Sekvenčné riadiace konštrukcie

Vetvenie (Conditional Control)

Príkaz if

Všeobecný opis syntaxe *príkazu if:*

if_statement ::=

[*if*_label:] **if** condition **then**

sequence_of_statements

{ **elsif** condition **then**

sequence_of_statements }

[**else**

sequence_of_statements **]**

end if [*if*_label] ;

Ilustračný príklad:

Buffer_Process:

process

begin

if Enable='0'**then**

Output <='Z' **after** Delay;

else

Output <=Input **after** Delay;

end if;

wait on Input;

end process;

Vo všeobecnosti platí nasledujúca ekvivalencia:

podmienené priradenie signálu:

```
target <= options
    waveform1 when condition1 else
        :
        :
     waveformN-1 when conditionN-1
else
    waveformN;
```

zodpovedajúci príkaz **if**:

```
if condition1 then
   target <= options waveform1;
        :
        :
elsif conditionN-1 then
   target <= options waveformN-1;
else
   target <= options waveformN;
end if ;
```

```
architecture Conditional of AND_Gate is
begin
    Y <= transport
     '1' after Delay when A = '1' and B = '1' else
       '0' after Delay;
end Conditional;

architecture Conditional_Equivalent of AND_Gate is
begin
    process (A, B)
    begin
        if A = '1' and B = '1' then
            Y <= transport '1' after Delay;
        else
            Y <= transport '0' after Delay;
        end if;
    end process;
end Conditional_Equivalent;
```

Pomocou príkazu **if** môžeme opísať aj správanie sa stráženého príkazu priradenia ekvivalentným príkazom process.

```
entity Guard_gate is
  port( A, EN: in Bit;
        Y: out Bit bus);
end Guard_gate;
architecture DF of Guard_gate is
signal temp_Y: bit bus;
begin
    B: block (EN = '1') is
    begin
        temp_Y <= guarded A ;
    end block;
    Y<= temp_Y;
end DF;
```

```
architecture Alg of Guard_gate is
signal temp_Y: bit bus;
begin
    B: process (A, EN) is
    begin
      if (EN = '1') then
         temp_Y <= A ;
      else
         temp_Y <= null;
      end if;
     Y<= temp_Y;
end Alg;
```

syntaxe *príkazu case*:

```
case_statement ::=  [case_label:]   case expression is
                    when choices => sequence_of_statements
                    { when choices => sequence_of_statements }
                    end case [case_label] ;
choices ::= choice { | choice}
choice  ::=   simple_expression
            | discrete_range
            | element_simple_name
            | others
```

```
Select_Process:
   process
    begin
     case X is
        when 1 => Out1 <= 0;
        when 2|3 => Out1 <= 1;
        when others => Out1 <= 2;
     end case;
    end process;
```

Výberové priradenie signálu:

```
with expression select
target <= options
waveform1 when choices1,
          :
          :
waveformN when choicesN;
```

zodpovedajúci príkaz **case**:

```
case expression is
   when choices1 =>
target <= options waveform1;
          :
          :
   when choicesN =>
target <= options waveformN;
end case ;
```

Príklad dvoch ekvivalentných architektúr:

```
entity Decoder is
    port ( Enable: in Bit;
           Sel:Bit_Vector(2 downto 0);
           DOut:out Bit_Vector(7 downto 0));
    constant Delay:Time:=5 ns;
end Decoder;
```

```
architecture Selected of Decoder is
begin
 guard_block: block (Enable='1') is
 begin
    with Sel select
    DOut <= guarded
    "00000001" after Delay when "000",
    "00000010" after Delay when "001",
    "00000100" after Delay when "010",
    "00001000" after Delay when "011",
    "00010000" after Delay when "100",
    "00100000" after Delay when "101",
    "01000000" after Delay when "110",
    "10000000" after Delay when "111";
 end block;
end Selected;
```

```
architecture SelectedEquivalent of Decoder is
begin
 process(Sel, Enable)
 begin
   if (Enable='1') then
     case Sel is
     when "000" => DOut <= "00000001" after Delay;
     when "001" => DOut <= "00000010" after Delay;
     when "010" => DOut <= "00000100" after Delay;
     when "011" => DOut <= "00001000" after Delay;
     when "100" => DOut <= "00010000" after Delay;
     when "101" => DOut <= "00100000" after Delay;
     when "110" => DOut <= "01000000" after Delay;
     when "110" => DOut <= "10000000" after Delay;
     end case;
 else
    DOut <=null;
 end if;
end process;
end SelectedEquivalent;
```

Syntax *príkazu cyklu:*

```
loop_statement ::=
   [loop_label:]
   [ iteration_scheme ] loop
      sequence_of_statements
   end loop [ loop_label ] ;
iteration_scheme ::=      while condition
                        | for identifier in discrete_range

exit [loop_label] [when condition];
```

```
P1:
 process
  variable A :Integer=0;
  variable B :Integer=1;
 begin
  Loop1:
  loop
   A:=A+1;
   B:=20;
   Loop2:
   loop
    if B< (A*A) then
      exit Loop2;
    end if;
    B:=B-A;
   end loop Loop2;
   exit Loop1 when A>=10;
  end loop Loop1;
  wait;
 end process;
```

```
P1:
 process
  variable B :Integer=1;
 begin
  Loop1:
  for A in 1 to 10 loop
   B:=20;
   Loop2:
   while B >= (A*A) loop
    B:=B-A;
   end loop Loop2;
  end loop Loop1;
  wait;
 end process;
```

Príkaz next

next [*loop*_label] [**when** condition];

```
P1: process
  variable B :Integer=1;
 begin
  Loop1:
  for A in 1 to 10 loop
   B:=20;
   Loop2:
   loop
    next Loop1 when B < (A*A);
    B:=B-A;
end loop Loop2;
  end loop Loop1;
  wait;
 end process;
```

Príkaz null

Príkaz ASSERTION

```
-- paralelný príkaz ASSERTION
entity SRFF is
  port (S, R : in Bit; Q, Qbar : out Bit);
begin
   SRFF_Constraint_Check:
   assert not (S = '1' and R = '1')
      report "Both S and R equal to '1'"
         severity Error;
end SRFF;
```

```
-- sekvenčný príkaz ASSERTION:
entity SRFF is
    port(S, R:in Bit; Q, Qbar:out Bit);
begin
  SRFF_Constraint_Check:
  process (S, R)
  begin
     assert not (S = '1' and R = '1')
       report "Both S and R equal to '1'"
          severity Error;
  end process;
end SRFF;
```

Príkaz return

return_statement ::= [label:] **return** [expression];

Príkaz volania procedúry

procedure_call_statement ::=
[*procedure_call*_label:] *procedure*_name (association_list);

paralelnému volaniu procedúry:

```
my_proc(in_signal, out_signal, in_variable);
```

je ekvivalentný proces:

```
process
begin
    my_proc(in_signal, out_signal, in_variable);
    wait on in_signal;
end process;
```

alebo proces:

```
process (in_signal)
begin
  my_proc (in_signal, out_signal, variable);
end process;
```

Opis štruktúry

pozostáva z opisu ***súčiastok*** a ich vzájomného ***prepojenia***.

Základná konštrukcia opisu štruktúry - ***príkaz vytvorenia inštancie komponentu*** (component instantiation statement).

Príkaz vytvorenia inštancie komponentu

Opis syntaxe:

component_instantiation_statement ::=

*instantiation*_label: instantiated_unit

[**generic map** (*generic*_association_list)]

[**port map** (*port*_assciation_list)] ;

instantiated_unit ::= [**component**] component_identifier

| **entity** entity_identifier [(architecture_identifier)]

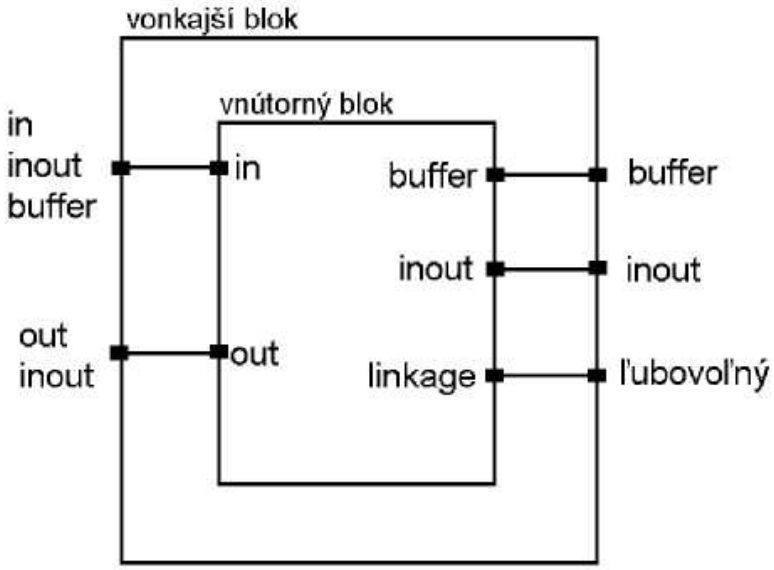
Skutočný (actual) objekt musí mať objektovú triedu **signal** a môže to byť:

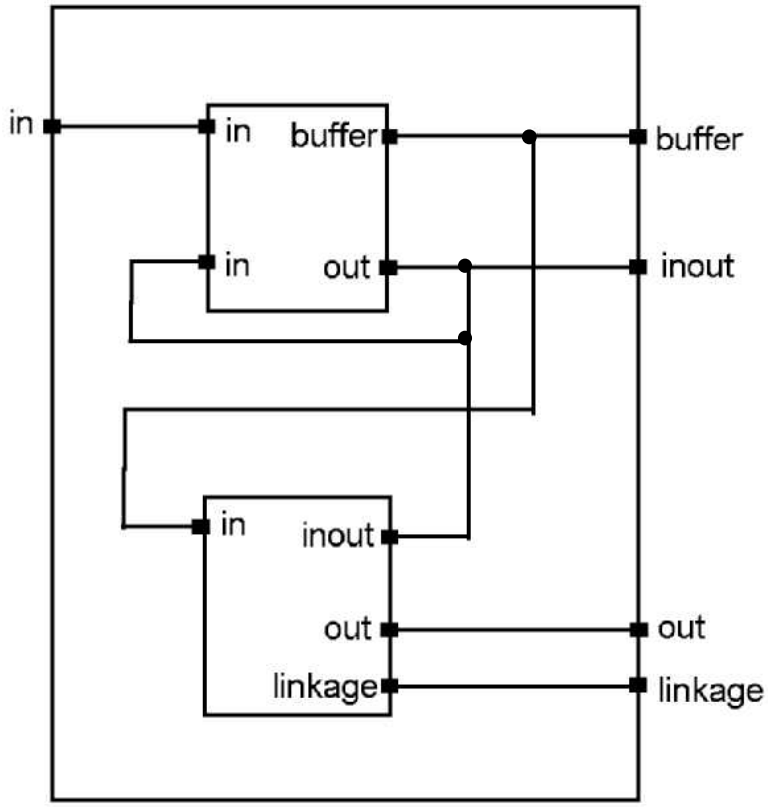
- Signál, deklarovaný v deklarácii signálu.
- Formálny port
- Statický výraz – len v prípade portu v režime **in**;

Pre priradenie skutočných objektov lokálnym portom platia obmedzenia:

- Typ skutočného objektu musí byť rovnaký, ako typ lokálneho portu.
- Ak je lokálny port čitateľný, potom aj skutočný objekt musí byť čitateľný. Naopak, ak sa dá do lokálneho portu zapisovať, potom aj do skutočného objektu sa musí dať zapisovať.
- Skutočný objekt, ktorého typ nie je rezolučný, nesmie byť priradený k lokálnemu portu s režimom **out** alebo **inout**, ak by toto priradenie mohlo spôsobiť, že skutočný objekt bude mať viac než jeden zdroj.
- K lokálnemu portu s režimom **buffer** môže byť priradený iba formálny port s režimom **buffer**, alebo signál, ktorý nemá žiaden iný zdroj.

Možné pripojenia lokálnych portov na formálne:





Pravidelné štruktúry

Objekty Generic

Typicky sa používajú na

- parametrizovanie časovania,
- rozsahu podtypov,
- počtu naväzovaných subkomponentov,
- veľkosti objektov typu pole,
- na dokumentáciu fyzikálnych charakteristík (napr. teplota),
- na riadenie príkazu generate

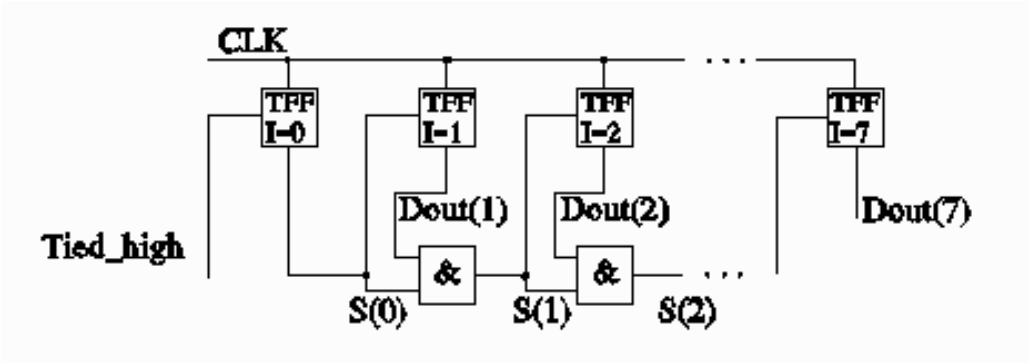
```
entity Decoder is
    generic (N: Positive);
    port(Enable: Bit;
        Sel: Bit_vector(N-1 downto 0);
        Dout:out Bit_vector( (2**N)-1 downto 0));
end Decoder;
```

Príkaz generate (Generate Statement)

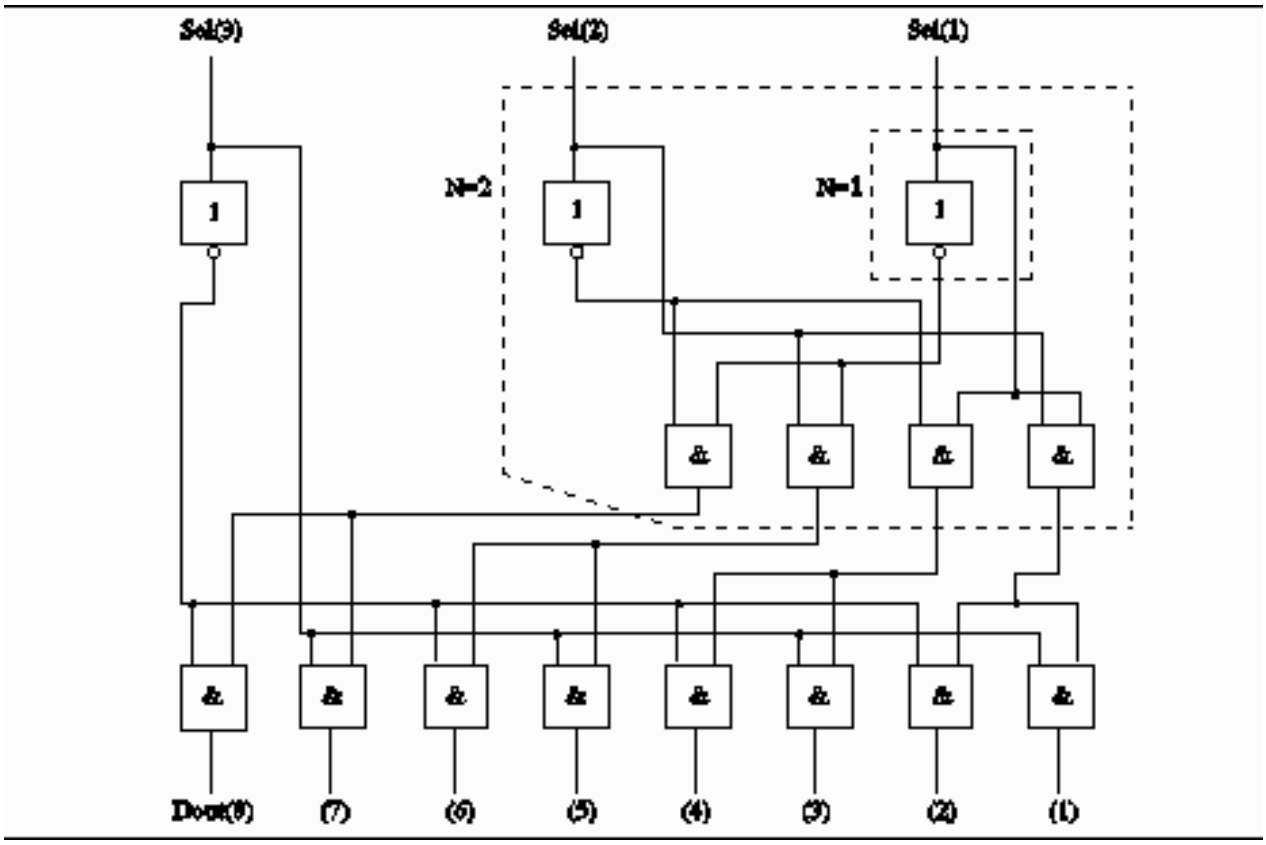
Všeobecný opis syntaxe *príkazu generate* má tvar:

```
generate_statement ::=
    generate_label:
        generation_scheme generate
            [{block_declarative item}
        begin]
            {concurrent_statement}
        end generate [generate_label];

generation_scheme ::=
    for identifier in discrete_range
    | if condition
```

```
entity Invert_8 is
    port (Inputs: Bit_vector (1 to 8);
        Outputs: out Bit_vector (1 to 8));
end Invert_8;
architecture Invert_8 of Invert_8 is
    component Inverter
        port (I1: Bit; O1: out Bit);
    end component;
begin
    G: for I in 1 to 8 generate
        Inv: Inverter port map (Inputs(I), Outputs(I));
    end generate;
end Invert_8;
```



```
entity Counter is
    port (CLK: Bit; Carry: Bit;
          Dout: buffer Bit_vector (7 downto 0));
end Counter;

architecture Counter of Counter is
    component TFF
        port (CLK: Bit; T: Bit; Q: buffer Bit);
    end component;
    component And2
        port (I1, I2: Bit; O1: out Bit);
    end component;
 signal S: Bit_vector (7 downto 0);
 signal Tied_high: Bit := '1';
begin
    G1: for I in 7 downto 0 generate
        G2: if I=7 generate
            TFF_7: TFF port map (CLK, S(I-1), Dout(I));
        end generate;
        G3:if I=0 generate
            TFF_0:TFF port map (CLK, Tied_high, Dout(I));
            S(I)<=Dout(I);
        end generate;
        G4: if I>0 and I<7 generate
            And_1: And2 port map (S(I-1), Dout(I), S(I));
            TFF_1: TFF port map (CLK, S(I-1), Dout(I));
        end generate;
    end generate;
end Counter;
```




```
entity Decoder is
    generic (N: Positive);
    port(Sel: Bit_vector(1 to N);
                Dout: out Bit_vector(1 to 2**N));
end Decoder;
architecture Generic_structure of Decoder is
    signal Sel_bar:Bit;
    component And2
        port (I1, I2: Bit; O1: out Bit);
    end component;
    component Inverter
        port (I1: Bit; O1: out Bit);
    end component;
    component Decoder
        generic (N: Positive);
        port(Sel: Bit_vector (1 to N); Dout: out Bit_vector(1 to 2**N));
    end component;
```

```
begin
    Inverter_select: Inverter port map (Sel(N), Sel_bar);
    Not_recursive: if N=1 generate
        Dout(N) <= Sel(N);
        Dout(2**(N-1)+1) <= Sel_bar;
    end generate;
    Recursive: if N>1 generate
        B1: block
            signal Temp: Bit_vector (1 to 2**(N-1));
        begin
        N_minus_1: Decoder generic map (N-1)
                                    port map (Sel(1 to N-1),Temp);
        For_each_output_from_N_minus_1:
              for I in 1 to 2**(N-1) generate
                And_each_N_minus_1_with_Sel:
                    And2 port map(Temp(I), Sel(N), Dout(2*(I-1)+1));
                And_each_N_minus_1_with_Sel_bar:
                    And2 port map(Temp(I), Sel_bar, Dout(2*I));
              end generate;
        end block;
    end generate;
end Generic_structure;
```