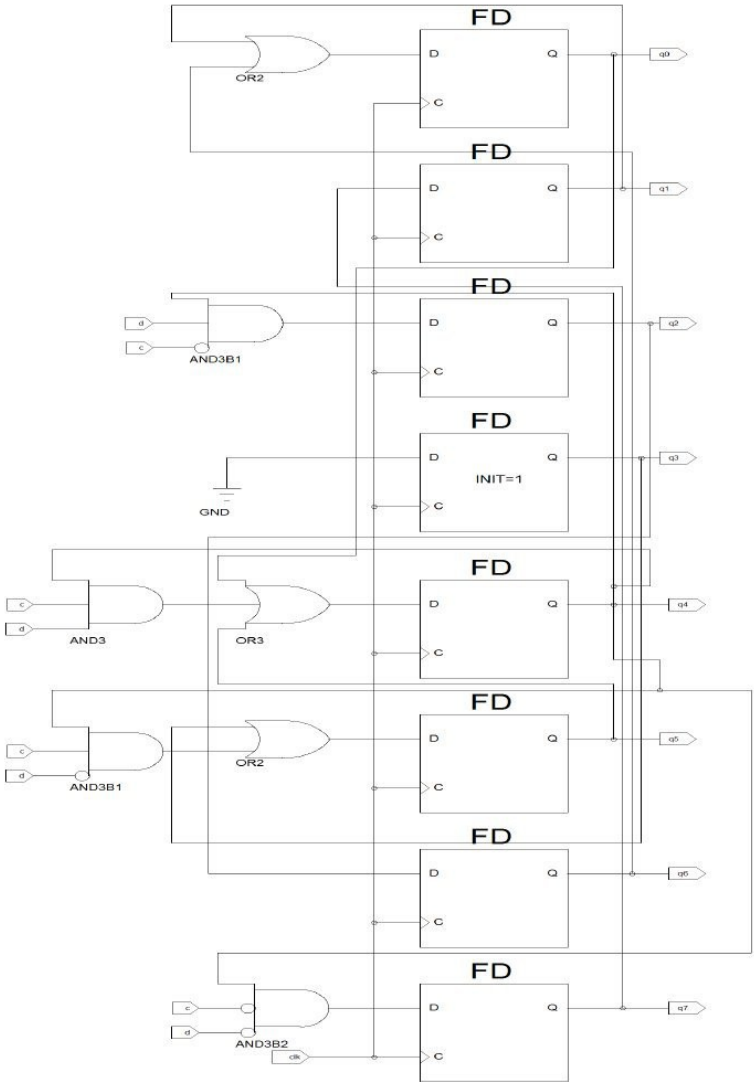
- Аутомат ће стално пролазити кроз стање s4. *(true)*



Слика 5.3

- Стање s2 је увек достижно. *(true)*
- Аутомат ће стално пролазити кроз стање s2. *(false)*
- Стање s3 је увек достижно. *(false)*
- Стање s3 је достижно. *(true)*

Стања треба кодирати са *one hot* кодом.


FD
D
Q
C
OR2
q0
FD
D
Q
C
q1
FD
d
c
AND3B1
D
Q
C
q2
FD
GND
D
Q
INIT=1
C
q3
FD
c
d
AND3
OR3
D
Q
C
q4
FD
c
d
AND3B1
OR2
D
Q
C
q5
FD
D
Q
C
q6
FD
c
d
AND3B2
clk
D
Q
C
q7

Слика 5.4

Задатак 3.

Коначни аутомат је моделован у **VHDL**-у:

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

entity fsm is
      port
      (
            clk : in std_logic;
            rst : in std_logic;
            long : in std_logic;
            y : out std_logic
      );
end entity;

architecture micro of fsm is

      constant IDLE  : std_logic_vector(1 downto 0) :=
"00";
      constant READY : std_logic_vector(1 downto 0) :=
"01";
      constant GEN   : std_logic_vector(1 downto 0) :=
"10";
      constant GEN2  : std_logic_vector(1 downto 0) :=
"11";

      signal adr_rom : std_logic_vector(1 downto 0);
      type ROM_TYPE is  array(3 downto 0) of
std_logic_vector(4 downto 0);
      signal instruction_rom : ROM_TYPE :=
      (
            GEN   & GEN   & '1',
            IDLE  & IDLE  & '1',
            GEN   & GEN2  & '0',
            READY & READY & '0'
      );

      alias state is adr_rom;
```

```
begin

      process(clk, rst)
      begin
            if rst = '1' then
                  adr_rom <= (others => '0');
            elsif rising_edge(clk) then
                  if long = '0' then
                        adr_rom <=
instruction_rom(to_integer(unsigned(adr_rom)))(4 downto 3);

                  else
                        adr_rom <=
instruction_rom(to_integer(unsigned(adr_rom)))(2 downto 1);

                  end if;
            end if;
      end process;

      y <= instruction_rom(to_integer(unsigned(adr_rom)))
(0);

end architecture micro;
```

За праћење тренутног стања система је уведен сигнал **`state`**. Користећи **IFV** испитати да ли за овај аутомат важе следеће особине:

- Стање GEN је увек достижно. *(true)*
- Стање GEN2 је увек достижно. *(true)*

Задатак 4.

Коначни аутомат је моделован следећим кодом:

```
module fsm
(
   input wire clk,
   input wire reset,
   input wire a, b, c,
   output reg y, z
);
   logic[3:0] cnt;
   enum {s0, s1, s2, s3, s4, s5, s6, s7, s8, s9, s10}
state;

   always @(posedge clk, posedge reset)
       if (reset) begin
           state <= s0;
           cnt <= 4'b0001;
       end
       else begin
           y <= 0;
           z <= 0;
           cnt <= { cnt[0] ^ cnt[1], cnt[3:1] };
           case (state)
           s0: begin
               if (a == 1)
                   state <= s1;
               else if (b == 1)
                   state <= s2;
               else if (c == 1)
                   state <= s3;
               else
                   state <= s0;
           end
           s1: begin
               y <= 1;
               state <= s4;
           end
           s2: begin
               z <= 1;
               state <= s5;
```

```
            end
            s3: state <= s6;
            s4:
                if (cnt == 0) state <= s7;
                else state <= s0;
            s5:
                if (cnt == 2) state <= s0;
                else  state <= s6;
            s6: state <= s8;
            s7: state <= s6;
            s8:
                if (b == 1)
                    state <= s9;
                else if (c == 1)
                    state <= s10;
                else state <= s0;
            s9: state <= s10;
            s10: state <= s8;
            endcase
        end
endmodule
```

Коришћењем формалне верификације, испитати следеће особине у **IFV**-у:

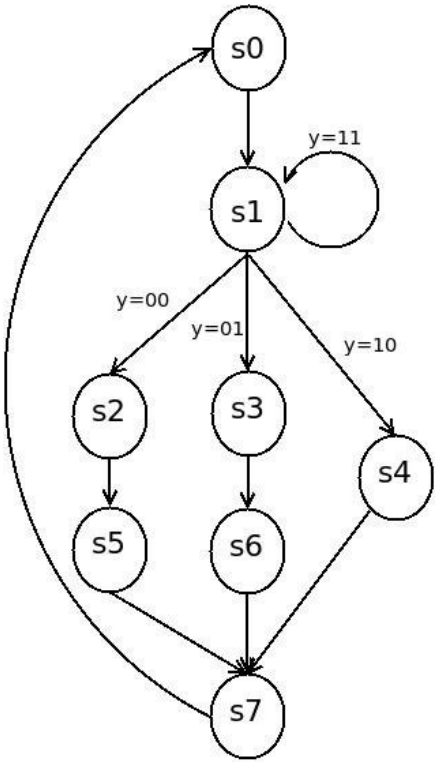
1. Да ли постоје недостижна стања у аутомату?
2. Да ли постоји прелаз из s5 у s0?
3. Да ли ће s0 бити достижно изнова и изнова?
4. Може ли се десити да аутомат од неког тренутка заувек остане у s0?

6. Откривање мртве петље у аутомату

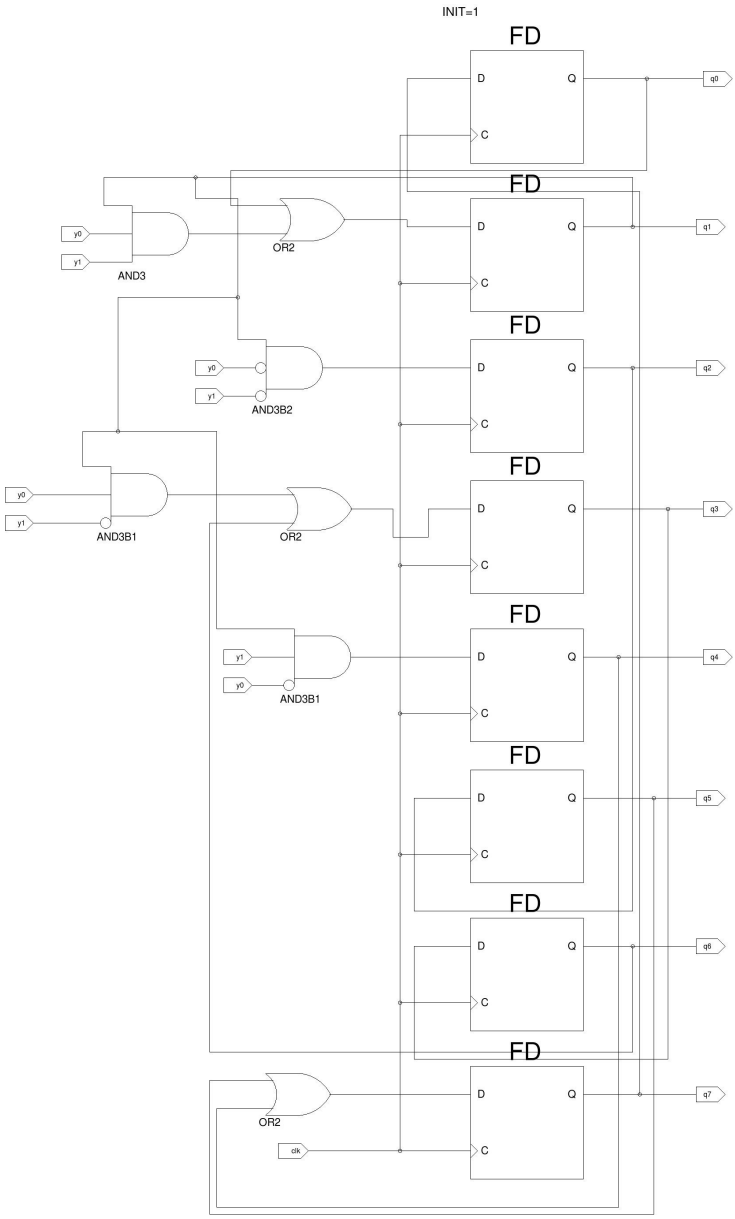
Ова вежба је посвећена откривању мртвих петљи у аутомату. Мртва петља је грешка при којој аутомат пролази стално исту секвенцу стања. Док је величина дизајна мала, оваква грешка се лако уочава и уклања, али како сложеност пројектованог система расте, решавање овог проблема постаје све комплексније и тада су методе формалне верификације веома корисне.

Задатак 1. (решен)

Имплементација коначног аутомата приказаног на слици 6.1, која је дата на слици 6.2, садржи грешку. Моделовати систем са слике 6.2 у **VHDL**-у, а затим коришћењем **IFV**-а испитати да ли у аутомату постоје мртве петље. Стања аутомата су кодирана „*one hot*“ а почетно стање је s0.



Слика 6.1


INIT=1
FD
D
Q
C
q0
q1
q2
q3
q4
q5
q6
q7
y0
y1
AND3
OR2
AND3B2
AND3B1
clk

Слика 6.2

Решење:

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

entity deadlock is
    port(
        clk : in  std_logic;
        rst : in  std_logic;
        y0  : in  std_logic;
        y1  : in  std_logic;
        q   : out std_logic_vector(7 downto 0));
end deadlock;

architecture rtl of deadlock is

    constant s0 : std_logic_vector(7 downto 0) := X"01";
    constant s1 : std_logic_vector(7 downto 0) := X"02";
    constant s2 : std_logic_vector(7 downto 0) := X"04";
    constant s3 : std_logic_vector(7 downto 0) := X"08";
    constant s4 : std_logic_vector(7 downto 0) := X"10";
    constant s5 : std_logic_vector(7 downto 0) := X"20";
    constant s6 : std_logic_vector(7 downto 0) := X"40";
    constant s7 : std_logic_vector(7 downto 0) := X"80";

    signal q_reg, q_next : std_logic_vector(7 downto 0);
    signal q0, q1, q2, q3, q4, q5, q6, q7 : std_logic;

begin
```

```
    -- psl default clock is rising_edge(clk);

    -- psl p1 : assert always (q = s0 or q = s1 or q = s2
or q =
    -- s3 or q = s4 or q = s5 or q = s6 or q = s7);

    -- psl p2_s0_1 : cover {q = s1; true[*]; q = s0};
    -- psl p2_s0_2 : cover {q = s2; true[*]; q = s0};
    -- psl p2_s0_3 : cover {q = s3; true[*]; q = s0};
    -- psl p2_s0_4 : cover {q = s4; true[*]; q = s0};
    -- psl p2_s0_5 : cover {q = s5; true[*]; q = s0};
    -- psl p2_s0_6 : cover {q = s6; true[*]; q = s0};
    -- psl p2_s0_7 : cover {q = s7; true[*]; q = s0};

    -- psl p2_s1_0 : cover {q = s0; true[*]; q = s1};
    -- psl p2_s1_2 : cover {q = s2; true[*]; q = s1};
    -- psl p2_s1_3 : cover {q = s3; true[*]; q = s1};
    -- psl p2_s1_4 : cover {q = s4; true[*]; q = s1};
    -- psl p2_s1_5 : cover {q = s5; true[*]; q = s1};
    -- psl p2_s1_6 : cover {q = s6; true[*]; q = s1};
    -- psl p2_s1_7 : cover {q = s7; true[*]; q = s1};

    -- psl p2_s2_0 : cover {q = s0; true[*]; q = s2};
    -- psl p2_s2_1 : cover {q = s1; true[*]; q = s2};
    -- psl p2_s2_3 : cover {q = s3; true[*]; q = s2};
    -- psl p2_s2_4 : cover {q = s4; true[*]; q = s2};
    -- psl p2_s2_5 : cover {q = s5; true[*]; q = s2};
    -- psl p2_s2_6 : cover {q = s6; true[*]; q = s2};
```

```
-- psl p2_s2_7 : cover {q = s7; true[*]; q = s2};

-- psl p2_s3_0 : cover {q = s0; true[*]; q = s3};
-- psl p2_s3_1 : cover {q = s1; true[*]; q = s3};
-- psl p2_s3_2 : cover {q = s2; true[*]; q = s3};
-- psl p2_s3_4 : cover {q = s4; true[*]; q = s3};
-- psl p2_s3_5 : cover {q = s5; true[*]; q = s3};
-- psl p2_s3_6 : cover {q = s6; true[*]; q = s3};
-- psl p2_s3_7 : cover {q = s7; true[*]; q = s3};

-- psl p2_s4_0 : cover {q = s0; true[*]; q = s4};
-- psl p2_s4_1 : cover {q = s1; true[*]; q = s4};
-- psl p2_s4_2 : cover {q = s2; true[*]; q = s4};
-- psl p2_s4_3 : cover {q = s3; true[*]; q = s4};
-- psl p2_s4_5 : cover {q = s5; true[*]; q = s4};
-- psl p2_s4_6 : cover {q = s6; true[*]; q = s4};
-- psl p2_s4_7 : cover {q = s7; true[*]; q = s4};

-- psl p2_s5_0 : cover {q = s0; true[*]; q = s5};
-- psl p2_s5_1 : cover {q = s1; true[*]; q = s5};
-- psl p2_s5_2 : cover {q = s2; true[*]; q = s5};
-- psl p2_s5_3 : cover {q = s3; true[*]; q = s5};
-- psl p2_s5_4 : cover {q = s4; true[*]; q = s5};
-- psl p2_s5_6 : cover {q = s6; true[*]; q = s5};
-- psl p2_s5_7 : cover {q = s7; true[*]; q = s5};

-- psl p2_s6_0 : cover {q = s0; true[*]; q = s6};
-- psl p2_s6_1 : cover {q = s1; true[*]; q = s6};
```

```
-- psl p2_s6_2 : cover {q = s2; true[*]; q = s6};
-- psl p2_s6_3 : cover {q = s3; true[*]; q = s6};
-- psl p2_s6_4 : cover {q = s4; true[*]; q = s6};
-- psl p2_s6_5 : cover {q = s5; true[*]; q = s6};
-- psl p2_s6_7 : cover {q = s7; true[*]; q = s6};

-- psl p2_s7_0 : cover {q = s0; true[*]; q = s7};
-- psl p2_s7_1 : cover {q = s1; true[*]; q = s7};
-- psl p2_s7_2 : cover {q = s2; true[*]; q = s7};
-- psl p2_s7_3 : cover {q = s3; true[*]; q = s7};
-- psl p2_s7_4 : cover {q = s4; true[*]; q = s7};
-- psl p2_s7_5 : cover {q = s5; true[*]; q = s7};
-- psl p2_s7_6 : cover {q = s6; true[*]; q = s7};

process(clk)
begin
    if rising_edge(clk) then
        if rst = '1' then
            q_reg <= s0;
        else
            q_reg <= q_next;
        end if;
    end if;
end process;

q0 <= q_reg(0);
q1 <= q_reg(1);
q2 <= q_reg(2);
```

```
    q3 <= q_reg(3);
    q4 <= q_reg(4);
    q5 <= q_reg(5);
    q6 <= q_reg(6);
    q7 <= q_reg(7);

    q_next(0) <= q7;
    q_next(1) <= q0 or (q1 and y0 and y1);
    q_next(2) <= q1 and (not y0) and (not y1);
    q_next(3) <= (q1 and y0 and (not y1)) or q6;
    q_next(4) <= q1 and (not y0) and y1;
    q_next(5) <= q2;
    q_next(6) <= q3;
    q_next(7) <= q4 or q5;

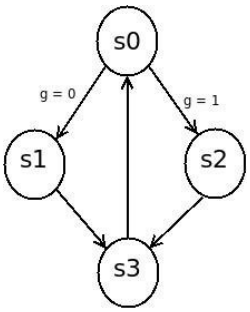
    q <= q_reg;

end rtl;
```

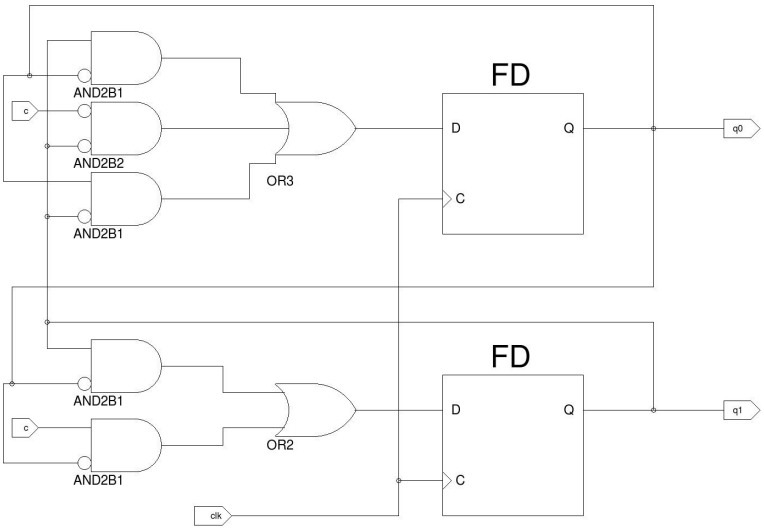
Један од начина да утврдимо да ли у аутомату постоје мртве петље, је да испитамо да ли је свако стање у сваком циклусу увек достижно у будућности. Уколико су само нека стања увек достижна, док друга нису, то значи да у аутомату постоји мртва петља. Мртву петљу у том случају сачињавају стања која су увек достижна. Постављањем скупа **cover** директива, у овом примеру се показује да су само два стања увек достижна, s3 и s6. То значи да у аутомату са слике 6.2. постоји мртва петља коју сачињавају стања s3 и s6, одакле закључујемо да аутомат са слике 6.2 није коректна имплементација описа аутомата задатог на слици 6.1.

Задатак 2.

Имплементација коначног аутомата приказаног на слици 6.3, која је дата на слици 6.4, садржи грешку. Моделовати систем са слике 6.2 у **VHDL**-у, а затим коришћењем **IFV**-а испитати да ли у аутомату постоје мртве петље. Стања аутомата су кодирана бинарно. Почетно стање је s0. (Постоји мртва петља: s1)



Слика 6.3



Слика 6.4

Задатак 3.

Коначни аутомат је моделован у *Verilog*-у:

```
module fsm
(
      input wire clk,
      input wire reset,
      input wire transite,
      output wire y
);

localparam [1:0]
      idle = 2'b00,
      op1 = 2'b01,
      op2 = 2'b10,
      op3 = 2'b11;

reg [1:0] state;
reg [3:0] lcnt;

always @(posedge clk, posedge reset)
      if (reset)
      begin
            state <= idle;
            lcnt <= 4'b0001;
      end
      else
      begin
            case (state)
            idle:
              if (transite)
                state <= op1;
            op1:
              if (transite)
                state <= op2;
              else
                state <= op3;
            op2:
              if (lcnt == 4'b0000)
                state <= idle;
            op3:
              if (lcnt == 4'b1111)
```

```
                state <= idle;
            endcase
            // conectation
            // VHDL : lcnt <= (lcnt(3) xor lcnt(0)) &
lcnt(3 downto 1);
            lcnt <= {lcnt[3] ^ lcnt[0], lcnt[3:1]};
        end

   assign y = (state == op1) | (state == op2 & transite);

endmodule
```

Користећи **IFV** пронаћи мртву петљу у аутомату.

Задатак 4.

Моделовати аутомат са следећим особинама:

- Постоји 6 стања, означених са s0, s1, s2, s3, s4 и s5.
- s0 је почетно стање, и аутомат у току свог рада не пролази опет кроз њега.
- Постоје 2 улаза у аутомат, означена са a и b, која су међусобно искучива.
- Уколико је a активан, аутомат остаје у тренутном стању.
- Уколико је b активан, парови могућих транзиција аутомата су прелази из тренутног стања s_n у наредно s_{n+1}, n=1..4, док је за s5 наредно стање s1. Уз то, ове транзиције ће се стално понављати (на пример, уколико је тренутно стање s5, секвенца стања која ће уследити је s5 –> s1 –> s5 –> s1 –> s5 –> s1... слично, уколико је тренутно стање s2, секвенца стања која ће уследити је s2 –> s3 –> s2 –> s3 –> s2 –> s3...). Ове транзиције престају када се b деактивира.
- Када су оба сигнала 0, тада аутомат мења стања на следећи начин: s1 –> s2 –> s3 –> s4 –> s5 –> s1... .

У моделованом аутомату, формално верификовати следеће особине:

- Да ли постоји могућност да аутомат неограничено дуго остане у истом стању?
- Да ли постоји могућност да аутомат неограничено дуго „скакуће“ између два стања?

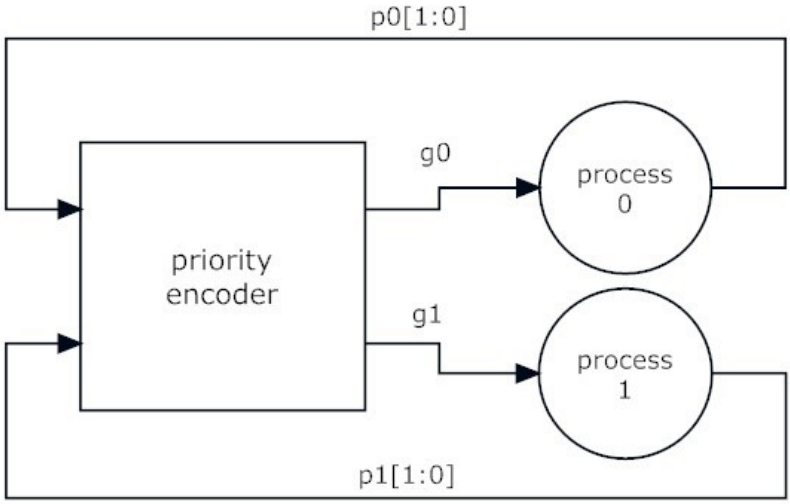
7. Распоређивач синхроних процеса

У раду дигиталних система, често се јавља ситуација да више различитих функционалних јединица дели исте ресурсе, којима не би смело да се приступа истовремено. На пример, више процеса не може истовремено приступити дељеној меморији за уписивање или очитавање података. Слична ситуација се јавља и у синхронизацији софтверских процеса, када нпр. не сме бити дозвољено истовремено писање или читање у исту датотеку. Примери ових ситуација су многобројни: заузеће магистрале, заузеће процесора итд. У том случају потребно је контролисати приступ дељеном критичном ресурсу. Овај проблем се назива „међусобно искључивање“ (енг. *Mutual exclusion*, скраћено *mutex*). За решавање овог проблема постоје многобројне технике и алгоритми који се могу имплементирати било у хардверу било у софтверу. Неки од њих су критичне секције, тзв. *spin-lock*, *busy-wait*, монитори, семафори итд.

У овој вежби, за решавање проблема међусобног искључивања два или више процеса је искоришћен једноставан енкодер приоритета (енг. *priority encoder*).

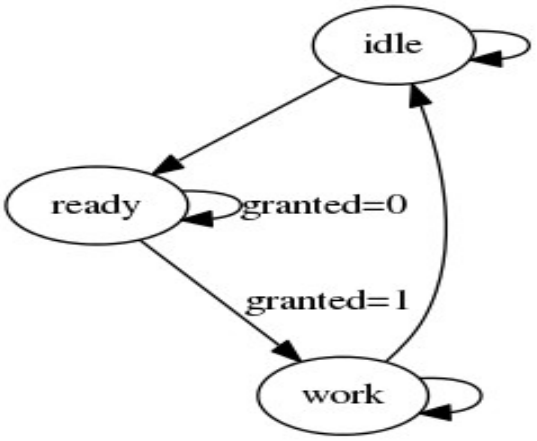
Задатак 1. (решен)

На слици 7.1 је приказано коло које служи за синхронизацију рада два процеса, *process0* и *process1*. Ови процеси могу прелазити из стања у стање само како је приказано на слици 7.2.



Слика 7.1

Понашање процеса описано на слици 7.2 је у **VHDL**-у моделовано на следећи начин:



Слика 7.2

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

entity proc is
    port(
        rst     : in  std_logic;
        clk     : in  std_logic;
        granted : in  std_logic;
        switch  : in  std_logic;
        p       : out std_logic_vector(1 downto 0));
end proc;

architecture rtl of proc is

    subtype state_t is std_logic_vector(1 downto 0);

    constant s_idle  : state_t := "00";
    constant s_ready : state_t := "01";
    constant s_work  : state_t := "10";

    signal state_reg, state_next : state_t;
begin

    process(clk, rst)
    begin
        if (rst = '1') then
            state_reg <= s_idle;
        elsif rising_edge(clk) then
```

```
            state_reg <= state_next;
        end if;
    end process;

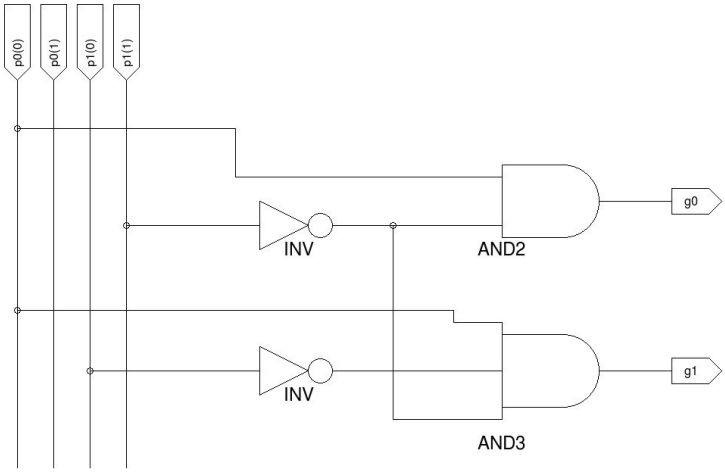
    next_state : process(granted, state_reg, switch)
    begin
        if (state_reg = s_idle) then
            if (switch = '1') then
                state_next <= s_idle;
            else
                state_next <= s_ready;
            end if;
        elsif (state_reg = s_ready and (granted = '1'))
then
            state_next <= s_work;
        elsif (state_reg = s_ready and (granted = '0'))
then
            state_next <= s_ready;
        elsif (state_reg = s_work) then
            if (switch = '1') then
                state_next <= s_idle;
            else
                state_next <= s_work;
            end if;
        else
            state_next <= s_idle;
        end if;
    end process next_state;
```

```
    p <= state_reg;

end rtl;
```

На слици 7.3 је дата једноставна хардверска имплементација *priority encoder*-а са слике 7.1. У имплементацији су стања процеса кодирана на следећи начин: *idle* – 00, *ready* – 01 и *work* – 10. Улазне двобитне речи **`p0`** и **`p1`** садрже кодове тренутних стања процеса *process0* и *process1*. У зависности од тренутних стања оба процеса, *priority encoder* даје дозволу за прелазак у стање *work* једном или другом процесу, тако што активира одговарајући **`g`** излаз. Користећи **IFV** испитати следеће особине:

- Да ли се оба процеса могу извршавати истовремено?
- Ако су оба процеса у стању *ready*, који има предност?



Слика 7.3

Решење:

Модел енкодера

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

entity enc is
   port(
      p : in std_logic_vector(3 downto 0);
      g : out std_logic_vector(1 downto 0)
   );
end enc;

architecture rtl of enc is
begin

   g(0) <= (not p(1)) and p(0) and (not p(3));
   g(1) <= (not p(1)) and  (not p(0)) and (not p(3)) and
p(2);

end rtl;
```

Модел за верификацију

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

entity scheduler is
    port (
        clk : in std_logic;
        rst : in std_logic;
        c   : in std_logic_vector(1 downto 0));
```

```
end scheduler;

architecture rtl of scheduler is

    signal g0, g1 : std_logic;
    signal p0, p1 : std_logic_vector(1 downto 0);

    subtype stanja is std_logic_vector(1 downto 0);

    constant s_idle  : stanja := "00";
    constant s_ready : stanja := "01";
    constant s_work  : stanja := "10";

    signal proc0_state, proc1_state : std_logic_vector(1
downto 0);

begin

    -- psl default clock is rising_edge(clk);

    -- psl P1p : assert always (proc0_state = s_work) ->
(proc1_state /= s_work);
    -- psl P2 : assert always (proc1_state = s_work) ->
(proc0_state /= s_work);

    -- psl P3 : assert always ((proc0_state = s_ready) and
(proc1_state = s_ready)) -> (next (proc0_state = s_work));
    -- psl P4 : assert always ((proc0_state = s_ready) and
(proc1_state = s_ready)) -> (next (proc1_state = s_work));
```

```
    proces0 : entity work.proc
        port map(
            rst     => rst,
            clk     => clk,
            granted => g0,
            switch  => c(0),
            p       => p0);

    proces1 : entity work.proc
        port map(
            rst     => rst,
            clk     => clk,
            granted => g1,
            switch  => c(1),
            p       => p1);

    priority_encoder : entity work.enc
        port map(
            p(3 downto 2) => p1,
            p(1 downto 0) => p0,
            g(1)          => g1,
            g(0)          => g0);

    proc0_state <= p0;
    proc1_state <= p1;

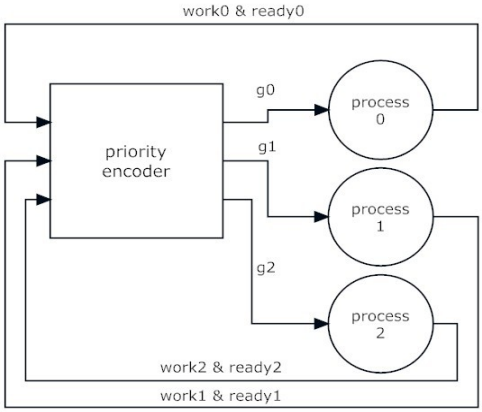
end rtl;
```

Помоћу прве две специфициране особине, испитујемо исправност рада *priority encoder*-а, тј. немогућност појављивања ситуације у којој су истовремено оба

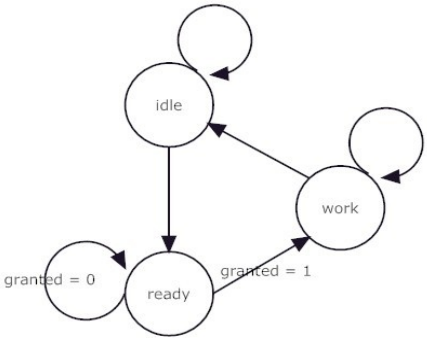
процеса у **`work`** стању. Помоћу преостале две специфициране особине, утврђујемо који од процеса има виши приоритет, будући да не могу истовремено да буду тачне обе формуле.

Задатак 2.

На слици 7.4 је приказано коло које служи за синхронизацију рада три процеса, *process0*, *process1* и *process2*. Процеси *process0*, *process1* и *process2* могу прелазити из стања у стање само како је приказано на слици 7.5.



Слика 7.4



Слика 7.5

Понашање процеса описано на слици 7.5 је у **VHDL**-у моделовано на бихевиоралном нивоу на следећи начин:

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

```
entity proc is
    port(
        reset   : in  std_logic;
        clk     : in  std_logic;
        granted : in  std_logic;
        rdy     : out std_logic;
        wrk     : out std_logic);
end proc;

architecture rtl of proc is

    subtype state_t is std_logic_vector(1 downto 0);

    constant s_idle  : state_t := "00";
    constant s_ready : state_t := "01";
    constant s_work  : state_t := "10";

    signal state_reg, state_next : state_t;

    signal pom : unsigned(1 downto 0);

begin

    process(clk, reset)
    begin
        if (reset = '1') then
            state_reg <= s_idle;
```

```
            pom       <= "00";
        elsif rising_edge(clk) then
            state_reg <= state_next;
            pom       <= pom + "01";
        end if;
    end process;

    next_state : process(granted, pom, state_reg)
    begin
        if (state_reg = s_idle) then
            if (pom = "00" or pom = "10") then
                state_next <= s_idle;
            else
                state_next <= s_ready;
            end if;
        elsif (state_reg = s_ready and (granted = '1'))
then
            state_next <= s_work;
        elsif (state_reg = s_ready and (granted = '0'))
then
            state_next <= s_ready;
        elsif (state_reg = s_work) then
            if (pom = "01" or pom = "10") then
                state_next <= s_idle;
            else
                state_next <= s_work;
            end if;
        else
            state_next <= s_idle;
```

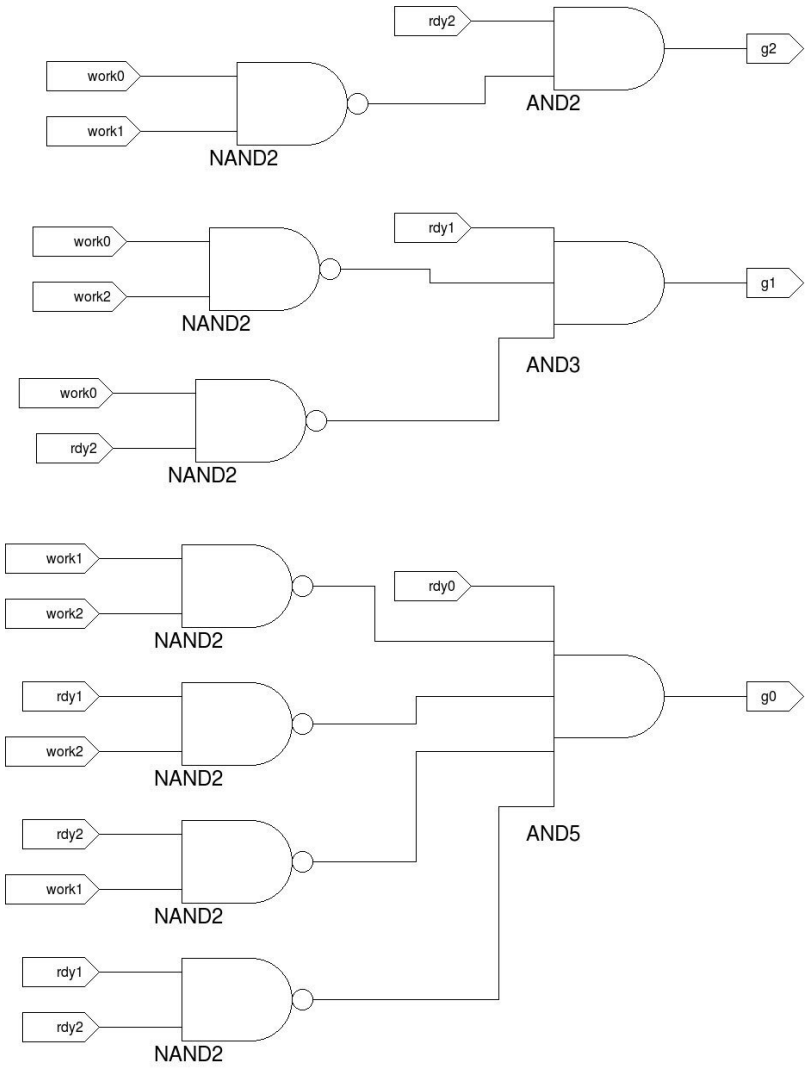
```
        end if;
    end process next_state;

    wrk <= '1' when state_reg = s_work  else '0';
    rdy <= '1' when state_reg = s_ready else '0';

end rtl;
```

У приказаном моделу који је исти за сва три процеса, показано је и како процеси активирају своје **work**$_i$ и **ready**$_i$ сигнале.

На слици 7.6 је дата хардверска имплементација *priority encoder*-а са слике 7.4. У зависности од тренутних вредности $work_i$ и $ready_i$ сигнала, *priority encoder* даје дозволу за прелазак у стање *work* неком од процеса, тако што активира одговарајући g излаз.



Слика 7.6

Користећи **IFV** испитати да ли се сва три процеса могу извршавати истовремено.

Задатак 3.

Дигитално коло **arbiter** моделовано је у *Verilog*-у на следећи начин:

```
module arbiter(
              input wire rst,
              input wire clk,
              input wire [2:0] req,
              input wire [2:0] free,
              output reg [2:0] granted,
              output reg [2:0] slaved
              );

  reg [2:0]                    g;
  reg [2:0]                    s;


  always@(posedge clk, posedge rst)
    if (rst)
      begin
        granted <= 3'b000;
        slaved <= 3'b000;
      end
    else
      begin
        g = 3'b000;
        s = 3'b000;
        if (req[0])
          begin
             if (free[1])
             begin
                g[0] = 1;
                s[1] = 1;
             end
             else if (free[0])
             begin
                g[0] = 1;
                s[0] = 1;
             end
```

```
        end // if (req[0])
      if(req[1])
        begin
           if(free[2])
           begin
              g[1] = 1;
              s[2] = 1;
           end
           else if(free[1] & s[1] == 0)
           begin
              g[1] = 1;
              s[1] = 1;
           end
        end // if (req[1])
      if (req[2])
        begin
           if (free[0] & s[0] == 0)
           begin
              g[2] = 1;
              s[0] = 1;
           end
           else if (free[2] & s[2] == 0)
           begin
              g[2] = 1;
              s[2] =1;
           end
        end // if (req[2])
      granted <= g;
      slaved <= s;
    end

endmodule // arbiter
```

Коло **arbiter** дозвољава приступ трима процесима да заузму три функционално иста ресурса. Сигнал **req** представља захтев процеса за ресурсом. Сигнал **free** означава да ли је одговарајући ресурс слободан. Сигнал **granted** је дозвола приступа ресурсу за одговарајући процес, док **slaved** садржи информацију о томе који је ресурс додељен ком процесу. Коло **arbiter** задовољава наредне особине, које треба верификовати уз помоћ **IFV**-а:

- Сва три процеса могу истовремено заузети сва три (различита) ресурса.
- Ако су слободни 0. и 1. ресурс и постоји захтев за ресурсом од стране процеса 0, тада ће процес 0 добити приступ.

- Ако су слободни 1. и 2. ресурс и постоји захтев за ресурсом од стране процеса 1, тада ће процес 1 добити приступ.
- Ако су слободни 0. и 2. ресурс и постоји захтев за ресурсом од стране процеса 2, тада процес 2 може добити приступ.
- Ако постоји само захтев процеса 0 за ресурсом, тада ресурс 2 неће бити додељен.
- Ако постоји само захтев процеса 1 за ресурсом, тада ресурс 0 неће бити додељен.
- Ако постоји само захтев процеса 2 за ресурсом, тада ресурс 1 неће бити додељен.

Задатак 4.

Моделовати меморију 16x4 бита, три процеса који јој приступају и модул који синхронизује рад процеса и меморије. Коришћена меморија има два излаза за податке и два адресна улаза, који одређују који подаци ће се истовремено очитати (тзв. *dual port memory*). Уз то постоји један улаз за упис података, одговарајући адресни порт и сигнал који допушта упис у меморију.

Процеси имају 4 стања, **`idle`**, **`ready_w`**, **`ready_r`** и **`work`**. У стању **`idle`**, не постоје захтеви процеса за меморијом. У сваком циклусу, процеси могу прећи у **`ready_w`** стање уколико захтевају писање, или у **`ready_r`** стање уколико захтевају читање. Сви процеси имају портове за адресу и податке, као и додатни сигнал који означава да ли је меморија расположива за обраду. Када добију приступ меморији, процеси прелазе у **`work`** стање, у којем остају од 2 до 5 циклуса.

Модул за синхронизацију треба да омогући рад ова три процеса се једним меморијским 16x4 модулом. Написати формалне упите који ће верификовати исправност рада овог система.

8. *Safety*, *Liveness* и *Fairness* особине система

Приликом формалне верификације хардвера, типичне особине које је пожељно да системи поседују су: а) да се нешто (лоше) никада неће десити, б) да ће се нешто (добро) десити барем једанпут и ц) да ће се нешто (добро) дешавати изнова и изнова (бесконачно много пута). У области формалне верификације, у литератури на енглеском језику ове особине се респективно називају *Safety*, *Liveness* и *Fairness*. Особине *Safety* и *Liveness* се у **CTL** логици исказују у виду следећих формула: $\mathbf{AG}\neg\phi$ (особина *Safety*) и $\mathbf{AG}(\phi \Rightarrow \mathbf{AF}\psi)$ (особина *Liveness*). Особина *Fairness* се не може директно исказати у **CTL** логици, али се може исказати у логици **LTL** у виду формуле $\mathbf{GF}\phi$.

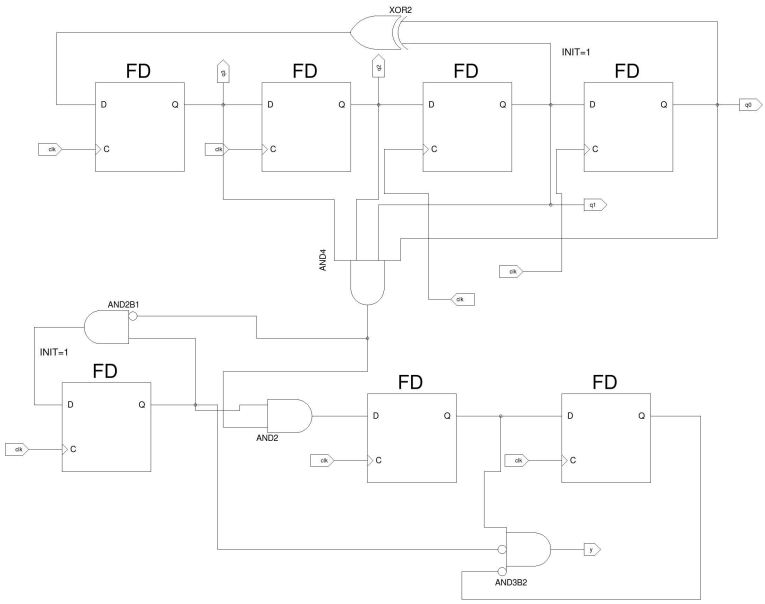
У овој вежби се испитују наведене три особине на примерима рада различитих дигиталних система.

Задатак 1. (решен)

За дигитални систем са слике 8.1 испитати следеће особине:

- Сигнали **q0**, **q1**, **q2** и **q3** никада нису сви истовремено неактивни (*Safety*).
- Сигнал **y** ће постати активан у неком тренутку у будућности (*Liveness*).
- Сигнал **y** ће изнова и изнова бивати активан (*Fairness*).
- Сигнали **q0**, **q1**, **q2** и **q3** ће изнова и изнова бивати сви истовремено активни (*Fairness*).

Иницијално, **q0** и **d0** су активни, а вредности уписане у осталим регистрима (**q1**, **q2**, **q3**, **d1** и **d2**) су 0.



Слика 8.1

Решење:

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

entity slf is
    port(
        rst : in  std_logic;
        clk : in  std_logic;
        q   : out std_logic_vector(3 downto 0);
        y   : out std_logic);
end slf;

architecture rtl of slf is

    signal q_reg, q_next : std_logic_vector(3 downto 0);
    signal d_reg, d_next : std_logic_vector(2 downto 0);

begin

    -- psl default clock is rising_edge(clk);

    -- Safety
    -- psl P1 : assert always (q /= "0000");
    -- Liveness
    -- psl P2 : assert eventually! (y = '1');
    -- Fairness
    -- psl P3 : assert always eventually! (y = '1');
    -- Fairness
```

```
    -- psl P4 : assert always eventually! (q = "1111");

    syn_ff : process(clk, rst)
    begin
        if (rst = '1') then
            q_reg <= "0001";
            d_reg <= "001";
        elsif rising_edge(clk) then
            q_reg <= q_next;
            d_reg <= d_next;
        end if;
    end process syn_ff;

    q_next(0) <= q_reg(1);
    q_next(1) <= q_reg(2);
    q_next(2) <= q_reg(3);
    q_next(3) <= q_reg(0) xor q_reg(1);

    d_next(0) <= d_reg(0) and (not (q_reg(0) and q_reg(1)
and q_reg(2) and q_reg(3)));
    d_next(1) <= q_reg(0) and q_reg(1) and q_reg(2) and
q_reg(3) and d_reg(0);
    d_next(2) <= d_reg(1);

    q <= q_reg;
    y <= (not d_reg(0)) and d_reg(1) and (not d_reg(2));

end rtl;
```

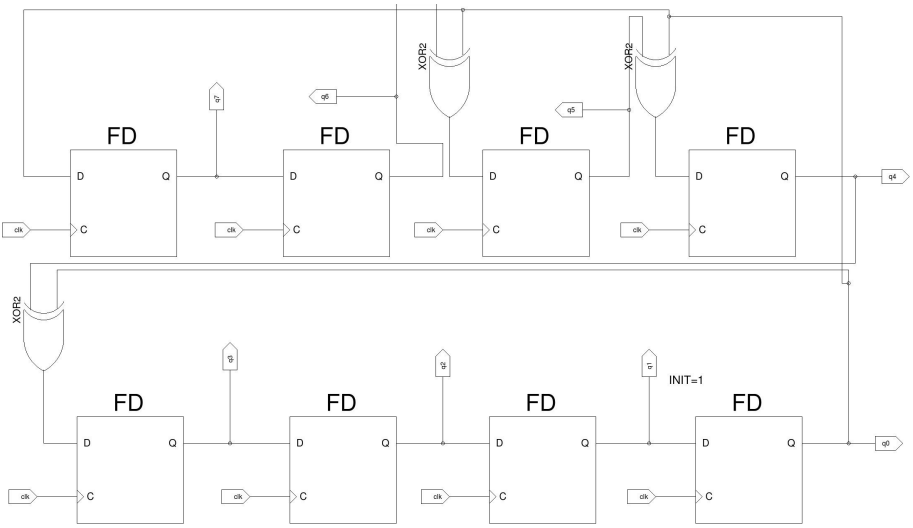
У овом примеру можемо видети случај у коме сигнал **`y`** задовољава особину *liveness*, али не задовољава особину *fairness*. Другим речима, сигнал **`y`** ће тачно једном постати активан, али се то неће дешавати изнова и изнова.

Задатак 2.

За дигитални систем са слике 8.2 испитати следеће особине:

- Сигнали **q0, q1, q2, q3, q4, q5, q6** и **q7** никада нису сви истовремено неактивни.
- Сигнали **q0, q1, q2, q3, q4, q5, q6** и **q7** ће изнова и изнова постајати сви истовремено активни.

Иницијално, регистар **q0** је постављен на 1, а остали регистри на 0.

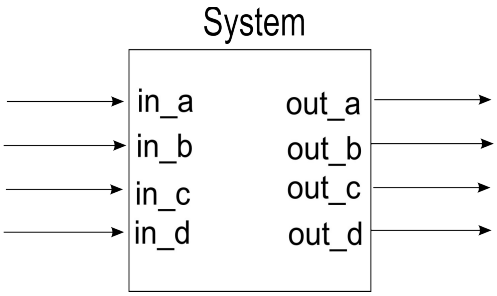


Слика 8.2

Задатак 3.

Дигитални систем са слике 8.3 треба да испуњава следеће специфициране захтеве:

- Сигнали out_a и out_b никада нису истовремено 1. (*Safety*)
- Сигнал out_a ће стално постајати 1. (*Fairness*)
- Сигнал out_c ће бар једном постати 1. (*Liveness*)
- Сигнал out_c не сме постајати 1 непрестано. (*Safety*)
- Када in_a постане 1 у наредном циклусу out_a је 0.
- Сигнал out_d не сме бити 1 више од једног циклуса. (*Safety*)
- Када се in_b и in_c истовремено активирају, out_c може бити 0 неограничено дуго.
- Уколико је in_d активан, out_c не сме постати 1, док out_d треба да пулсира са периодом 1. (*Safety*)



Слика 8.3

Дизајнирати систем у неком од **HDL** језика, формално специфицирати особине система у **PSL**-у и формално их доказати у **IFV**-у.