Sekvencijalne naredbe

- Za razliku od konkurentnog koda, u sekvencijalnom kodu, baš kao i kod tradicionalnih programskih jezika, naredbe se izvršavaju po redosledu kako su napisane.
- Konkurentne i sekvencijalne naredbe se ne mogu ″mešati″ u kodu arhitekture, već se sekvencijalne sekcije koda mogu pisati samo u okviru posebnih jezičkih konstrukcija: **procesa**, funkcija i procedura.
- Proces je, sam za sebe, kao celina, konkurentna naredba i kao takva može se kombinovati s drugim konkurentnim naredbama.
- Funkcije i procedure su potprogrami i mogu se pozivati iz drugih delova koda, bilo konkurentnog, bilo sekvencijalnog.

Sekvencijalne naredbe

Sekvencijalni kod je sadržan u:

- Procesu (PROCESS)
- Procedure (PROCEDURE)
- Funkciji (FUNCTION)

Sekvencijalne naredbe:

- IF
- WAIT
- CASE
- LOOP

PROCESS

- Okvir za sekvencijalni kod - definiše oblast arhitekture u kojoj se naredbe izvršavaju na sekvencijalan način.
- Kao celina, proces je konkurentna naredba i može se kombinovati s drugim konkurentnim naredbama i procesima u okviru iste arhitekture.
- U odnosu na to kako se definišu uslovi za aktiviranje, razlikujemo dve vrste procesa:

1. proces sa listom senzitivnosti i
2. proces sa *wait* naredbom.

Proces sa listom senzitivnosti

- Sintaksa:

[labela:] PROCESS (lista senzitivnosti)

[VARIABLE ime tip [opseg] [:= inicijalna_vrednost]]

BEGIN

(sekvencijalni kod)

END PROCESS [labela];

- *Lista senzitivnosti*: spisak signala na koje je proces senzitivan (osetljiv).
- **Promena vrednosti bilo kog signala iz liste senzitivnosti aktivira process.**
- Nakon aktiviranja, naredbe iz dela *sekvencijalni kod* se izvršavaju jedna za drugom.
- Nakon što se izvrši i poslednja naredba, proces se deaktivira, i ostaje neaktivan sve do ponovnog aktiviranja.

Proces sa listom senzitivnosti – opis kombinacionog kola

```
SIGNAL a, b, c : STD_LOGIC;
. . .
PROCESS(a, b, c)
BEGIN
y <= a OR b OR c;
END PROCESS;
```

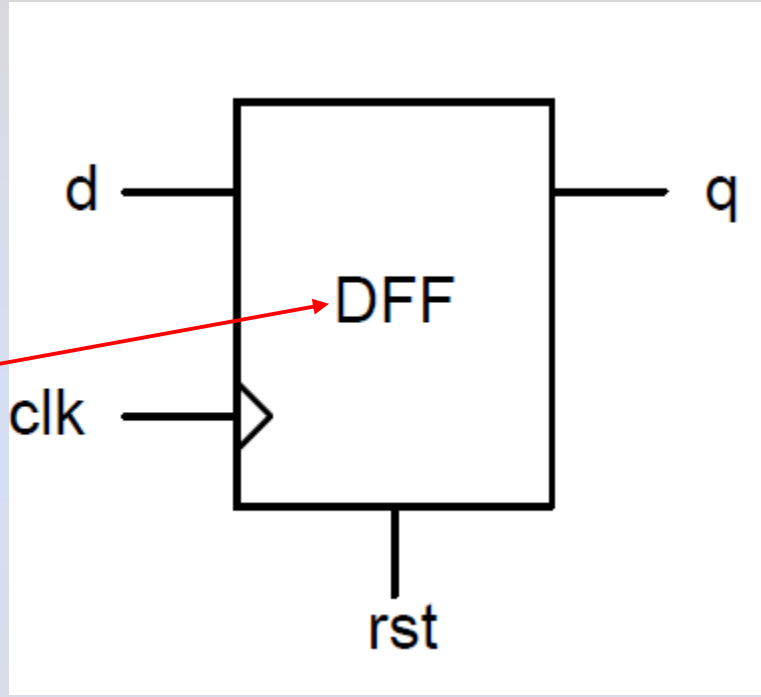
Proces se izvrši jedanput uvek kad se promeni vrednost nekog od signala *a*, *b* ili *c*.

Isti efekat kao jednostavna konkurentna naredba dodele:

```
y <= a OR b OR c;
```

Proces sa listom senzitivnosti - opis sekvencijalnog kola

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.all;
 3  ENTITY dff IS
 4  PORT (d,clk,rst : IN STD_LOGIC;
 5  q : OUT STD_LOGIC);
 6  END dff;
 7  ARCHITECTURE arch_proc OF dff IS
 8  BEGIN
 9  PROCESS(clk,rst)
10  BEGIN
11  IF(rst='1') THEN
12  q <= '0';
13  ELSIF(clk'EVENT AND clk='1') THEN
14  q <= d;
15  END IF;
16  END PROCESS;
17  END arch_proc;
```



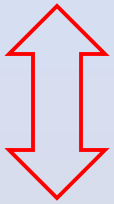
Proces sa *wait* naredbom

- **Ako se WAIT koristi u procesu, lista senzitivnosti mora biti izostavljena**
- WAIT sustenduje izvršenje procesa do ispunjenja zadatog uslova
- Tri oblika:

1. **WAIT UNTIL uslov;**
2. **WAIT ON signal1 [, signal2, …];**
3. **WAIT FOR vreme;** -- samo u simulaciji

Proces sa *wait* naredbom - opis kombinacionog kola

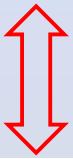
```
SIGNAL a, b, c : STD_LOGIC;
. . .
PROCESS(a, b, c)
BEGIN
y <= a OR b OR c;
END PROCESS;
```



```
SIGNAL a, b, c : STD_LOGIC;
. . .
PROCESS
BEGIN
WAIT ON a, b, c;
y <= a OR b OR c;
END PROCESS;
```

Proces sa *wait* naredbom - opis sekvencijalnog kola

```
 1  PROCESS (clk)
 2  VARIABLE v : INTEGER RANGE 0 TO 10;
 3  BEGIN
 4  IF(clk'EVENT AND clk='1') THEN
 5  v := v + 1;
 6  IF(v = 10) THEN
 7  v := 0;
 8  END IF;
 9  cifra <= v;
10  END IF;
11  END PROCESS;
```



```
 1  PROCESS
 2  VARIABLE v : INTEGER RANGE 0 TO 10;
 3  BEGIN
 4  WAIT UNTIL (clk'EVENT AND clk='1');
 5  v := v + 1;
 6  IF(v = 10) THEN
 7  v := 0;
 8  END IF;
 9  cifra <= v;
10  END PROCESS;
```

Signali u procesu

- Deklarišu se u deklarativnom delu arhitekture, a vidljivi su, tj. mogu se koristiti, kako u konkurentnim sekcijama arhitekture, tako i unutar procesa koji su obuhvaćeni arhitekturom.

```
ARCHITECTURE …
SIGNAL a,b : STD_LOGIC;
BEGIN
. . .
PROCESS(…)
BEGIN
b <= …;
END PROCESS;
. . .
a <= b …;
END …
```

Odložena dodela signala u procesu

- Dodela vrednosti signalu, obavljena unutar procesa, odlaže se do trenutka završetka tekućeg izvršenja procesa.
- U toku jednog izvršenja procesa, signal zadržava vrednost koju je imao u trenutku aktiviranja procesa (a koja potiče iz prethodnog izvršenja procesa), a novu vrednost dobija tek nakon deaktiviranja procesa.
- Ako se unutar procesa istom signalu više puta dodeli vrednost, samo će poslednja dodela imati vidljiv efekat.

Odložena dodela signala u procesu

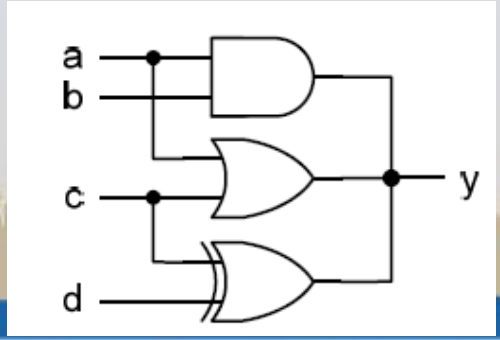
```
SIGNAL a, b, c, d : STD_LOGIC;
. . .
PROCESS(a,b,c,d)
BEGIN
y <= a AND b;
y <= a OR c;
y <= c XOR d;
END PROCESS;
```

```
PROCESS(a,b,c,d)
BEGIN
y <= c XOR d
END PROCESS;
```

Iste naredbe u konkurentnom kodu:

```
SIGNAL a,b,c,d : STD_LOGIC;
. . .
-- sledece naredbe nisu u procesu
y <= a AND b;
y <= a OR c;
y <= c XOR d;
```

Višestruke dodele u konkurentnom kodu nisu zabranjene, ali stvaraju kratak spoj.



Varijable

- Koriste se samo u sekvencijalnom kodu (PROCESS, PROCEDURE, FUNCTION).
- Samo kao lokalne promenljive.
- Ažuriranje varijable je trenutno (kao promenljive iz programskih jezika).
- Naredba dodele za varijable:

var := izraz;

Proces s varijablom

```
SIGNAL a, b, y : STD_LOGIC;
...
PROCESS(a,b)
VARIABLE v : STD_LOGIC;
BEGIN
v := '0';
v := v XOR a;
v := v XOR b;
y <= v;
END PROCESS;
```

Varijable se deklarišu u deklarativnoj sekciji procesa

Isti tipovi podataka kao za signale

Neposredno ažuriranje (odmah nakon izvršene naredbe)

Isti efekat kao:
`y <= a XOR b;`

Rezultat izračunavanja se iznosi van procesa

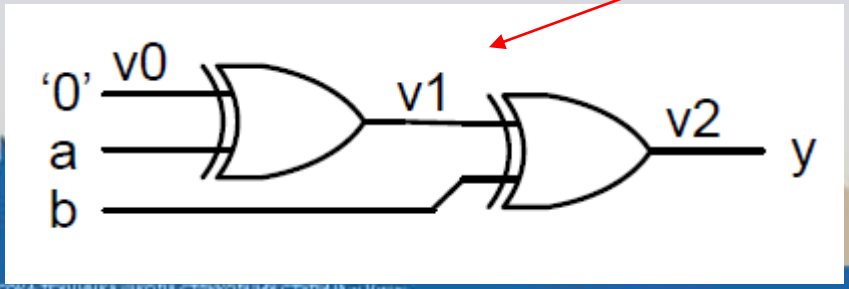
Sinteza procesa s varijablom

```
SIGNAL a, b, y : STD_LOGIC;
...
PROCESS(a,b)
VARIABLE v : STD_LOGIC;
BEGIN
v := '0';
v := v XOR a;
v := v XOR b;
y <= v;
END PROCESS;
```

```
PROCESS(a,b)
VARIABLE v0, v1, v2 : STD_LOGIC;
BEGIN
v0 := '0';
v1 := v0 XOR a;
v2 := v1 XOR b;
y <= v2;
END PROCESS;
```

Direktna sinteza nije moguća.

Neophodna je transformacija koda => zamena varijable v sa više novih varijabli, tako da se svaka koristi samo jedanput s leve strane u naredbi dodele. (Automatski sprovodi softver za sintezu)

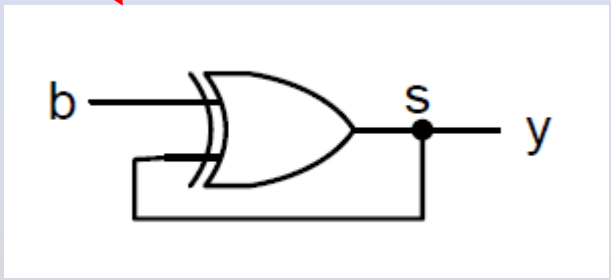


Signali vs varijable
(kod sa prethodnog slajda, ali sa signalima)

```
SIGNAL a, b, s : STD_LOGIC;
...
PROCESS(a,b,s)
BEGIN
s <= '0';
s <= s XOR a;
s <= s XOR b;
y <= s;
END PROCESS;
```

```
PROCESS(a,b,s)
BEGIN
s <= s XOR b;
y <= s;
END PROCESS;
```

Zbog odložene dodele



Signal se ne može delarisati u procesu.

Lista senzitivnosti za kombinaciono kolo sadrži sve signale koji se kao ulazni koriste u procesu (uključujući i s).

IF

- Sintaksa:

IF uslov_1 THEN
sekvencijane naredbe;
ELSIF uslov_2 THEN
sekvencijane naredbe;
ELSIF uslov_3 THEN
sekvencijane naredbe;
. . .
ELSE
sekvencijane naredbe;
END IF;

Uslovi se ispituju redom

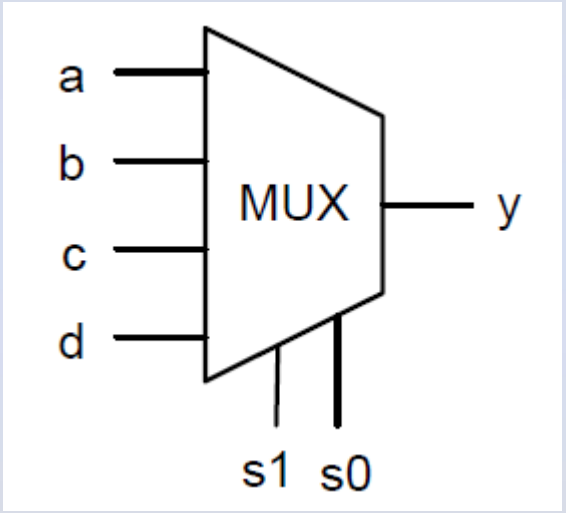
```
IF(x<y) THEN
v := "11111111";
w := '1';
ELSIF(x=y) THEN
v:="11110000";
w := '0';
ELSE
v:=(OTHERS => '0');
w := '1';
END IF;
```

Izvršava se samo ako ni jedan uslov nije tačan

IF - Multiplekser 4-u-1

```
 1  ENTITY mux IS
 2  PORT ( a,b,c,d : IN STD_LOGIC;
 3  s: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
 4  y : OUT STD_LOGIC);
 5  END mux;
 6  ARCHITECTURE if_arch OF mux IS
 7  BEGIN
 8  PROCESS(a,b,c,d,s)
 9  BEGIN
10  IF(s="00") THEN
11  y <= a;
12  ELSIF(s="01") THEN
13  y <= b;
14  ELSIF(s="10") THEN
15  y <= c;
16  ELSE
17  y <= d;
18  END IF;
19  END PROCESS;
20  END if_arch;
```

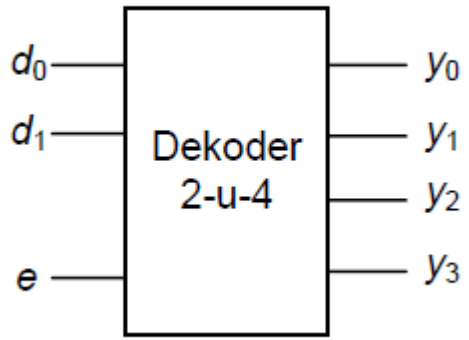
Multiplekser je kombinaciono kolo => u listi senzitivnosti moraju biti navedni svi ulazni signali.



IF - Dekoder 2-u-4

```
2 ARCHITECTURE if_arch OF dek2u4 IS
3 BEGIN
4 PROCESS(d, e)
5 BEGIN
6 IF(e = '0') THEN
7 y <= "0000";
8 ELSIF(d = "00") THEN
9 y <= "0001";
10 ELSIF(d = "01") THEN
11 y <= "0010";
12 ELSIF(d = "10") THEN
13 y <= "0100";
14 ELSE
15 y <= "1000";
16 END IF;
17 END PROCESS:
18 END if_arch;
```

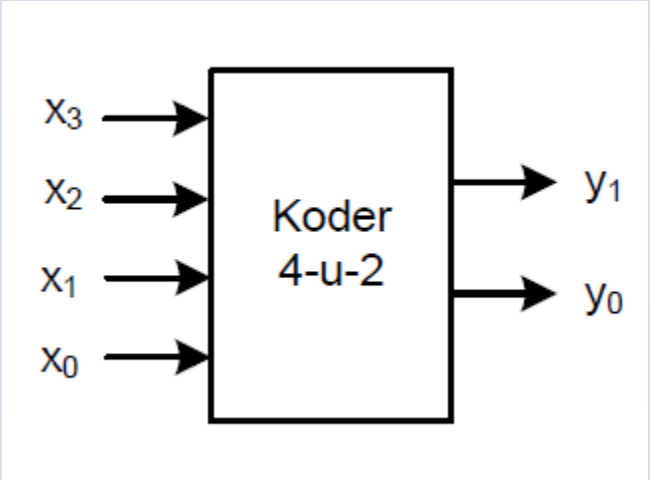
Dekoder je kombinaciono kolo => u listi senzitivnosti moraju biti navedni svi ulazni signali.



IF - Koder 4-u-2

```
2 ARCHITECTURE if_arch OF encoder IS
3 BEGIN
4 PROCESS(x)
5 BEGIN
6 IF(x = "0001") THEN
7 y <= "00";
8 ELSIF(x = "0010") THEN
9 y <= "01";
10 ELSIF(x = "0100") THEN
11 y <= "01";
12 ELSE
13 y <= "11";
14 END IF;
15 END PROCESS:
16 END if_arch;
```

Koder je kombinaciono kolo => u listi senzitivnosti moraju biti navedni svi ulazni signali.



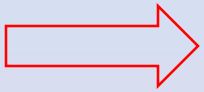
IF vs WHEN

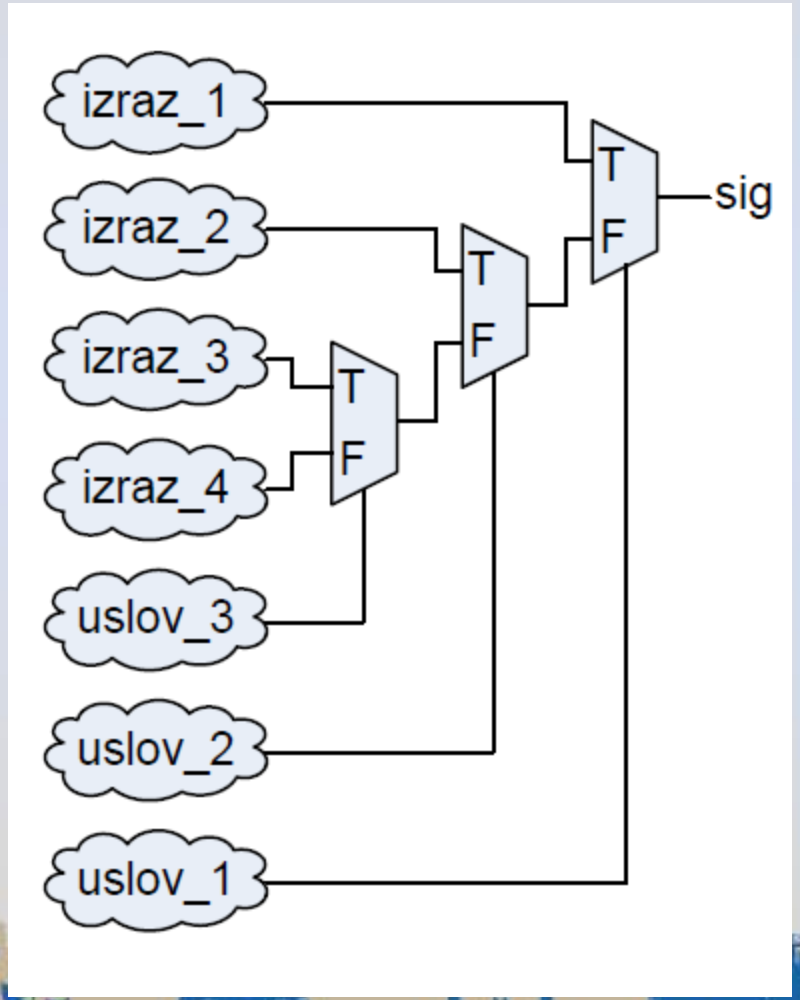
- Naredba IF donekle je slična konkurentnoj naredbi WHEN.
- Međutim, IF naredba je opštija od WHEN:
- **Grane IF naredbe mogu sadržati više od jedne naredbe.**
- **Dozvoljeno ugnježđavanje IF naredbi.**

IF - Konceptualna implementacija

- Ako postavlja samo jedan izlazni signal, isto kao *when.*

```
IF uslov_1 THEN
sig <= izraz_1;
ELSIF uslov_2 THEN
sig <= izraz_2;
ELSIF uslov_3 THEN
sig <= izraz_3;
ELSE
sig <= izraz_4;
END IF;
```

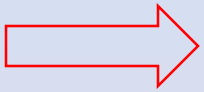


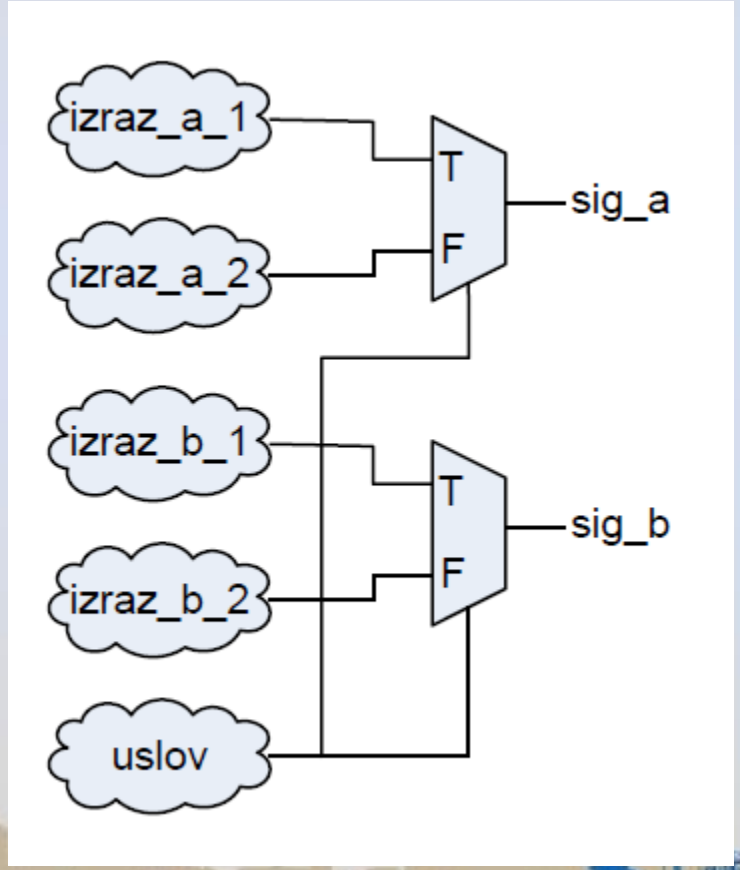


IF - Konceptualna implementacija

- Ako postavlja više od jednog izlaznog signala, za svaki signal zasebna multiplekserska mreža.

```
IF uslov_1 THEN
sig_a <= izraz_a_1;
sig_b <= izraz_b_1;
ELSE
sig_a <= izraz_a_2;
sig_b <= izraz_b_2;
END IF;
```

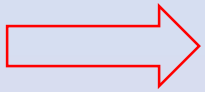


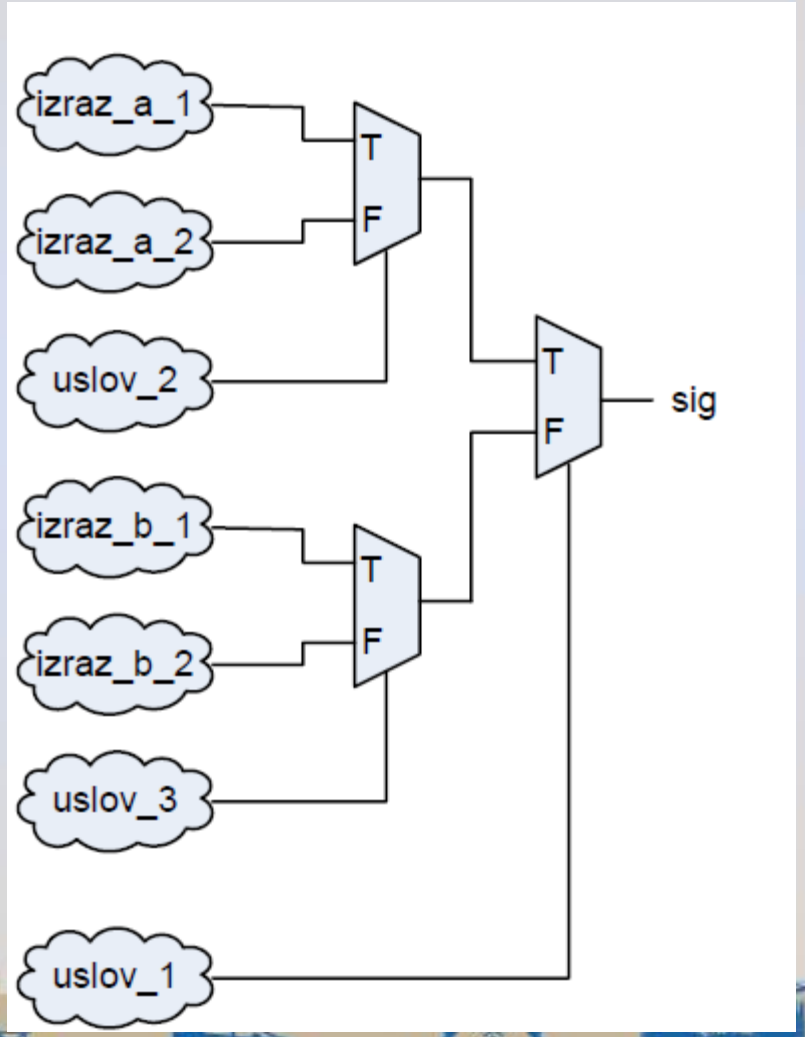


IF - Konceptualna implementacija

- Ugnježdene *if* naredbe.

```
IF uslov_1 THEN
IF uslov_2 THEN
sig <= izraz_a_1;
ELSE
sig <= izraz_a_2;
END IF;
ELSE IF uslov_3 THEN
sig <= izraz_b_1;
ELSE
sig <= izraz_b_2;
END IF;
END IF;
```





CASE

Sintaksa:

```
CASE selekcioni_izraz IS
WHEN vrednost => sekvencijalne_naredbe;
WHEN vrednost => sekvencijalne_naredbe;
. . .
END CASE;
```

Primer:

```
CASE ctrl IS
WHEN "00" => x<=a; y<=b;
WHEN "01" => x<=b; y<=c;
WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

Dozvoljeno više od jedne narebe.

Pokriva sve vrednosti koje nedostaju.

CASE - Primer (Multiplekser 4-u-1)

```
 1  ENTITY mux IS
 2  PORT ( a,b,c,d : IN STD_LOGIC;
 3  s: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
 4  y : OUT STD_LOGIC);
 5  END mux;
 6  ARCHITECTURE case_arch OF mux IS
 7  BEGIN
 8  PROCESS(a,b,c,d,s)
 9  BEGIN
10  CASE sel IS
11  WHEN "00" =>
12  y <= a;
13  WHEN "01" =>
14  y <= b;
15  WHEN "10" =>
16  y <= c;
17  WHEN OTHERS
18  y <= d;
19  END CASE;
20  END PROCESS;
21  END case_arch;
```

Neophodno, jer je s je tipa STD_LOGIC.

CASE – Binarni dekoder 2-u-4

```
2 ARCHITECTURE case_arch OF dek2u4 IS
3 SIGNAL ed : STD_LOGIC;
4 BEGIN
5 ed <= e & d;
6 PROCESS(ed)
7 BEGIN
8 CASE ed IS
9 WHEN "100" =>
10 y <= "0001";
11 WHEN "101" =>
12 y <= "0010";
13 WHEN "110" =>
14 y <= "0100";
15 WHEN "111" =>
16 y <= "1000";
17 WHEN OTHERS
18 y <= "0000";
19 END CASE;
20 END PROCESS;
21 END case_arch;
```

CASE – Binarni koder 4-u-2

```
2 ARCHITECTURE case_arch OF encoder IS
3 BEGIN
4 PROCESS(x)
5 BEGIN
6 CASE x IS
7 WHEN "0001" =>
8 y <= "00";
9 WHEN "0010" =>
10 y <= "01";
11 WHEN "0100" =>
12 y <= "10";
13 WHEN OTHERS
14 y <= "11";
15 END CASE;
16 END PROCESS;
17 END case_arch;
```

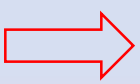
CASE vs SELECT

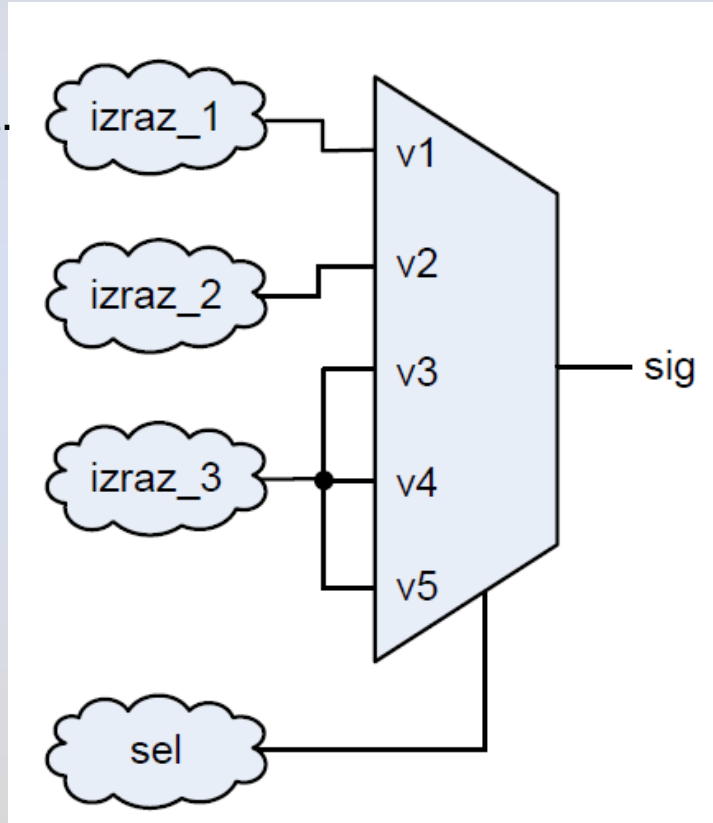
- Naredba *case* je slična konkurentnoj naredbi *select*.
- **CASE je opštija nareba jer dozvoljava više sekvencijalnih naredbi u jednoj grani.**

Konceptualna implementacija naredbe CASE - jedna naredba po grani

Isto kao konkurentna naredba select.

```
CASE sel IS
WHEN v1 => sig <= izraz_1;
WHEN v2 => sig <= izraz_2;
WHEN OTHERS sig <= izraz_3;
END CASE;
```

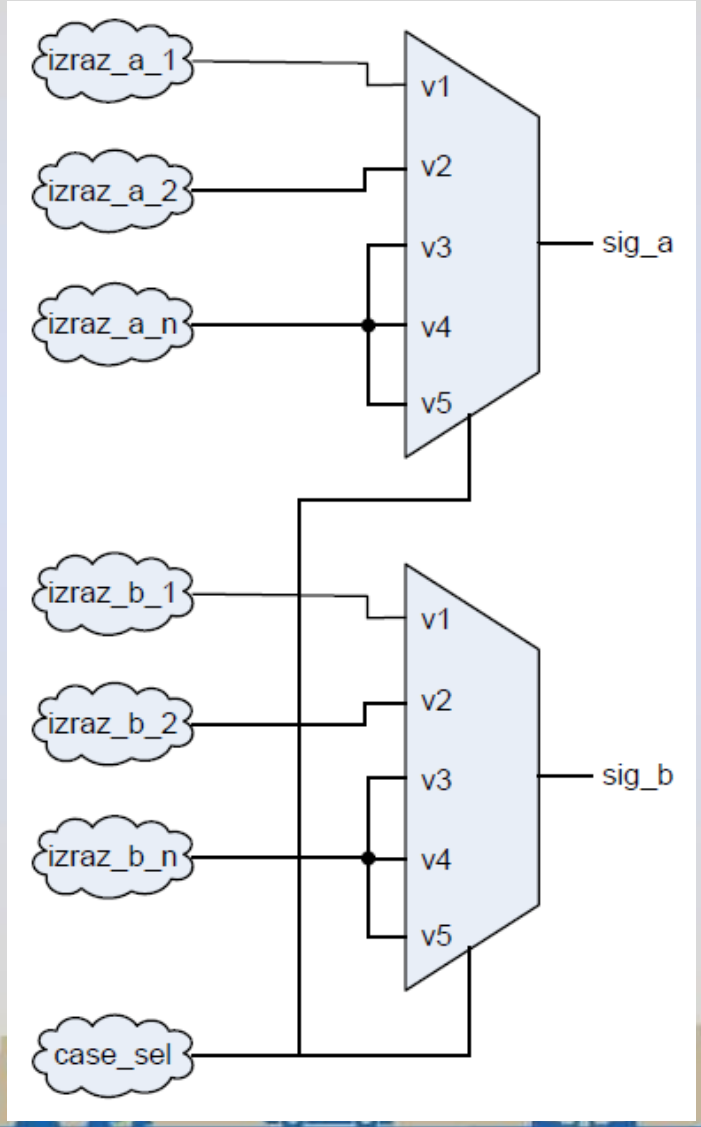




Konceptualna implementacija naredbe CASE -više od jedne naredba po grani

```
CASE case_uslov IS
WHEN v1 =>
sig_a <= izraz_a_0;
sig_b <= izraz_b_0;
WHEN v2 =>
sig_a <= izraz_a_1;
sig_b <= izraz_b_1;
WHEN OTHERS
sig_a <= izraz_a_n;
sig_b <= izraz_b_n;
END CASE;
```

Za svaki signal poseban multiplekser.



Sekvencijalni kod za kombinaciona kola

- Sekvencijalni kod se koristi za opis:

1. Sekvencijalnih kola (koriste se memorijski elementi)
2. Kombinacionih kola (memorijski elementi se ne koriste)

- Kako kompajler ″zna″ da se radi o kombinacinom kolu ?

1. Svi ulazni signali koji se koriste u procesu moraju biti navedeni u listi senzitivnosti.

2. U kodu su sadržane sve kombinacije ulaznih/izlaznih signala, tako da kompajler analizom koda može da rekonstruiše potpunu tabelu istinitosti.

Tipične greške u pisanju procesa za kombinaciona kola

ILI kolo:

```
PROCESS(a)
BEGIN
y <= a OR b OR c;
END PROCESS;
```

Nekomplenta lista senzitivnosti:

U simulaciji: Proces ne reaguje na promene signala *b* i *c*. Izlaz *y* zadržava svoju staru vrednost bez obzira na novu vrednost signala *b* ili *c* - slično flip-flopu.

U sintezi: Samo upozorenje, ali se ipak sintetiše ILI kolo.

Tipične greške u pisanju procesa za kombinaciona kola

Komparator jednakosti:

```
PROCESS(a,b)
BEGIN
IF(a=b) THEN
eq <= '1';
END IF;
END PROCESS;
```

A šta ako je a ≠ b?

Nedostaje ELSE grana.

Ispravno:

```
IF(a=b) THEN
eq <= '1';
ELSE
eq <= '0';
END IF;
```

Bez obzira na vrednosti signala a i b, signal eq uvek dobija definisanu vrednost, 0 ili 1.

Nedefinisan odziv stvara memoriju u kolu.
Kao da piše:

```
IF(a=b) THEN
eq <= '1';
ELSE
eq <= eq;
END IF;
```

Tipične greške u pisanju procesa za kombinaciona kola

Kompaktan, ali pogrešan opis potpunog komparatora:

```
PROCESS(a, b)
BEGIN
IF(a > b) THEN
gt <= '1';
ELSIF(a = b) THEN
eq <= '1';
ELSE
lt <= '1';
END IF;
END PROCESS;
```

IF naredba je kompletna, ali grane IF naredbe nisu.

Treba ovako:

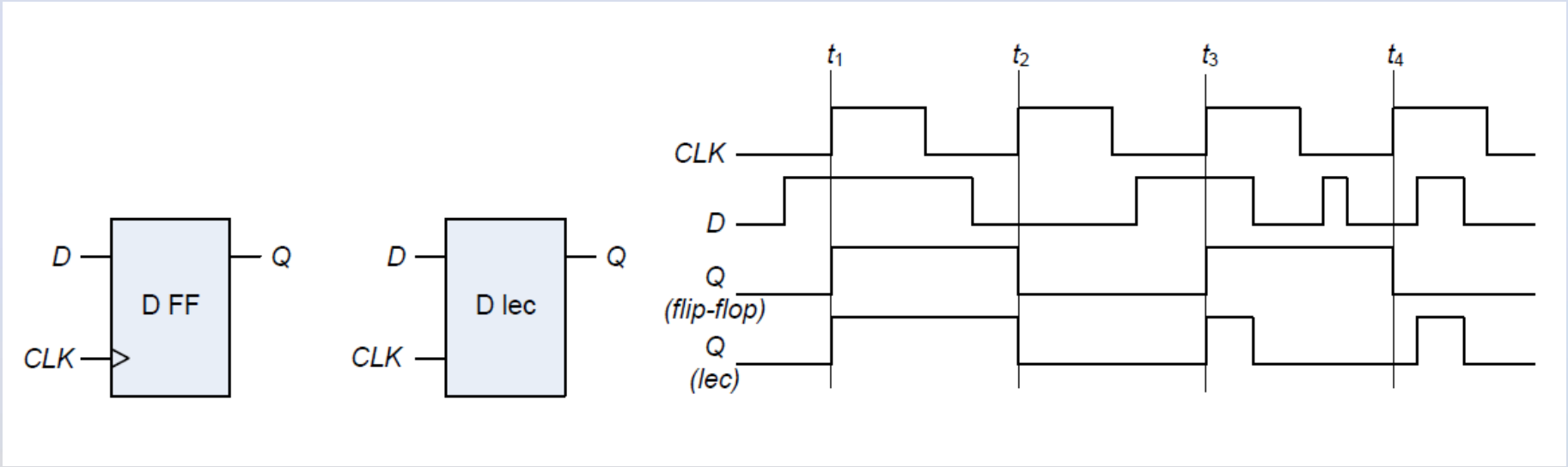
```
PROCESS(a, b)
BEGIN
IF(a > b) THEN
gt <= '1';
eq <= '0';
lt <= '0';
ELSIF (a = b) THEN
gt <= '0';
eq <= '1';
lt <= '0';
ELSE
gt <= '0';
eq <= '0';
lt <= '1';
END IF;
END PROCESS;
```

ili ovako:

```
PROCESS(a, b)
BEGIN
gt <= '0';
eq <= '0';
lt <= '0';
IF(a > b) THEN
gt <= '1';
ELSIF (a = b)
THEN
eq <= '1';
ELSE
lt <= '1';
END IF;
END PROCESS;
```

Leč kola i flip-flopovi

- **Leč je transparentan dok je CLK=1.**
- **Flip-flop nikada nije transparentan. Rastuća (ili opadajuća) ivica CLK inicira upis u flip-flop.**



D flip - flop

```
 2  LIBRARY ieee;
 3  USE ieee.std_logic_1164.all;
 4
 5  ENTITY dff IS
 6  PORT (d,clk : IN STD_LOGIC; q : OUT STD_LOGIC);
 7  END dff;
 8
 9  ARCHITECTURE flip_flop OF dff IS BEGIN
10  PROCESS(clk)
11  BEGIN
12  IF(clk'EVENT AND clk='1') THEN
13  q <= d;
14  END IF;
15  END PROCESS;
16  END flip_flop;
```

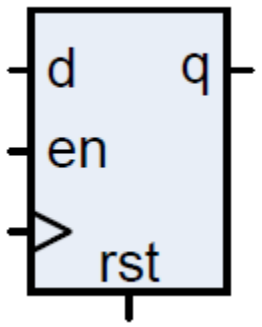
D Leč

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.all;
 3  ENTITY dlec IS
 4  PORT (d,clk : IN STD_LOGIC; q : OUT STD_LOGIC);
 5  END dlec;
 6  ARCHITECTURE lec OF dlec IS BEGIN
 7  PROCESS(d,clk)
 8  BEGIN
 9  IF (clk = '1') THEN
10  q <= d;
11  END IF;
12  END PROCESS;
13  END lec;
```

D FF sa dozvolom

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.all;
 3  ------------------------------------
 4  ENTITY dffe IS
 5  PORT (d,clk,en,rst : IN STD_LOGIC;
 6  q : OUT STD_LOGIC);
 7  END dff;
 8  ------------------------------------
 9  ARCHITECTURE dffe_v1 OF dffe IS
10  BEGIN
11  PROCESS(clk,rst)
12  BEGIN
13  IF(rst = '1') THEN
14  q <= '0';
15  ELSIF(clk'EVENT AND clk='1') THEN
16  IF(en = '1') THEN
17  q <= d;
18  END IF;
19  END IF;
20  END PROCESS;
21  END dffe_v1;
```

rst	clk	en	q*
1	x	x	0
0	0	x	q
0	1	x	q
0	↓	0	q
0	↓	1	d



Sinteza signala u flip-flop

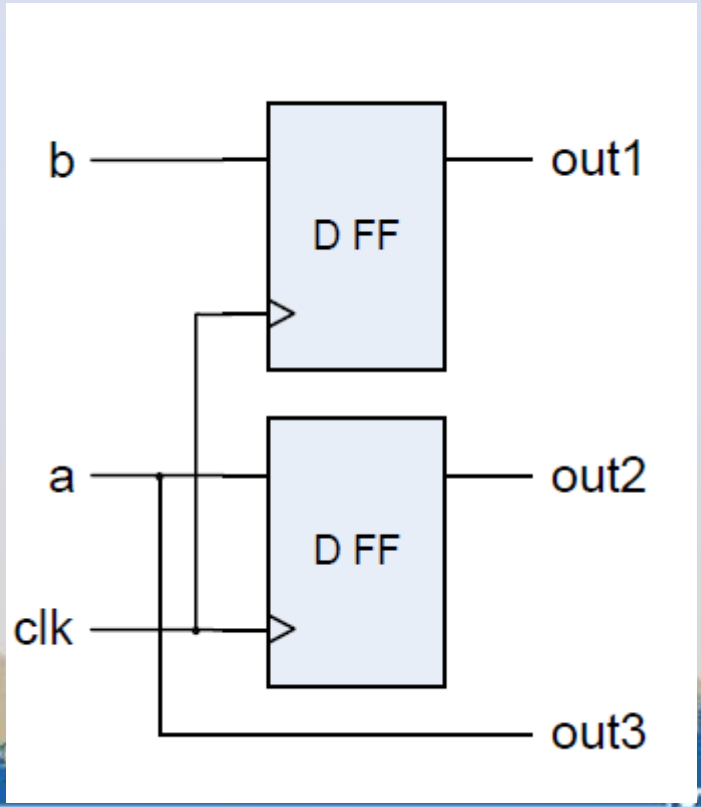
- Pod kojim uslovima se signali sintetišu u registre (flip-flopove) ?
- Signal se sintetiše u flip-flop ako je dodela vrednosti tom signalu uslovljena tranzicijom (promenom vrednosti) nekog drugog signala.
- Takve naredbe dodele se nazivaju sinhronim naredbama dodele i mogu se javiti samo u sekvencijalnom kodu, obično nakon konstrukcija tipa: ″IF signal´EVENT …″ ili ″WAIT UNTIL …″.

Sinteza signala u flip-flop

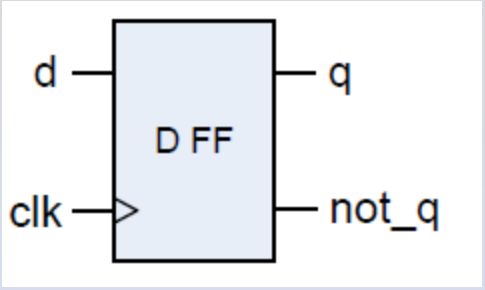
```
IF(clk'EVENT AND clk='1') THEN
q <= d;
END IF;
```

Sinhrona naredba dodele - uslovljena ivicom signala => q se sintetiše u flip-flop

```
PROCESS(clk,a)
BEGIN
IF(clk'EVENT AND clk='1')THEN
out1 <= b;
out2 <= a;
END IF;
out3 <= a;
END PROCESS;
```

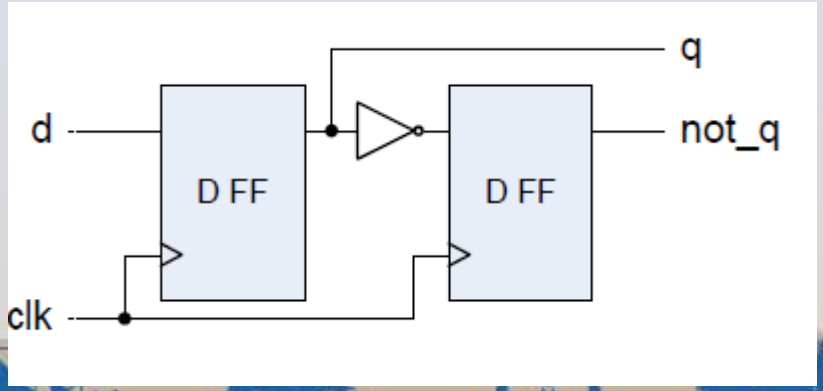


D flip-flop sa komplementarnim izlazima




```
ARCHITECTURE neispravna OF dff
IS
SIGNAL q_reg : STD_LOGIC;
BEGIN
PROCESS(clk)
BEGIN
IF(clk'EVENT AND clk='1') THEN
q_reg <= d;
not_q <= NOT q_reg;
END IF;
END PROCESS;
END neispravna;
```

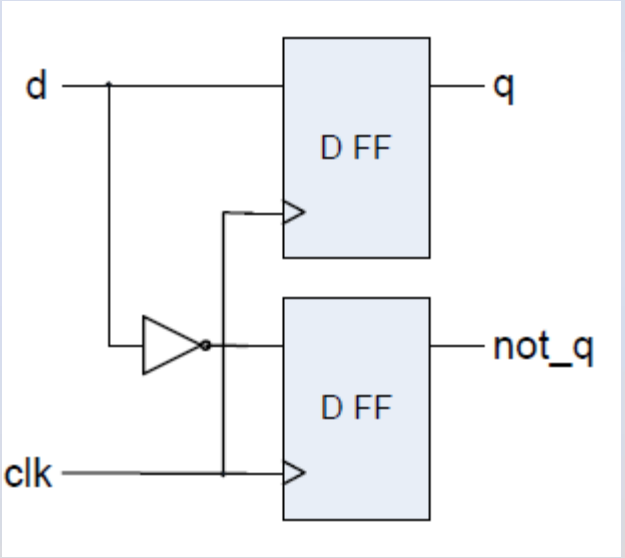
Neispravno !!!
Za signale u procesu važi odložena dodela.



D flip-flop sa komplementarnim izlazima

Ispravno, ali dva flip-flopa.

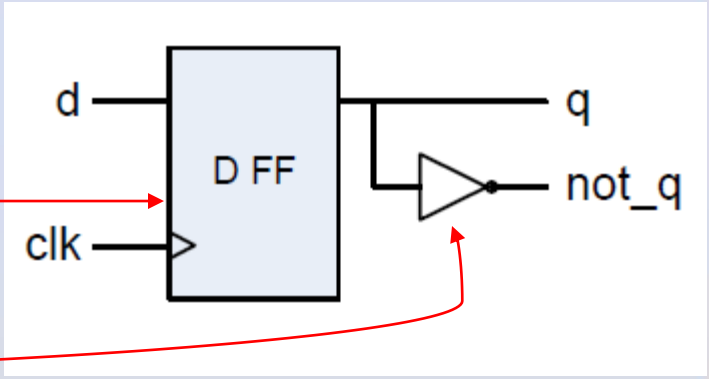
```
ARCHITECTURE dva_flip_flopa OF dff IS
BEGIN
PROCESS(clk)
BEGIN
IF(clk'EVENT AND clk='1') THEN
q <= d;
not_q <= NOT d;
END IF;
END PROCESS;
END dva_flip_flopa;
```



D flip-flop sa komplementarnim izlazima

```
ARCHITECTURE ispravna OF dff IS
BEGIN
PROCESS(clk)
BEGIN
IF(clk'EVENT AND clk='1') THEN
q <= d;
END IF;
END PROCESS;
not_q <= NOT q;
END ispravna;
```

Ispravno, jedan flip-flop i jedan invertor.



Sinteza varijable u flip-flop

- Pod kojim uslovima se varijable sintetišu u registre (flip-flopove) ?

1. Dodela vrednosti varijabli je uslovljena tranzicijom nekog drugog signala.

ili

Varijabla se koristi u kodu pre nego što joj je dodeljena vrednost.

i

2. Vrednost varijable se prenosi izvan procesa (putem dodele nekom signalu).

Sinteza varijable u flip-flop

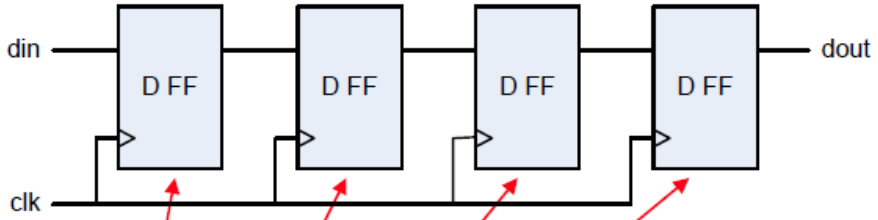
```
ARCHITECTURE . . .
SIGNAL x : BIT;
BEGIN
. . .
PROCESS(clk)
VARIABLE v : BIT;
BEGIN
IF(clk'EVENT AND clk='1')THEN
v := a;
END IF;
x <= v;
END PROCESS;
. . .
END ARCHITECTURE;
```

Sinhrona dodela.

Vrednost varijable se prenosi izvan procesa.

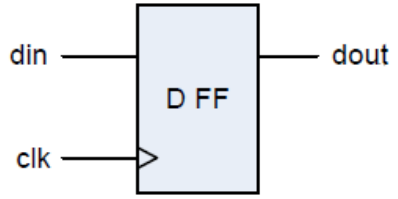
Varijabla *v* se sintetiše u flip-flop.

Sinteza varijable u flip-flop




```
ARCHITECTURE pomreg OF pomreg IS
BEGIN
  PROCESS (clk)
    VARIABLE a,b,c : BIT;
  BEGIN
    IF(clk'EVENT AND clk='1') THEN
      dout <= c;
      c := b;
      b := a;
      a := din;
    END IF;
  END PROCESS;
END pomreg;
```

Svaka varijabla se koristi pre nego što joj je dodeljena vrednost




```
ARCHITECTURE pomreg OF pomreg IS
BEGIN
  PROCESS (clk)
    VARIABLE a,b,c : BIT;
  BEGIN
    IF(clk'EVENT AND clk='1') THEN
      a := din;
      b := a;
      c := b;
      dout <= c;
    END IF;
  END PROCESS;
END pomreg;
```

Sažima se na:
c := din;

Signali vs Varijable

	SIGNAL	VARIABLE
Operator dodele	<=	:=
Upotreba	Povezivanje entiteta i komponenti	Predstavlja lokalne promenljive
Opseg važenja	Može biti globalan	Isključivo lokalna (vidljiva samo unutar procesa, procedure ili funkcije)
Ponašanje	Promena vrednosti nije trenutna (nova vrednost nije raspoloživa sve do završetka procesa, funkcije ili procedure)	Promena vrednosti je trenutna (nova vrednost se može koristiti sledećoj naredbi)
Koristi se u	Paketu, entitetu i arhitekturi. U entitetu, svi portovi su signali.	Isključivo u sekvencijalnom kôdu.

LOOP

- Za kreiranje petlji u sekvencijalnom kôdu
- Tri oblika:

1. **FOR** LOOP
2. WHILE LOOP (ne može se sintetizovati)
3. LOOP (ne može se sintetizovati)

FOR LOOP

- Petlja se ponavlja zadati broj puta
- Sintaksa

[labela:] FOR identifikator IN opseg LOOP

(sekvencijalne naredbe)

END LOOP [labela];

- Primer:

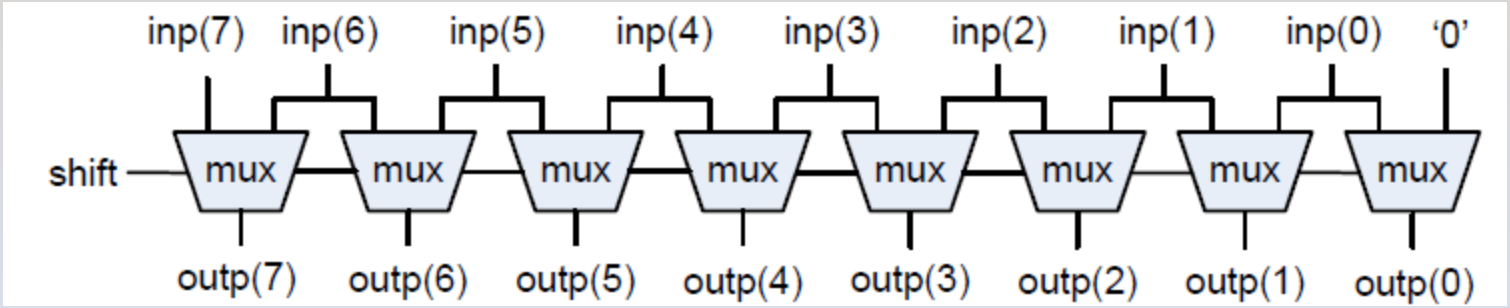
```
faktorijel := 1;
FOR broj IN 2 TO 10 LOOP
faktorijel := faktorijel * broj;
END LOOP;
```

- Ograničenja:

U telu petlje, nije dozvoljeno dodeljivati vrednosti identifikatoru.

U kodu namenjenom sintezi obe granice opsega FOR LOOP naredbe moraju biti statičke.

Jednostavan *barrel* pomerač

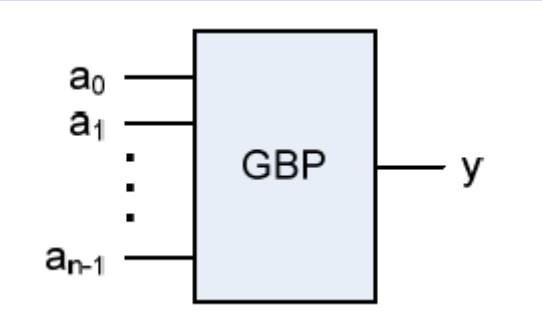



```
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 ----------------------------------------------
5 ENTITY barrel IS
6 PORT (inp : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7 shift : IN STD_LOGIC;
8 outp : OUT STD_LOGIC_VECTOR(n-1 DOWNTO 0));
9 END barrel;
10 ----------------------------------------------
11 ARCHITECTURE loop_arch OF barrel IS
12 BEGIN
13 PROCESS(inp,shift)
14 BEGIN
15 IF(shift='0') THEN
16 outp <= inp;
17 ELSE
18 outp(0)<='0';
19 FOR i IN 1 TO 7 LOOP
20 outp(i) <= inp(i-1);
21 END LOOP;
22 END IF;
23 END PROCESS;
24 END loop_arch;
```

Generator bita parnosti

```
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 ----------------------------------------
5 ENTITY pargen IS
6 PORT (a : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7 y : OUT STD_LOGIC);
8 END pargen;
9 ----------------------------------------
10 ARCHITECTURE loop_arch OF pargen IS
11 BEGIN
12 PROCESS(a)
13 VARIABLE p : STD_LOGIC;
14 BEGIN
15 p := a(0);
16 FOR i IN 1 TO 7 LOOP
17 p := p XOR a(i);
18 END LOOP;
19 y <= p;
20 END PROCESS;
21 END loop_arch;
```

Vrednost na izlazu dopunjuje broj ulaznih 1-ca do parnog broja.

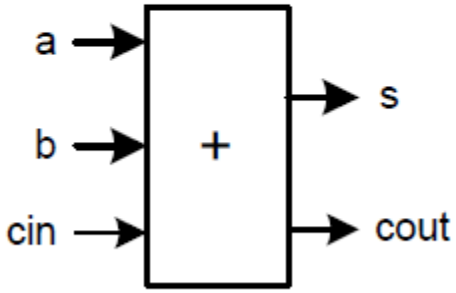


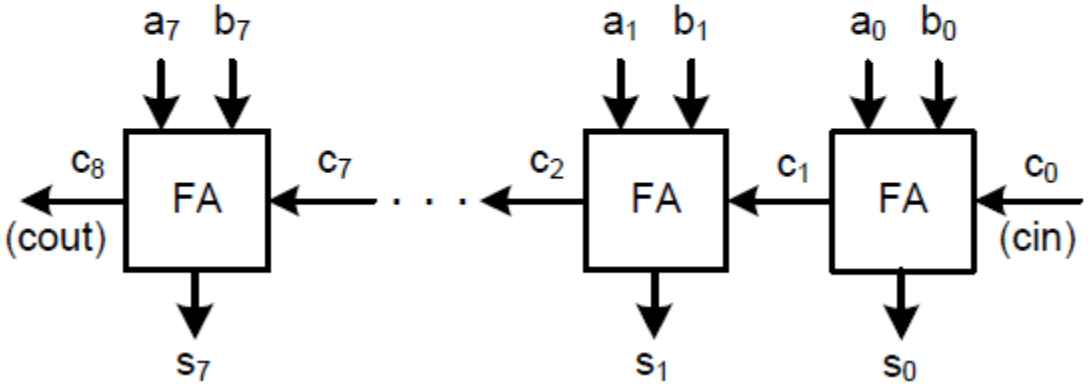

```
1  y <= a(0) XOR a(1) XOR … XOR a(7);
```

Sabirač sa rednim prenosom

$$s_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = a_i \cdot b_i + a_i \cdot c_i + b_i \cdot c_i$$





Sabirač sa rednim prenosom

```
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 ----------------------------------------------------------------
5 ENTITY sabirac IS
6 PORT (a,b : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
7 cin : IN STD_LOGIC;
8 s : OUT STD_LOGIC_VECTOR(7 DOWNTO 0);
9 cout: OUT STD_LOGIC);
10 END sabirac;
11 ----------------------------------------------------------------
12 ARCHITECTURE loop_arch OF sabirac IS
13 BEGIN
14 PROCESS(a,b,cin)
15 VARIABLE c : STD_LOGIC_VECTOR(8 DOWNTO 0);
16 BEGIN
17 c(0):=cin;
18 FOR i IN 0 TO 7 LOOP
19 s(i) <= a(i) XOR b(i) XOR c(i);
20 c(i+1):=(a(i) AND b(i))OR(a(i) AND c(i))OR(b(i) AND c(i));
21 END LOOP;
22 cout <= c(8);
23 END PROCESS;
24 END loop_arch;
```

Konceptualna implementacija FOR LOOP naredbe

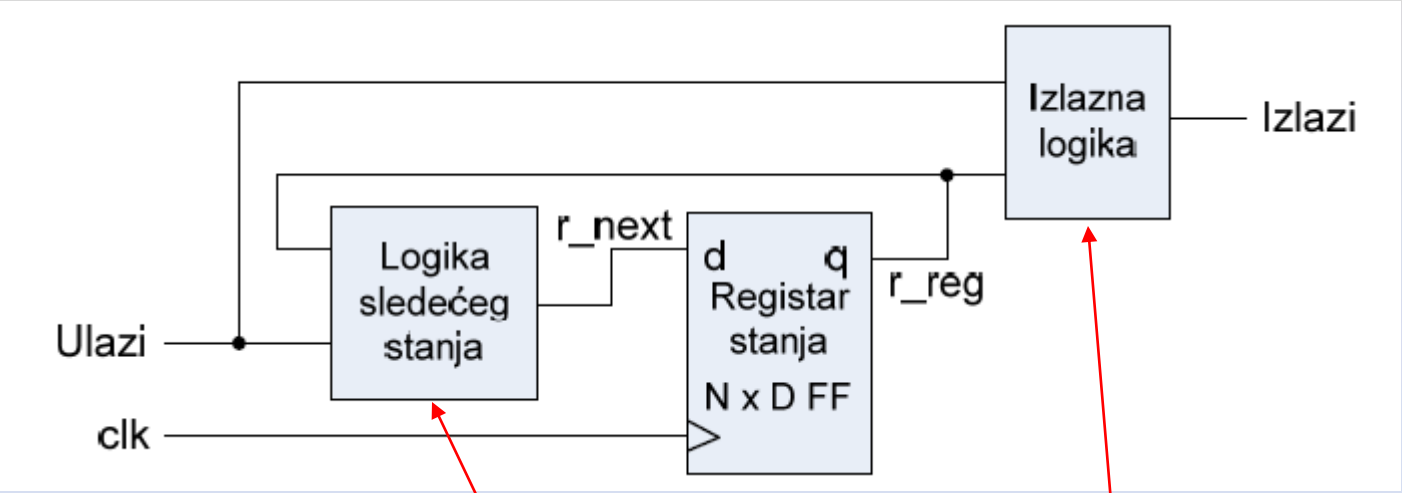
- Osnovni način za realizaciju *for loop* naredbe u hardveru zasniva se na *razmotavanju* petlje, odnosno na transformaciji polaznog koda u kod koji ne sadrži petlje.
- To znači da se hardver koji je opisan telom petlje ponavlja (replicira) za svaku iteraciju.
- Da bi razmotavanje bilo moguće, neophodno je da opseg indeksa petlje bude konstantan i poznat u vremenu kad se vrši sinteza (tj. mora biti statički).
- Na primer, kod sabirača iz prethodnog primera u razmotanom obliku postaje:

```
1  c(0):=cin;
2  -- i = 0 ------------------------------------------------
3  s(0) <= a(0) XOR b(0) XOR c(0);
4  c(1):=(a(0) AND b(0)) OR (a(0) AND c(0)) OR (b(0) AND c(0));
5  -- i = 1 ------------------------------------------------
6  s(1) <= a(1) XOR b(1) XOR c(1);
7  c(2):=(a(1) AND b(1)) OR (a(1) AND c(1)) OR (b(1) AND c(1));
8  -- i = 2, 3 ... 6 ---------------------------------------
9  . . .
10 -- i = 7 ------------------------------------------------
11 s(7) <= a(7) XOR b(7) XOR c(7);
12 c(8):=(a(7) AND b(7)) OR (a(7) AND c(7)) OR (b(7) AND c(7));
13 ------------------------------------------------------------
14 cout <= c(8);
15 ------------------------------------------------------------
```

Registarske komponente

- Sinhrona sekvencijalna digitalna kola standardne (prepoznatljive) funkcije, poput prihvatnih (stacionarnih), pomeračkih i brojačkih registara.
- U osnovi svake registarske komponente se nalazi kolekcija D flip-flopova sa zajedničkim signalom takta – tzv. *registar stanja*.
- Registar stanja memoriše tekuće stanje kola.
- Kombinacioni blok ″logika sledećeg stanja″ određuje naredno stanje kola na osnovu vrednosti ulaznih signala i tekućeg stanja.
- ″Logika izlaza″ je još jedan kombinacioni blok, koji na osnovu tekućeg stanja i vrednosti ulaznih signala generiše izlazne signale kola.

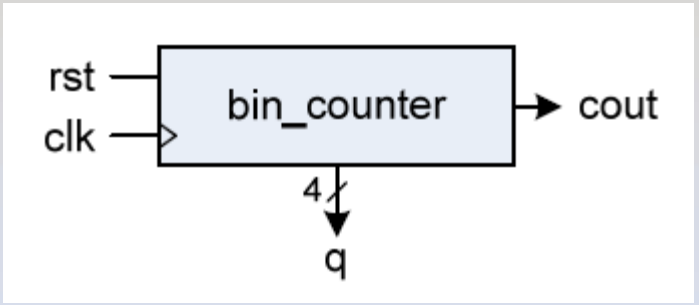
Registarske komponente

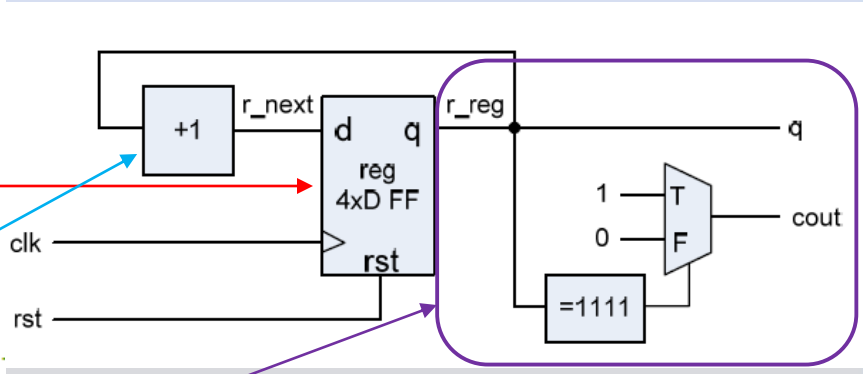


Funkcije blokova sledećeg stanja i izlaza mogu se izraziti jednostavnim regularnim logičkim ili aritmetičkim funkcijama.

Binarni brojač – logika sledećeg stanja i izlaza van procesa

```
1  ------------------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  USE IEEE.NUMERIC_STD.ALL;
5  ------------------------------------------------
6  ENTITY bin_counter IS
7  PORT (clk,rst : IN STD_LOGIC;
8  cout : OUT STD_LOGIC;
9  q : OUT STD_LOGIC_VECTOR(3 DOWNTO 0));
10 END bin_counter;
11 -- Ispravan trosegmentni opis ---------------
12 ARCHITECTURE arch_v1 OF bin_counter IS
13 SIGNAL r_reg, r_next : UNSIGNED(3 DOWNTO 0);
14 BEGIN
15 --- Registar stanja -------------------------
16 PROCESS(clk,rst)
17 BEGIN
18 IF(rst = '1') THEN
19 r_reg <= (OTHERS => '0');
20 ELSIF(clk'EVENT AND clk = '1') THEN
21 r_reg <= r_next;
22 END IF;
23 END PROCESS;
24 -- Logika sledeceg stanja ------------------
25 r_next <= r_reg + 1;
26 -- Izlazna logika --------------------------
27 q <= STD_LOGIC_VECTOR(r_reg);
28 cout <= '1' WHEN r_reg = "1111" ELSE '0';
29 END arch_v1;
```

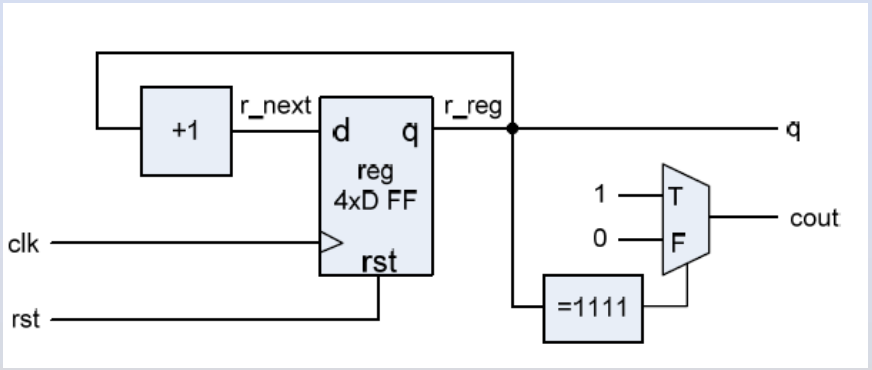




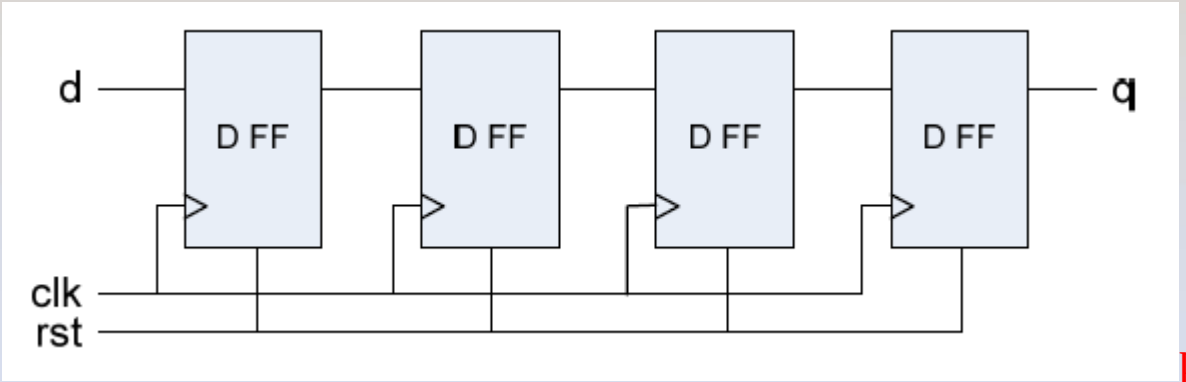
Binarni brojač - celokupan kod u procesu, koristi varijablu

```
50 -- Ispravan jednosegmentni opis -------
51 ARCHITECTURE arch_v3 OF bin_counter IS
52 BEGIN
53 PROCESS(clk,rst)
54 VARIABLE q_tmp : UNSIGNED(3 DOWNTO 0);
55 BEGIN
56 IF(rst = '1') THEN
57 q_tmp := (OTHERS => '0');
58 ELSIF(clk'EVENT AND clk = '1') THEN
59 q_tmp := q_tmp + 1;
60 END IF;
61 IF(q_tmp = "1111") THEN
62 cout <= '1';
63 ELSE
64 cout <= '0';
65 END IF;
66 q <= STD_LOGIC_VECTOR(q_tmp);
67 END PROCESS;
68 END arch_v3;
```

q_tmp ima vrednost koju je dobila u prethodnoj IF naredbi. (Za varijable ne važi odložena dodela)



Pomerački registar



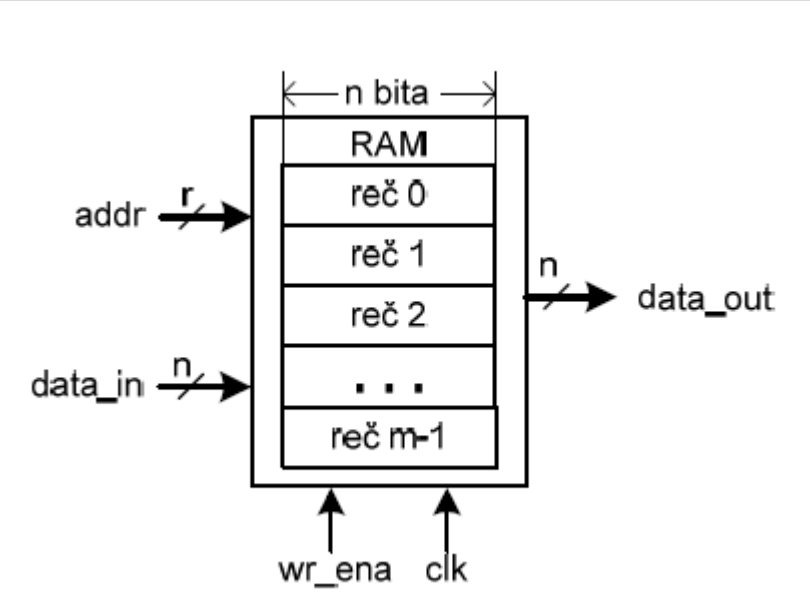
proces i signali

```
1  ----------------------------------------
2  LIBRARY IEEE;
3  USE IEEE.STD_LOGIC_1164.ALL;
4  ----------------------------------------
5  ENTITY pomreg IS
6  PORT ( d,clk,rst: IN STD_LOGIC;
7  q: OUT STD_LOGIC);
8  END pomreg;
9  -- Resenje 1 -----------------------------
10 ARCHITECTURE arch_v1 OF pomreg IS
11 SIGNAL reg : STD_LOGIC_VECTOR(3 DOWNTO 0);
12 BEGIN
13 PROCESS (clk, rst)
14 BEGIN
15 IF(rst='1') THEN
16 reg <= (OTHERS => '0');
17 ELSIF(clk'EVENT AND clk='1') THEN
18 reg <= d & reg(3 DOWNTO 1);
19 END IF;
20 END PROCESS;
21 q <= reg(0);
22 END arch_v1;
```

proces i varijabla

```
23 -- Resenje 2 -----------------------------
24 ARCHITECTURE arch_v2 OF pomreg IS
25 BEGIN
26 PROCESS (clk, rst)
27 VARIABLE reg : STD_LOGIC_VECTOR(3 DOWNTO 0);
28 BEGIN
29 IF(rst='1') THEN
30 reg := (OTHERS => '0');
31 ELSIF(clk'EVENT AND clk='1') THEN
32 reg := d & reg(3 DOWNTO 1);
33 END IF;
34 q <= reg(0);
35 END PROCESS;
36 END arch_v2;
```

RAM sa razdvojenim ulazim i izlazim portovima za podatke



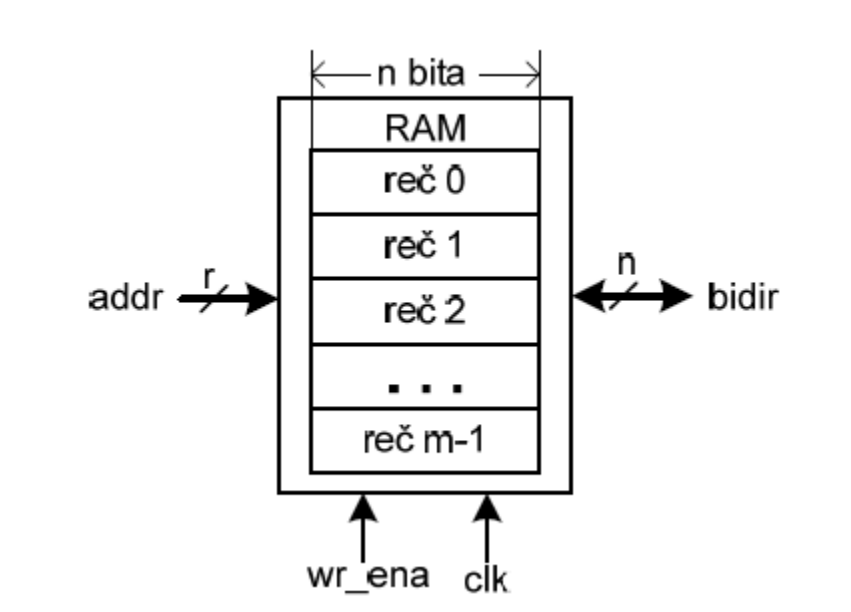
Vrednost prisutna na adresnom portu, *addr*, bira (adresira) memorijsku reč kojoj se pristupa. Pri *wr_ena*=´1´, vrednost prisutan na ulaznom portu za podatke, *data_in*, upisuje se u adresiranu reč pod dejstvom rastuće ivice takta, *clk*. Sadržaj adresirane reči je dostupan na izlaznom portu za podatke, *data_out*. Između broja reči u RAM-u, *m*, i broja bita adrese, *r*, postoji sledeća veza:

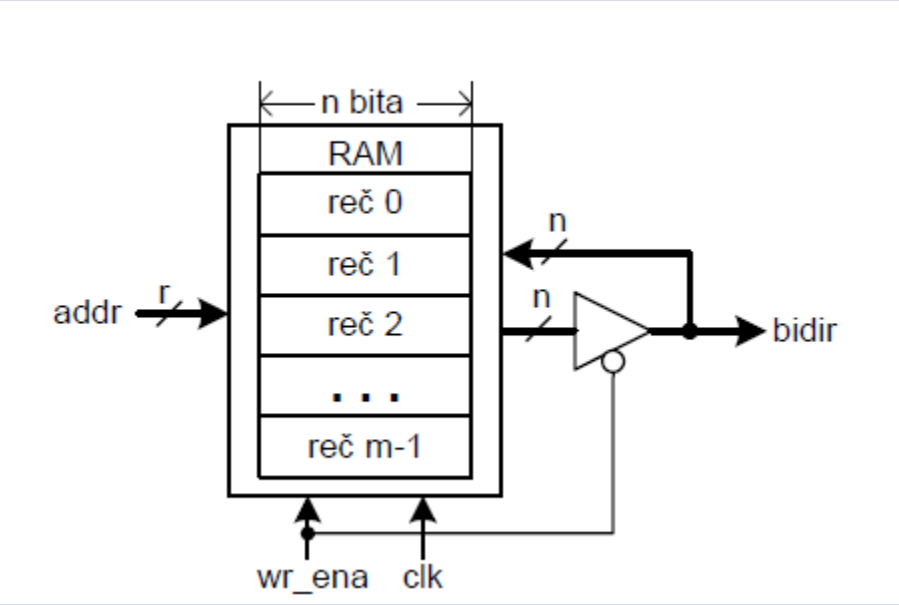
$$r = \lceil \log_2 m \rceil$$

RAM sa razdvojenim ulazim i izlazim portovima za podatke

```
1 ----------------------------------------------------------------------
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 USE IEEE.NUMERIC_STD.ALL;
5 ----------------------------------------------------------------------
6 ENTITY RAM IS
7 PORT (wr_ena, clk : IN STD_LOGIC;
8 addr : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9 data_in : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
10 data_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0));
11 END RAM;
12 ----------------------------------------------------------------------
13 ARCHITECTURE ram OF RAM IS
14 TYPE mem_array IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
15 SIGNAL memory : mem_array;
16 BEGIN
17 PROCESS(clk)
18 BEGIN
19 IF(clk'EVENT AND clk='1') THEN
20 IF(wr_ena = '1') THEN
21 memory(TO_INTEGER(UNSIGNED(addr))) <= data_in;
22 END IF;
23 END IF;
24 END PROCESS;
25 data <= memory(TO_INTEGER(UNSIGNED(addr)));
26 END ram;
27 ----------------------------------------------------------------------
```

RAM sa bidirekcionim ulaznoizlaznim portom za podatke





Kad je *wr_ena*=´1´, a pod dejstvom rastuće ivice taktnog signala, podatak koji je spolja postavljen na port *bidir*, upisuje se u reč adresiranu vrednošću koja je prisutnom na adresnom portu, *addr*. Kad je *wr_ena*=´0´, podatak iz adresirane reči se prenosi na port *bidir*. Na taj način, signal *wr_ena* reguliše smer bidirekcionog porta *bidir*: za *wr_ena*=´1´, *bidir* se ponaša kao ulazni, a za *wr_ena*=´0´ kao izlazni port.

RAM sa bidirekcionim ulaznoizlaznim portom za podatke

```
1 ----------------------------------------------------------------------
2 LIBRARY IEEE;
3 USE IEEE.STD_LOGIC_1164.ALL;
4 USE IEEE.NUMERIC_STD.ALL;
5 ----------------------------------------------------------------------
6 ENTITY RAM IS
7 PORT (wr_ena, clk : IN STD_LOGIC;
8 addr : IN STD_LOGIC_VECTOR(3 DOWNTO 0);
9 bidir : INOUT STD_LOGIC_VECTOR(7 DOWNTO 0));
10 END RAM;
11 ----------------------------------------------------------------------
12 ARCHITECTURE ram OF RAM IS
13 TYPE mem_array IS ARRAY (0 TO 15) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
14 SIGNAL memory : mem_array;
15 BEGIN
16 PROCESS(clk)
17 BEGIN
18 IF(clk'EVENT AND clk='1') THEN
19 IF(wr_ena = '1') THEN
20 memory(TO_INTEGER(UNSIGNED(addr))) <= bidir;
21 END IF;
22 END IF;
23 END PROCESS;
24 bidir <= memory(TO_INTEGER(UNSIGNED(addr))) WHEN wr_ena = '0'
25 ELSE (OTHERS => 'Z');
26 END ram;
27 ----------------------------------------------------------------------
```

Oblasti za mini-projekat/završni rad:

1. Projektovanje konačnih automata (Finite-State Machine)
2. Hijerarhijsko projektovanje
3. Parametrizovano projektovanje
4. RTL (Register Transfer Level) projektovanje