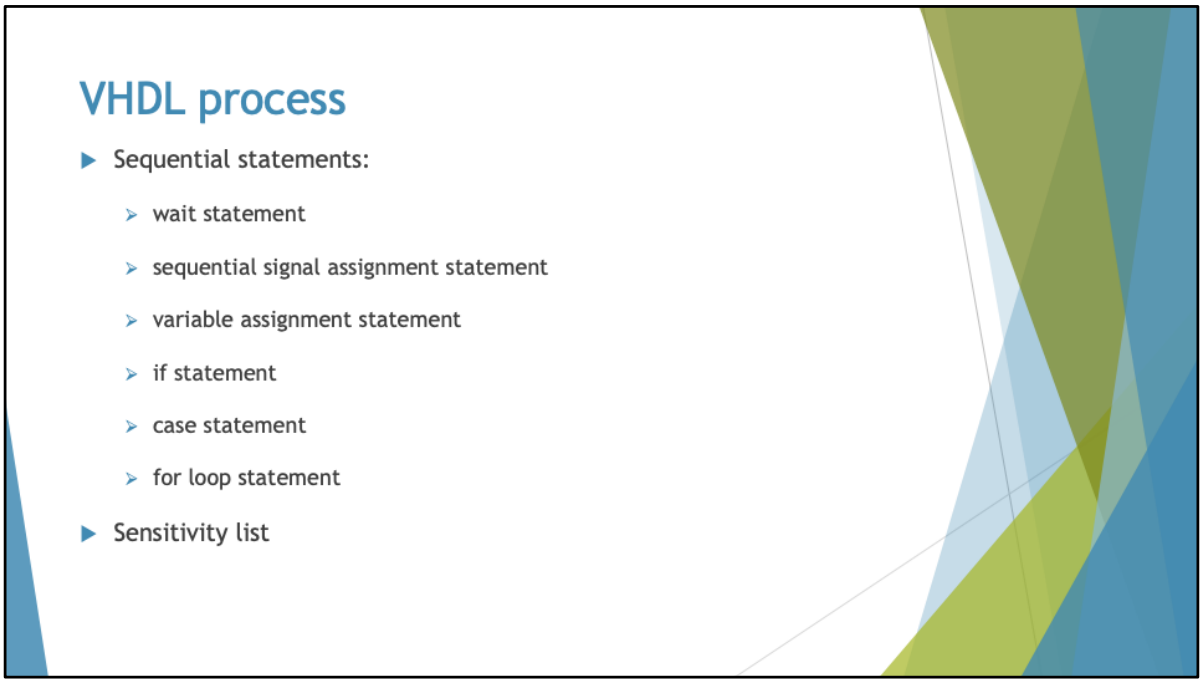
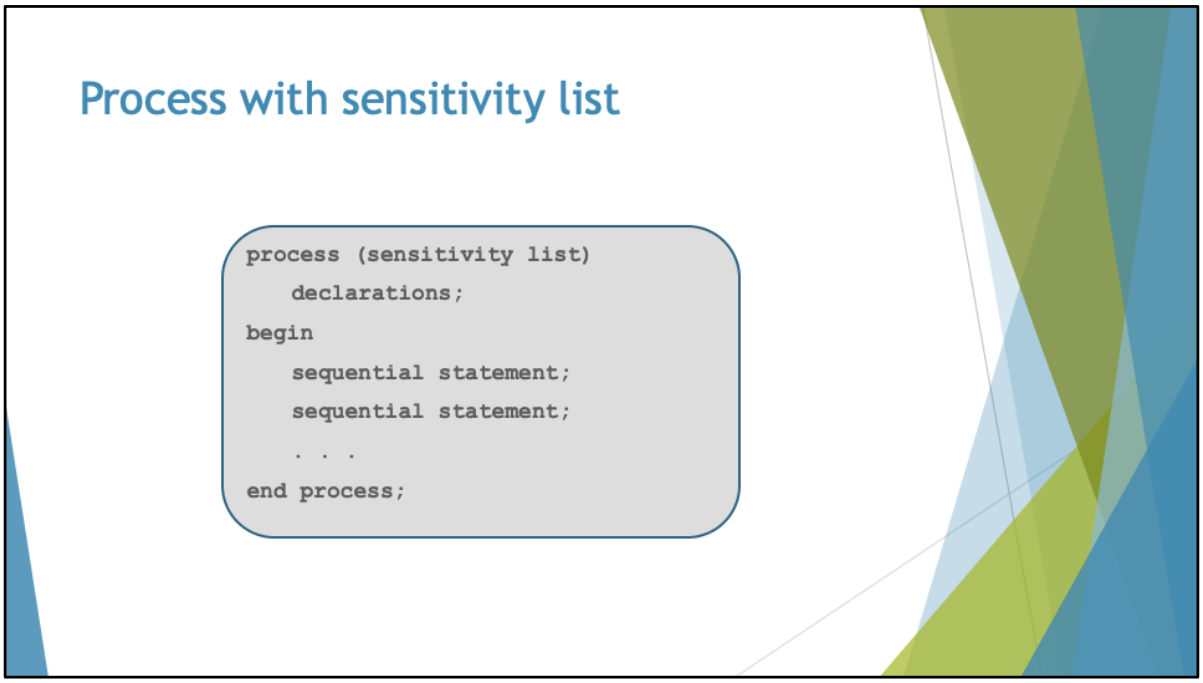
Semantika sekvencijalnih izraza je slična tradicionalnim programskim jezicima. Kako nisu kompatibilni sa generalnim concurrent VHDL modelom izvršavanja, sekvencijalni izrazi se ostvaruju kroz konstrukciju **process**. Osnovna namjena sekvencijalnih izraza jeste da opišu i modeluju apstraktno ponašanje kola. Ne postoji jasno mapiranje između sekvencijalnih izraza i hardverskih komponenti. Pojedini kodovi koji sadrže sekvencijalne izraze su veoma nepogodni za sintezu.



Process je konstrukcija u VHDL-u koja sadrži set akcija koje se izvršavaju sekvencijalno. Pomenute akcije se označavaju kao sekvencijalni izrazi (sequential statements). Sam process je concurrent statement. Može se interpretirati kao “black box” čije ponašanje je opisano kroz sekvencijalne izraze. Sekvencijalni izrazi uključuju veliki broj konstrukcija koje mogu egzistirati samo u okviru procesa. Izvršavanje u okviru procesa je sekvencijalno, tako da je redosljed navođenja izraza bitan. Process i sekvencijalni izrazi koji ga opisuju može se koristiti prilikom opisa kombinacionih i sekvencijalnih kola.

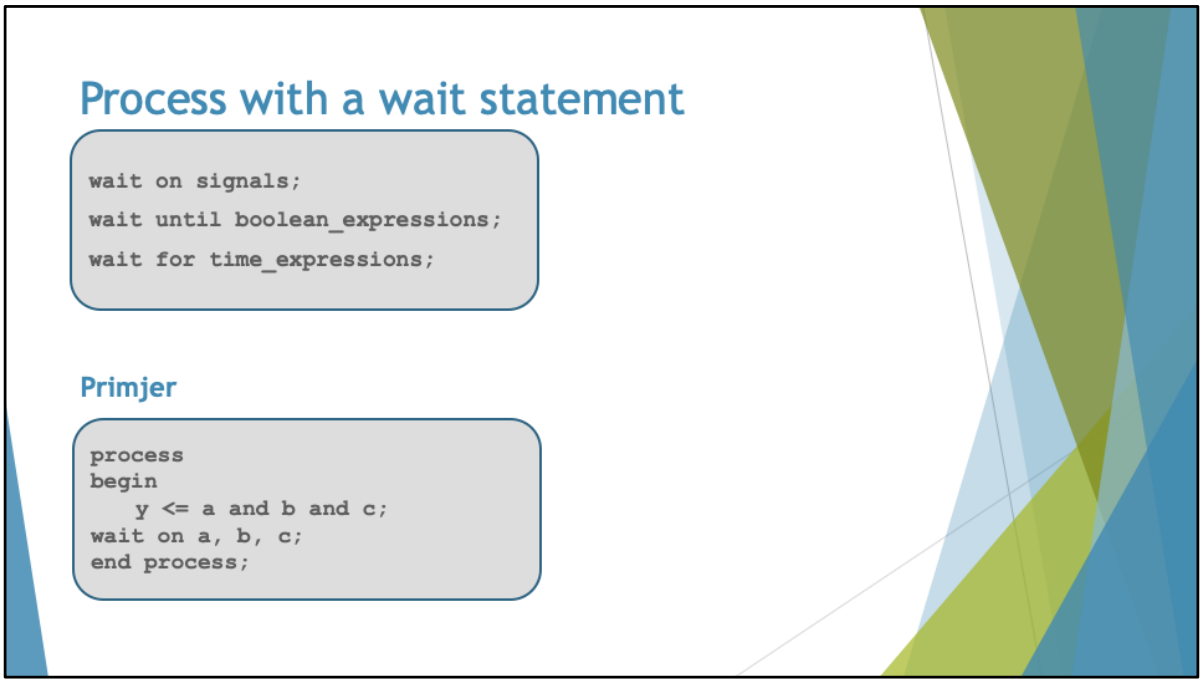


Sensitivity list je lista signala na koje je proces “osjetljiv”. Sekcija declarations se odnosi na deklaracije koje su lokalne za proces. Dok VHDL proces možda podsjeća na funckije ili procedure tradicionalnih programskih jezika, ponašanje VHDL procesa se značajno razlikuje. VHDL proces se ne poziva od strane druge rutine. Ponaša se kao dio kola, koji je aktivan, ili neaktivan (suspended). VHDL proces je aktivan kada neki od signala iz sensitivity liste promijeni vrijednost, kao što kolo odgovara na ulazni signal. Kada se proces jednom aktivira, izrazi unutar njega će se izvršavati sekvencijalno do kraja procesa. Nakon toga, proces je neaktivan, do naredne promjene signala iz sensitivity liste..

Process with sensitivity list

```
signal a,b,c,y : std_logic; -- in architecture declaration
. . .
process(a, b, c)
begin
    y <= a and b and c;
end process;
```

Kod kombinacionih kola izlaz zavisi od svih ulaza. Dakle, potrebno je sve ulaze navesti u sensitivity listi.



Proces sa wait izrazom sadrži jedan ili više wait izraza, ali ne sadrži sensitivity listu. Proces se izvršava dok ne dođe do wait izraza, tada se suspenduje. Po promjeni a ili b ili c, proces se aktivira, izvršava do kraja, ponovo započinje, sve do wait izraza gdje se suspenduje.
Uz pomoć wait izraza može se izvršiti vremenski kompleksno modelovanje, kao i modelovanje sekvencijalnih događaja.

Sequential signal assignment statement

```
signal_name <= value_expression;
```

Primjer

```
a,b,c,d,y: std_logic;
. . .
process(a,b,c,d)
begin
    y <= a or c;
    y <= a and b;
    y <= c and d;
end process;
```

⟺

```
a,b,c,d,y: std_logic;
. . .
process(a,b,c,d)
begin
    y <= c and d;
end process;
```

Concurrent conditional i selected signal assignment statements se ne mogu koristiti u okviru procesa.
Vrijednost signalu se dodjeljuje na kraju procesa. Treba izbjegavati višestruko dodjeljivanje vrijednosti signalu u okviru procesa. Jedini izuzeci su default-ne vrijednsoti u if i case izrazima.

Variable assignment statement

```
variable_name := value_expression;
```

Primjer

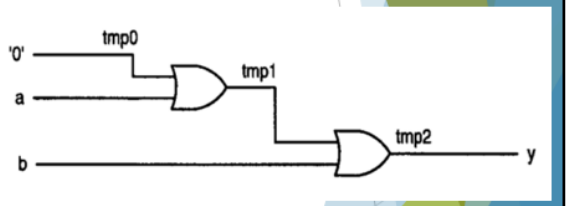
```
signal a,b,y: std_logic;
. . .
process(a,b)
    variable tmp: std_logic;
begin
    tmp := '0';
    tmp := tmp or a;
    tmp := tmp or b;
    y <= tmp;
end process;
```

Dodjela vrijednsoti varijabli se izvršava trenutno. Varijable su lokalne za proces.

Variable assignment statement

Primjer

```
process(a,b)
    variable tmp0, tmp1, tmp2: std_logic;
begin
    tmp0 := '0';
    tmp1 := tmp0 or a;
    tmp2 := tmp1 or b;
    y <= tmp2;
end process;
```



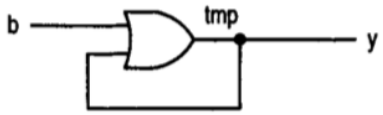
Za potrebe sinteze, ovdje se varijable mogu tretirati kao provodne linije. Varijable generalno treba koristiti samo ukoliko je nemoguće određenu karakteristiku sistema opisati signalom.

Variable assignment statement

Primjer

```
signal a,b,y,tmp: std_logic;
. . .
process(a,b,tmp)
begin
    tmp <= '0';
    tmp <= tmp or a;
    tmp <= tmp or b;
    y <= tmp;
end process;
```

```
signal a,b,y,tmp: std_logic;
. . .
process(a,b,tmp)
begin
    tmp <= tmp or b;
    y <= tmp;
end process;
```



Signal ne može biti lokalno vezan za proces.

IF statement

```
if boolean_expr_1 then
    sequential statements;
elsif boolean_expr_2 then
    sequential_statements;
elsif boolean_expr_3 then
    sequential_statements;
. . .
else
    sequential_statements;
end if;
```

Primjer: Multiplekser 4 u 1

```
architecture if_arch of multiplekser is
begin
    process(a,b,c,d,s)
    begin
        if(s = "00") then
            x <= a;
        elsif(s = "01") then
            x <= b;
        elsif(s = "10") then
            x <= c;
        else
            x <= d;
        end if;
    end process;
end if_arch;
```

Primjer: Binary decoder

```
architecture if_arch of binary_decoder is
begin
    process(s)
    begin
        if(s = "00") then
            x <= "0001";
        elsif(s = "01") then
            x <= "0010";
        elsif(s = "10") then
            x <= "0100";
        else
            x <= "1000";
        end if;
    end process;
end if_arch;
```

Input	Output
s	x
00	0001
01	0010
10	0100
11	1000

Primjer: Priority encoder

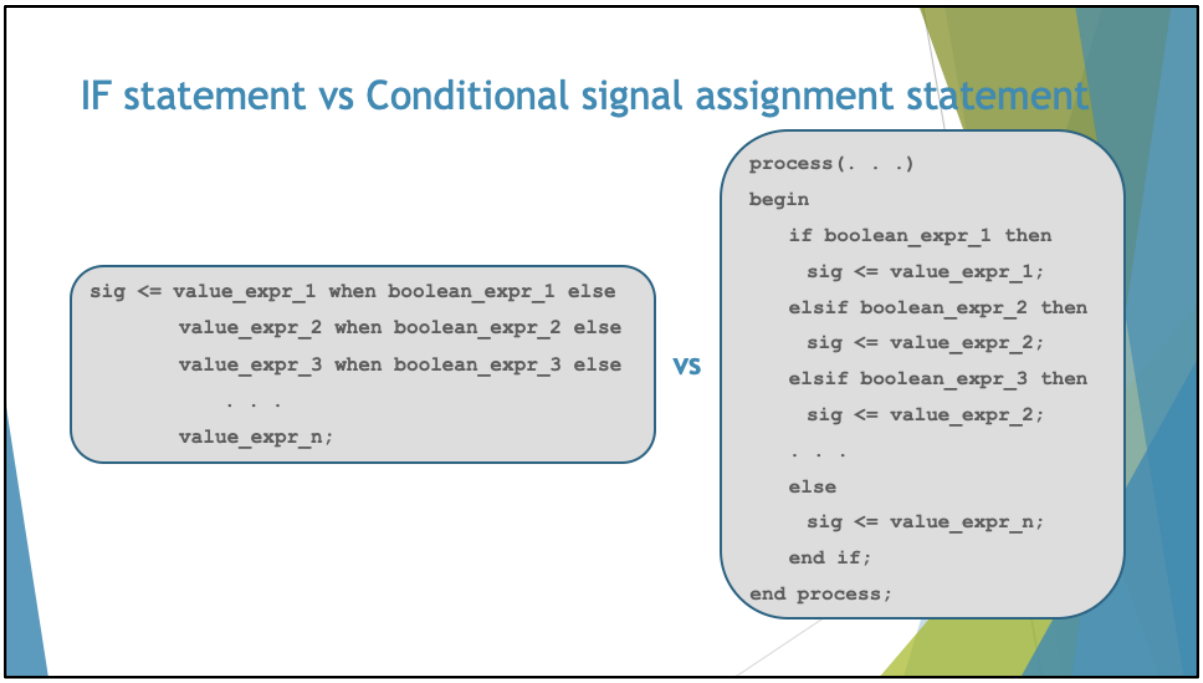
```
architecture if_arch of binary_decoder is
begin
    process(r)
    begin
        if(r(3) = '1') then
            code <= "11";
        elsif(r(2) = '1') then
            code <= "10";
        elsif(r(1) = '1') then
            code <= "01";
        else
            code <= "00";
        end if;
    end process;
    active <= r(3) or r(2) or r(1) or r(0);
end if_arch;
```

Input	Output	
r	code	active
1---	11	1
01--	10	1
001-	01	1
0001	00	1
0000	00	0

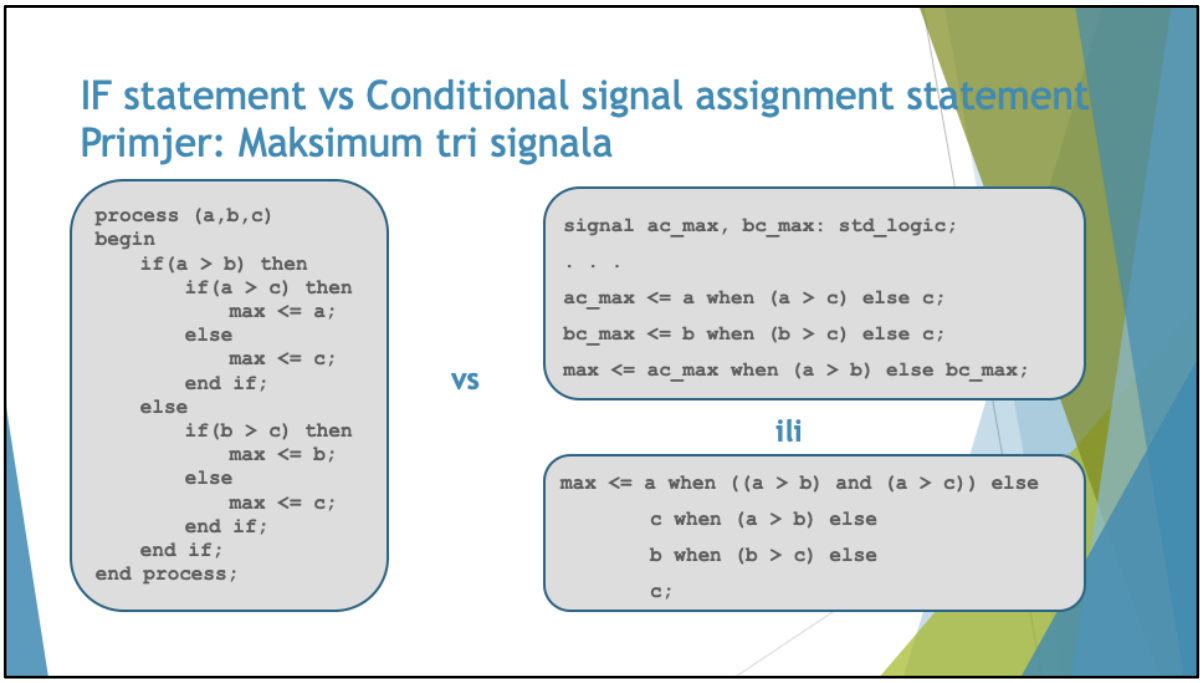
Primjer: Simple ALU

```
architecture if_arch of Simple_ALU is
    signal src0s, src1s: signed (7 downto 0);
begin
    src0s <= signed(src0);
    src1s <= signed(src1);
    process(ctrl, src0, src1, src0s, src1s)
    begin
        if(ctrl(2) = '0') then
            result <= std_logic_vector(src0s + 1);
        elsif(ctrl(1 downto 0) = "00") then
            result <= std_logic_vector(src0s + src1s);
        elsif(ctrl(1 downto 0) = "01") then
            result <= std_logic_vector(src0s - src1s);
        elsif(ctrl(1 downto 0) = "10") then
            result <= std_logic_vector(src0 and src1);
        else
            result <= src0 or src1;
        end if;
    end process;
end if_arch;
```

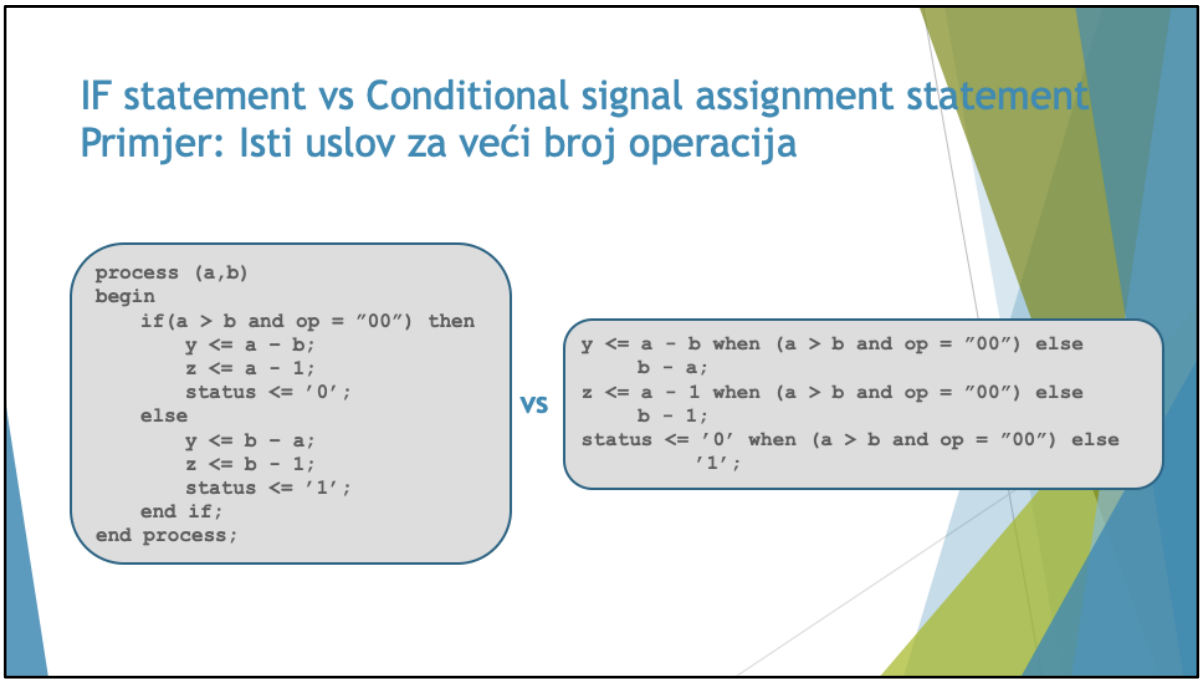
Input	Output
ctrl	result
0--	src0 + 1
100	src0 + src1
101	src0 - src1
110	src0 and src1
111	src0 or src1



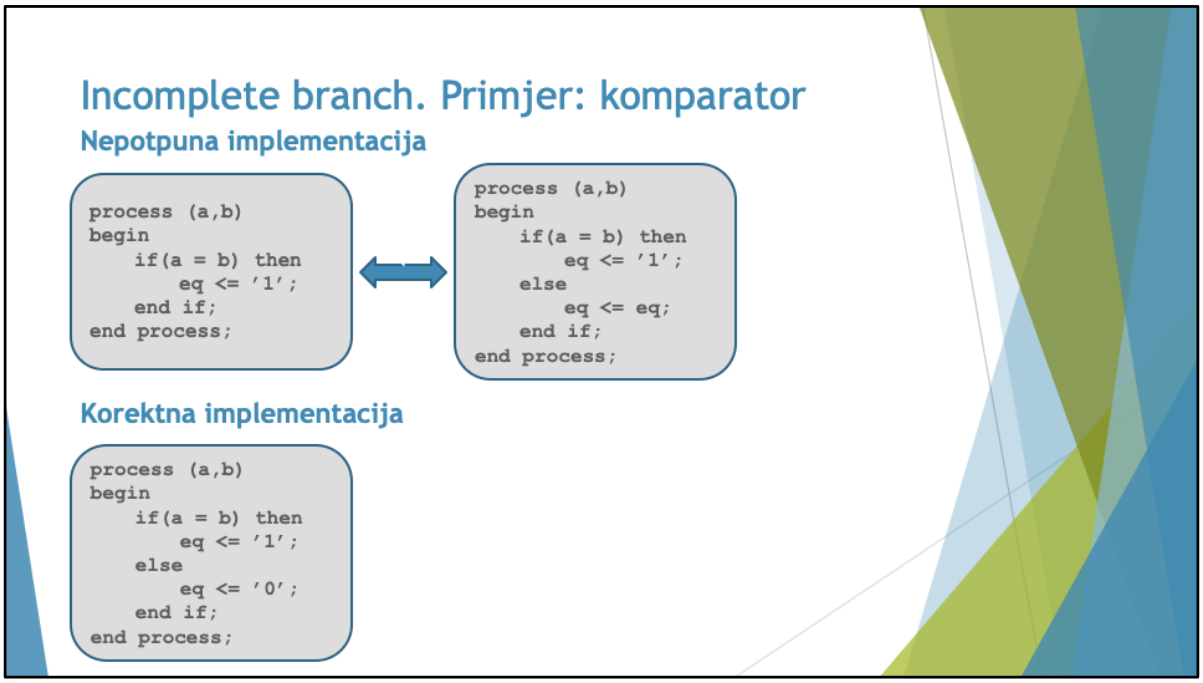
Ukoliko sekvencijalni izrazi u okviru if izraza sadrže samo dodjelu vrijednosti jednom signalu, ova dva pristupa (if statement i conditional signal assignment statement) su ekvivalentni. Jednakost je istinita jedino u ovako jednostavnim slučajevima. IF izraz je mnogo generalniji jer jedna grana može sadržati sekvencu sekvencijalnih izraza. Pravlina i disciplinovana upotreba if izraza može kod učiniti čak i efikasnijim.



Iako je kod na bazi conditional signal assignement statement kraći, nije deskriptivan i nije jednostavan za razumijevanje, kao što je slučaj sa nested if statement.



U ovom primjeru, potrebno je koristiti 3 conditional signal assignment izraza, za razliku od IF izraza gdje istu if-then-else strukturu koriste tri signala



Prema VHDL semantici, grananje je opciono i vrijednost signalu ne mora biti dodijeljena u svakoj grani. Međutim, iako je sintaksički ispravno, izostavljanje ovoga tipa može dovesti do uvođenja neželjenih memorijskih elemenata (latches).
U VHDL-u, jedino je **then** grana obavezna dok ostale grane mogu biti izostavljene. Ukoliko je **else** grana izostavljena (primjer komparatora), VHDL semantika specificira da se signal eq nije mijenjao, odnosno, zadržao je prethodnu vrijednost. Kreirano kolo sadrži nepotrebne memorijske elemente (interna stanja).

Incomplete signal assignment. Primjer: komparator

Nepotpuna implementacija

```
process (a,b)
begin
    if(a > b) then
        gt <= '1';
    elsif (a = b) then
        eq <= '1';
    else
        lt <= '1';
    end if;
end process;
```

Korektna implementacija

```
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;
```

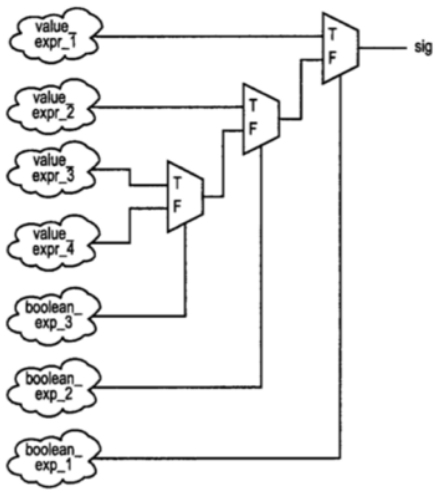
ili

```
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;
```

Sintatički je korektno signalu dodijeliti vrijednost u pojedinim granama, a u ostalim ne. Prethodno dovodi do uvođenja neželjenih memorijskih elementa u sistem.
VHDL semantika specificira da će signal zadržati prethodnu vrijednost dok mu druga nije dodijeljena.
Dobra praksa je dodjela default vrijednosti na početku procesa.

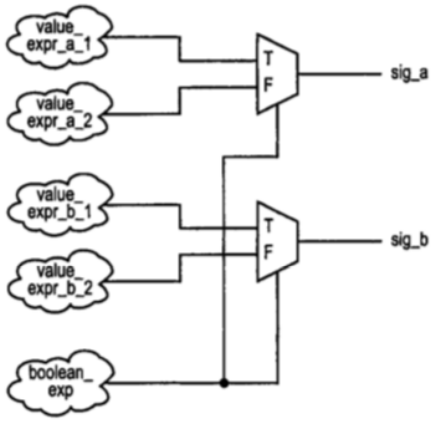
Konceptualna implementacija

```
if boolean_expr_1 then
    sig <= value_expr_1;
elsif boolean_expr_2 then
    sig <= value_expr_2;
elsif boolean_expr_3 then
    sig <= value_expr_3;
else
    sig <= value_expr_4;
end if;
```



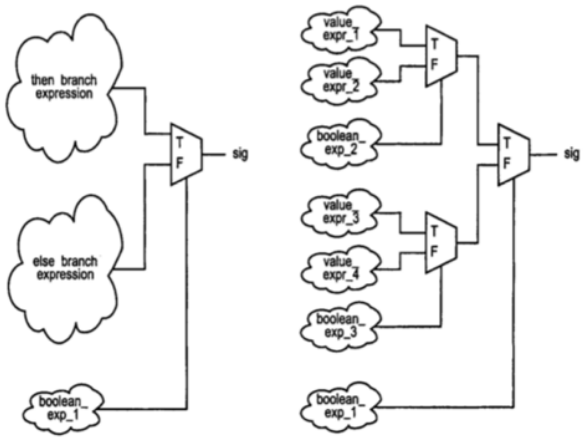
Konceptualna implementacija

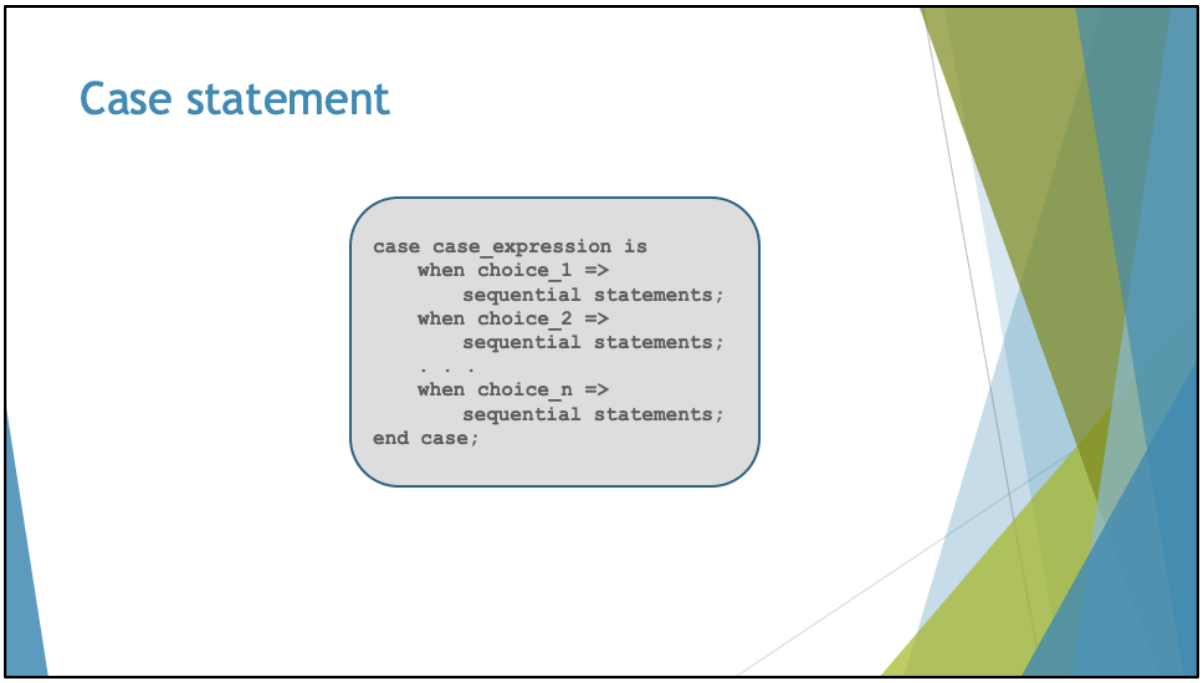
```
if boolean_expr then
    sig_a <= value_expr_a_1;
    sig_b <= value_expr_b_1;
else
    sig_a <= value_expr_a_2;
    sig_b <= value_expr_b_2;
end if;
```



Konceptualna implementacija

```
if boolean_expr_1 then
    if boolean_expr_2 then
        sig_a <= value_expr_1;
    else
        sig_a <= value_expr_2;
    end if;
else
    if boolean_expr_3 then
        sig_a <= value_expr_3;
    else
        sig_a <= value_expr_4;
    end if;
end if;
```





na osnovu case_expression se vrši odabir seta sekvencijalnih izraza. Izbori choice_i moraju biti međusobno isključivi i sveobuhvatni.

Primjer: Multiplekser 4 u 1

```
architecture case_arch of multiplekser is
begin
   process(a,b,c,d,s)
   begin
      case s is
         when "00" =>
            x <= a;
         when "01" =>
            x <= b;
         when "10" =>
            x <= c;
         when others =>
            x <= d;
      end case;
   end process;
end case_arch;
```

Primjer: Binary decoder

```
architecture case_arch of binary_decoder is
begin
    process(s)
    begin
        case s is
            when "00" =>
                x <= "0001";
            when "01" =>
                x <= "0010";
            when "10" =>
                x <= "0100";
            when others =>
                x <= "1000";
        end case;
    end process;
end case_arch;
```

Input	Output
s	x
00	0001
01	0010
10	0100
11	1000

Primjer: Priority encoder

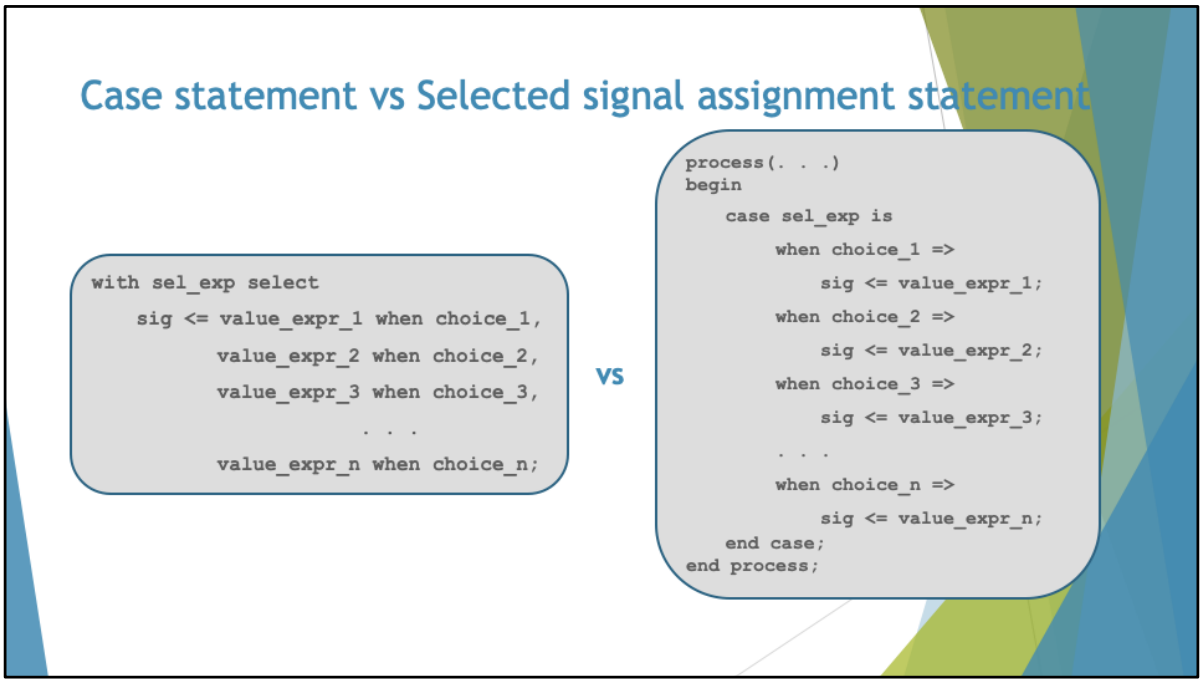
```
architecture case_arch of binary_decoder is
begin
    process(r)
    begin
        case r is
            when "1000"|"1001"|"1010"|"1011"|
                 "1100"|"1101"|"1110"|"1111" =>
                code <= "11";
            when "0100"|"0101"|"0110"|"0111" =>
                 code <= "10";
            when "0010"|"0011" =>
                code <= "01";
            when others =>
                code <= "00";
        end case;
    end process;
    active <= r(3) or r(2) or r(1) or r(0);
end case_arch;
```

Input	Output	
r	code	active
1---	11	1
01--	10	1
001-	01	1
0001	00	1
0000	00	0

Primjer: Simple ALU

```
architecture case_arch of Simple_ALU is
    signal src0s, src1s: signed (7 downto 0);
begin
    src0s <= signed(src0);
    src1s <= signed(src1);
    process(ctrl, src0, src1, src0s, src1s)
    begin
        case strl is
            when "000"|"001"|"010"|"011" =>
                result <= std_logic_vector(src0s + 1);
            when "100" =>
                result <= std_logic_vector(src0s + src1s);
            when "101" =>
                result <= std_logic_vector(src0s - src1s);
            when "110" =>
                result <= std_logic_vector(src0 and src1);
            when others =>
                result <= src0 or src1;
        end case;
    end process;
end case_arch;
```

Input	Output
ctrl	result
0--	src0 + 1
100	src0 + src1
101	src0 - src1
110	src0 and src1
111	src0 or src1



Ukoliko sekvencijalni izrazi u okviru case izraza sadrže samo dodjelu vrijednsoti jednom signalu, ova dva pristupa (case statement i selected signal assignment statement) su ekvivalentni. Jednakost je istinita jedino u ovako jednostavnim slučajevima. Case izraz je mnogo generalniji jer jedna grana može sadržati sekvencu sekvencijalnih izraza.

Incomplete signal assignment. Primjer: priority encoder

Nepotpuna implementacija

```
process (a)
begin
    case a is
        when ”100”|”101”|”110”|”111” =>
            high <= ”1”;
        when ”010”|”011” =>
            middle <= ”1”;
        when others =>
            low <= ”1”;
    end case;
end process;
```

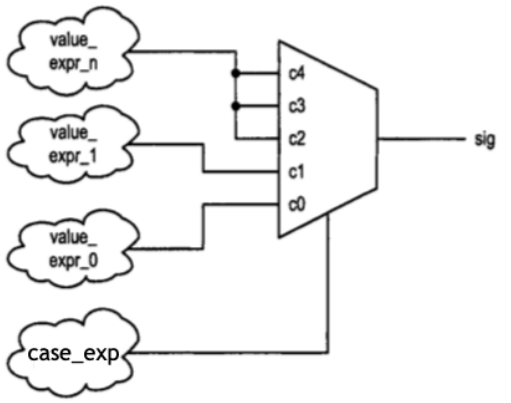
Korektna implementacija

```
process (a)
begin
    high <= ”0”;
    middle <= ”0”;
    low <= ”0”;
    case a is
        when ”100”|”101”|”110”|”111” =>
            high <= ”1”;
        when ”010”|”011” =>
            middle <= ”1”;
        when others =>
            low <= ”1”;
    end case;
end process;
```

Ukoliko je a = “111”, prvi when uslov je ispunjen i high će imati vrijednost 1. Međutim, kako signalima middle i low nije dodijeljene vrijednost, zadržaće prethodnu vrijednost. Kreirani su nepotrebni memorijski elementi. Iz tog ralzloga potrebno je dodijeliti vrijednost svim signalima u okviru svakog when uslova.

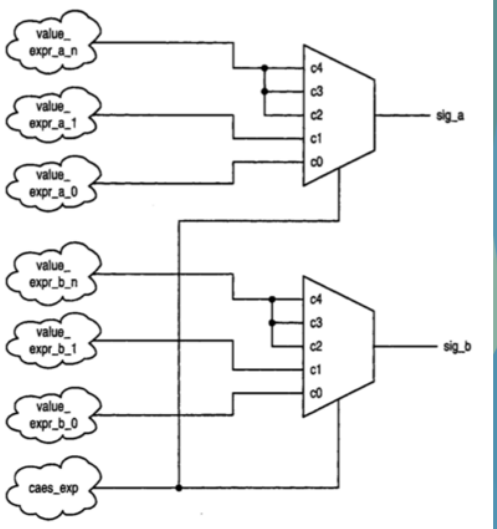
Konceptualna implementacija

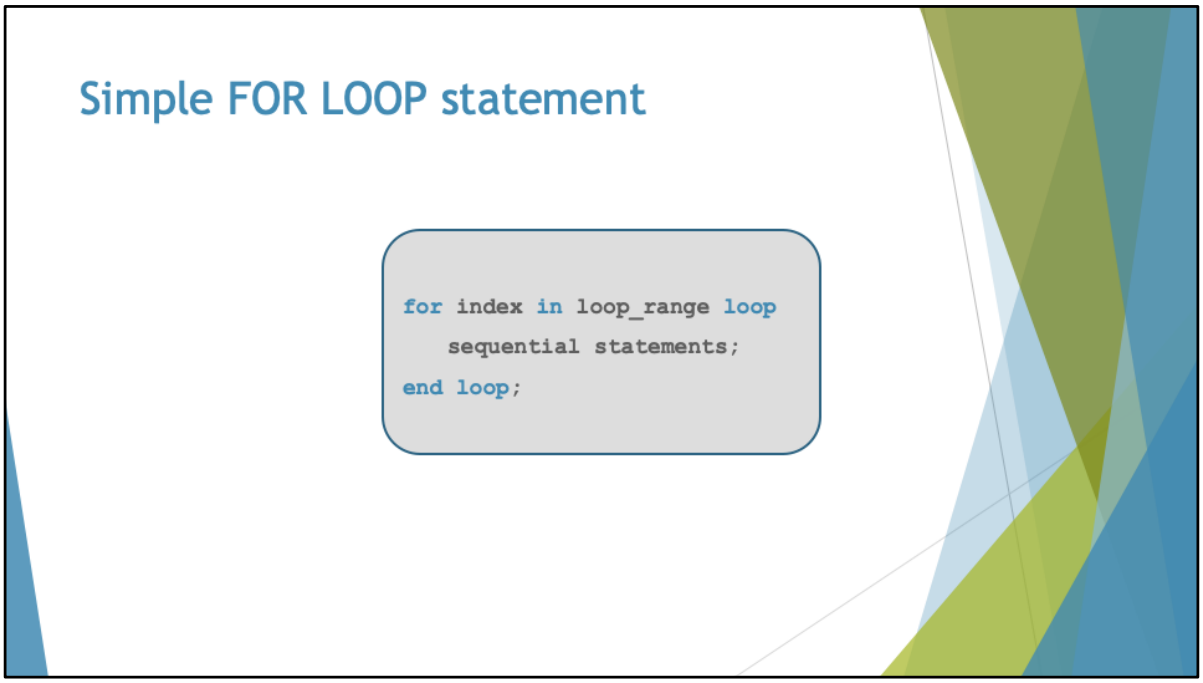
```
case case_exp is
    when c0 =>
        sig <= value_expr_0;
    when c1 =>
        sig <= value_expr_1;
    when others =>
        sig <= value_expr_n;
end case;
```



Konceptualna implementacija

```
case case_exp is
    when c0 =>
        sig_a <= value_expr_a_0;
        sig_b <= value_expr_b_0;
    when c1 =>
        sig_a <= value_expr_a_1;
        sig_b <= value_expr_b_1;
    when others =>
        sig_a <= value_expr_a_n;
        sig_b <= value_expr_b_n;
end case;
```



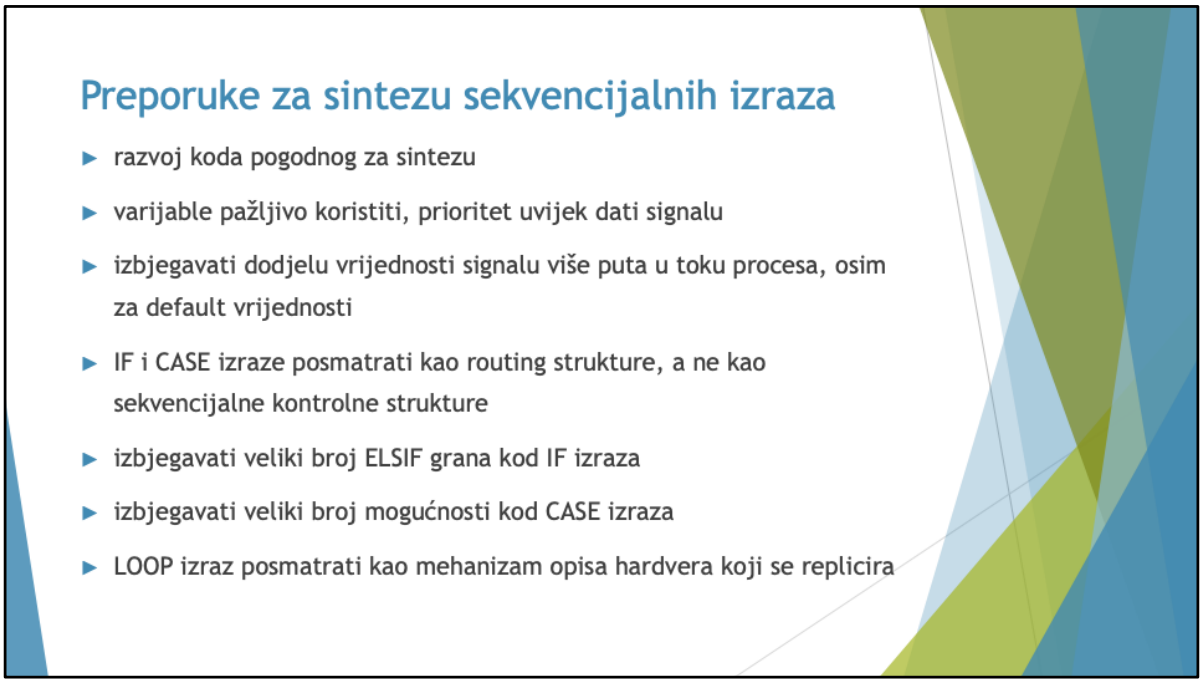


loop_range mora biti određen u trenutku sinteze i ne može zavisiti od ulaznih signala.

Primjer: Bitwise XOR

```
library ieee;
use ieee.std_logic_1164.all;
entity bit_xor is
    port
        (
        a, b : in std_logic_vector (3 downto 0);
        y : out std_logic_vector (3 downto 0)
        );
end bit_xor;
architecture demo_arch of bit_xor is
    constant WIDTH : integer := 4;
begin
process(a, b)
begin
    for i in (WIDTH-1) downto 0 loop
        y(i) <= a(i) xor b(i);
    end loop;
end process;
end demo_arch;
```

Dati kod je samo demonstracija upotrebe **for** petlje. Ista operacija se naravno mogla ostvariti jednom naredbom **y <= a xor b**;
Uz pomoć petlje moguće je kontrolisati replikaciju hardvera.



Priroda concurrent i sekvencijlanih izraza se u potpunosti razlikuje. Concurrent statements su modelovani na bazi hardvera, dakle, postoji jasna veza između njih i hardverske strukture. Sa druge strane, sekvencijalni izrazi su namijenjeni da opišu apstraktno ponašanje sistema, i za neke konstrukcije je teško odrediti hardversku paralelu. Sekvencijalni izrazi su značajno fleksibilniji.

Preporuke za sintezu kombinacionih kola

- uključiti sve ulaze u sensitivity listu
- uključiti sve uslove kod IF izraza, kako se ne bi kreirali neželjeni memorijski elementi
- izlaznom signalu dodijeliti vrijednost u okviru svake grane IF i CASE izraza, kako se ne bi kreirali neželjeni memorijski elementi
- Dodijeliti default vrijednost svakom signalu na početku procesa