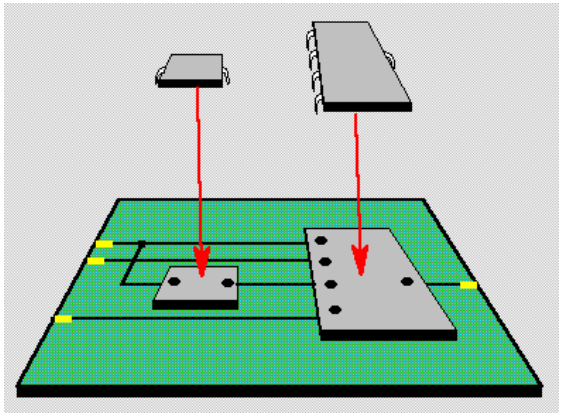
Komponenty (funkční bloky)

- vhodné pro skládání hotových funkčních bloků,
- komponenta musí být definována entitou,
- tato entita musí mít přiřazenu architekturu,
- komponenta musí být deklarována v deklarační části architektury nebo v package,
- lokální signály propojující komponenty musí být deklarovány v deklarační části architektury.



Deklarace komponenty

Syntaxe deklarace komponenty:

COMPONENT jméno_komponenty [**IS**]
[**GENERIC** (jméno_parametru : integer := hodnota {;
jméno_parametru : integer := hodnota }) ;]
PORT (jméno_signálu {, jméno_signálu} : [mód] typ {;
jméno_signálu {, jméno_signálu} : [mód] typ });
END COMPONENT [jméno_komponenty];

Použití komponenty podle jména

Syntaxe použití - asociace podle jména (named association):

jméno_instance : jméno_komponenty
[**GENERIC MAP** (formální_jméno => aktuální_jméno
{, formální_jméno => aktuální_jméno})]
PORT MAP (formální_jméno => aktuální_jméno
{, formální_jméno => aktuální_jméno});

Jméno_signálu při definici komponenty může být následně použito jako „formální_jméno“ při použití komponenty.

Příklad:
U1 : decode PORT MAP (r => rd, op => in, w =>wr);

Použití komponenty podle polohy

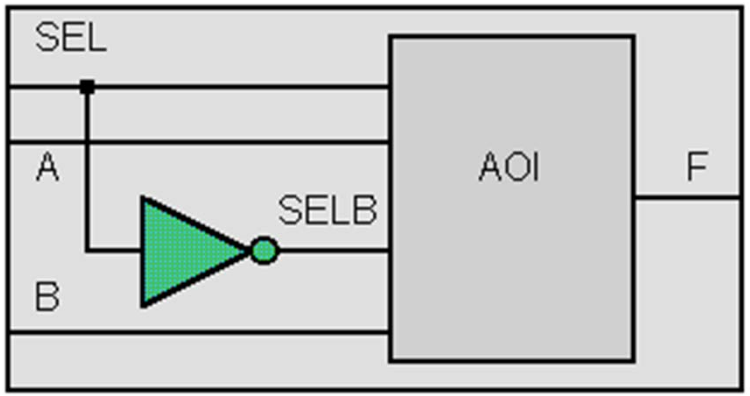
Syntaxe použití - asociace podle polohy (positional association):

jméno_instance : jméno_komponenty
[**GENERIC MAP** (aktuální_jméno {, aktuální_jméno})]
PORT MAP (aktuální_jméno {, aktuální jméno });

Příklad:
demx23 : demultipl GENERIC MAP (8) PORT MAP (in, rd, wr);

Komponenta - příklad

```
ENTITY mux2 IS
     PORT (SEL, A, B : IN std_logic;
                       F : OUT std_logic);
END mux2;
ARCHITECTURE structure OF mux2 IS
COMPONENT INV
     PORT ( A : IN std_logic ;
               F : OUT std_logic ) ;
END COMPONENT ;
COMPONENT AOI
     PORT ( A, B, C, D : IN std_logic ;
               F : OUT std_logic ) ;
END COMPONENT ;
SIGNAL SELB : std_logic ;
BEGIN                                   -- tělo architektury (vložení komponent)
G1 : INV PORT MAP ( SEL, SELB ) ;
G2 : AOI PORT MAP (SEL, A, SELB, B, F ) ;
END structure ;
```



Přímé vkládání entity

- (direct entity instantiation) - dovoluje standard VHDL-93;
  někdy označováno také jako *přímé vkládání komponenty;*
- není nutné deklarovat komponentu (v deklarační části architektury je možné vynechat části COMPONENT … END COMPONENT;)

Příklad (ekvivalentní předešlému příkladu):

```
ARCHITECTURE structure OF mux2 IS
SIGNAL SELB : std_logic;
BEGIN
G1 : ENTITY work.INV  PORT MAP ( SEL, SELB );
G2 : ENTITY work.AOI  PORT MAP (SEL, A, SELB, B, F );
END structure;
```

Podprogramy

Dva typy: *funkce* a *procedury*

- jsou syntetizovány při každém volání;
- pro viditelnost podprogramů platí stejná pravidla jako pro datové typy;
- příkazy uvnitř funkcí a procedur se vykonávají sekvenčně (obdoba procesu – možnost použití proměnných, příkazů IF, CASE, LOOP, …);
- proměnné uvnitř funkce nebo procedury si nezachovávají svoji hodnotu - jsou vždy inicializovány při jejich volaní (na rozdíl od procesu).

Funkce

- podprogram, který vrací hodnotu (datový typ);
- funkce mají vstupní operandy a jeden výstupní operand;
- typ hodnoty může být skalární nebo složený;
- ve funkci nemohou být deklarovány signály (pouze konstanty a proměnné – jejich platnost je omezena pouze na funkci).

Syntaxe definice funkce:

FUNCTION jméno_funkce (parametry) **RETURN** typ **IS**

deklarace

BEGIN

sekvenční příkazy

END jméno_funkce ;

Funkce - příklad

```
ENTITY decoder IS
    PORT ( input : IN BIT_VECTOR(0 TO 1);
           output : OUT BIT_VECTOR(0 TO 3));
END decoder;
ARCHITECTURE arch_fce OF decoder IS
    FUNCTION bit_to_int (bit_in : IN BIT) RETURN INTEGER IS
    BEGIN
        IF bit_in = '0' THEN RETURN 0;
        ELSIF bit_in = '1' THEN RETURN 1;
        ELSE RETURN 0; END IF;
    END bit_to_int;
BEGIN
    ...
    State := bit_to_int (input(0));      -- volání funkce
    ...
END arch_fce;
```

Posuvný registr – pomocí funkce

```
PACKAGE ops IS
    FUNCTION shift (
        shftreg : bit_vector(3 DOWNTO 0);
        bitin : bit )
    RETURN bit_vector;
END ops;
PACKAGE BODY ops IS
    FUNCTION shift ( shftreg : bit_vector(3 DOWNTO 0); bitin : bit )
        RETURN bit_vector IS
        VARIABLE result : bit_vector(3 DOWNTO 0);
        BEGIN
            result(2 DOWNTO 0) := shftreg(3 DOWNTO 1);
            result(3) := bitin;
        RETURN result;
    END shift;
END ops;
```

Procedury

- podprogram, který modifikuje vstupní parametry;
- módy parametrů mohou být: IN, OUT, INOUT (implicitně IN);
- proceduře lze předávat signály, konstanty i proměnné;
- z procedury lze volat další proceduru.

Syntaxe definice procedury:

PROCEDURE jméno_procedury (seznam_parametrů) **IS**

deklarace

BEGIN

sekvenční příkazy

END jméno_procedury;

Volání procedury

- asociace parametrů podle jména nebo podle polohy;

Příklady volání procedury:

```
PROCEDURE examine_data (my_port : IN string;
                        read_data : OUT bit_vector(0 TO 23);
                        prop_delay : IN time := 1 ns);
```

a) examine_data (“shifter”, data_contents, 25 ns);

b) examine_data (my_port => “shifter”,
prop_delay => 25 ns,
read_data => data_contents);

c) examine_data (“shifter”, data_contents);
-- defaultně je použito prop_delay = 1 ns

Volání procedury

v citlivých proměnných uvést všechny IN a INOUT parametry

Dva ekvivalentní příklady:

```
ARCHITECTURE a OF e IS
BEGIN
    aprocedure (in_sig1, inout_sig2, const1);
END a;

ARCHITECTURE a OF e IS
BEGIN
    PROCESS (in_sig1, inout_sig2)
    BEGIN
        aprocedure (in_sig1, inout_sig2, const1);
    END PROCESS;
END a;
```

Procedura - příklad

```
ENTITY add IS
    PORT ( arg1, arg2 : IN BIT_VECTOR(3 DOWNTO 0);
           result : OUT BIT_VECTOR(3 DOWNTO 0); carry : OUT BIT );
END add ;

ARCHITECTURE arch_proced OF add IS

    PROCEDURE add_carry (SIGNAL a1, a2 : IN BIT_VECTOR(3 DOWNTO 0);
    SIGNAL result : OUT BIT_VECTOR(3 DOWNTO 0); SIGNAL carry : OUT BIT) IS
    VARIABLE temp : BIT_VECTOR(4 DOWNTO 0);
    BEGIN
        temp := (“0” & a1) + (“0” & a2);
        carry <= temp(4);
        result <= temp(3 DOWNTO 0);
    END add_carry;

BEGIN
    …
    add_carry (arg1, arg2, result, carry);   -- volání procedury
    …
END arch_proced;
```

Definice stavových automatů

Popis stavového automatu musí obsahovat:

- stavovou proměnnou (současný a následující stav)
- hodinový signál (pouze jediný)
- specifikace změny stavu (Q_n => Q_{n+1})
- specifikace výstupů
- reset (synchronní nebo asynchronní) – není nutné
- každá definice by měla mít svůj proces, příp. procesy

a) **jednoprocesorový popis automatu**
(registrová část je popsána procesem)

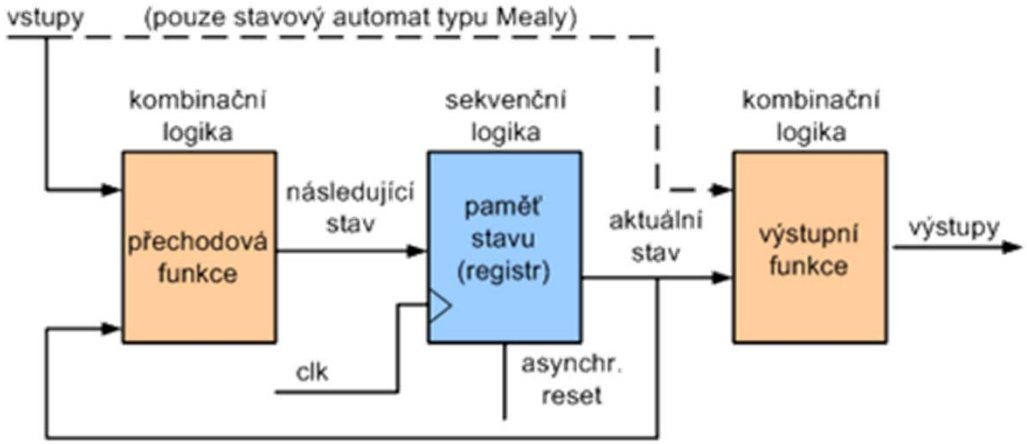
b) **dvouprocesorový popis automatu**
(kombinační i registrová část je popsána jedním procesem)

c) **tříprocesorový popis automatu** (proces popisující hodiny, proces pro změnu stavu a proces pro definování výstupů)

Stavové automaty

Automaty typu Mealy a Moore

- oba typy automatů jsou mezi sebou převoditelné,
- výstup Mealyho automatu se mění ihned po změně vstupů,
- Mooreho automat může vést na větší počet vnitřních stavů,
- Mealyho automat může mít složitější přechodovou a výstupní funkci,
- výstupní funkce je u obou automatů kombinační → možnost hazardů → za výstupní funkci se někdy doplňují registry.



Stavové proměnné

- musí být signály nebo proměnné,
- musí být následujícího typu:
  enumerated, integer, bit, bit_vector, boolean
- nesmí být „port“ signál,
- operace nad stavovou proměnnou je limitována na “=“ a “/=“,
- při behaviorálním stylu popisu se obecně ve VHDL nedefinuje chování automatu v nevyužitých stavech,
- pokud automat nemá nevyužité stavy, mluvíme o *spolehlivém* (safe) stavovém automatu,
- ošetření nevyužitých stavů ale způsobuje nárůst logiky.

Kódování vnitřních stavů

- Pro zakódování „*n*“ vnitřních stavů je třeba „*k*“ bitů $(2^k \geq n \geq k)$ (předpokládáme jednoznačné kódování stavů),
- kódování vnitřních stavů vznikne při syntéze (lze ovlivnit nastavením návrhového systému),
- volba kódování má významný vliv na složitost a dynam. parametry.

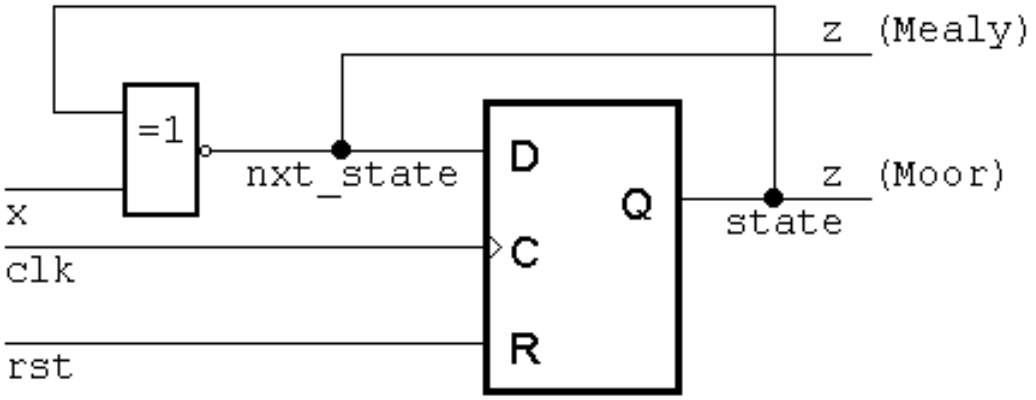
Kódování	Sekvenční	Gray	Johnson	One-hot	Almost O-h	One-cold
Stav0	0000	0000	00000	0000000001	000000000	1111111110
Stav1	0001	0001	00001	0000000010	000000001	1111111101
Stav2	0010	0011	00011	0000000100	000000010	1111111011
Stav3	0011	0010	00111	0000001000	000000100	1111110111
Stav4	0100	0110	01111	0000010000	000001000	1111101111
Stav5	0101	0111	11111	0000100000	000010000	1111011111
Stav6	0110	0101	11110	0001000000	000100000	1110111111
Stav7	0111	0100	11100	0010000000	001000000	1101111111
Stav8	1000	1100	11000	0100000000	010000000	1011111111
Stav9	1001	1101	10000	1000000000	100000000	0111111111

Moorův stavový automat - příklad

```
ENTITY moor IS
    PORT (clk, x, rst : IN    std_logic;
                  z : OUT  std_logic);
END moor;
ARCHITECTURE ar_moor OF moor IS
    TYPE states IS (s0, s1);
    SIGNAL state : states := s0;
    SIGNAL nxt_state : states := s0;
BEGIN
clkd: PROCESS (clk, rst)   --  Proces pro zachyceni hodnoty stavu
BEGIN
    IF (rst = '0') THEN
    state <= s0;
    ELSIF (clk'EVENT AND clk = '1') THEN
    state <= nxt_state;
    END IF;
END PROCESS clkd;
```

Moorův stavový automat (pokrač.)

```
state_trans: PROCESS (state, x) -- Proces k určeni následujícího stavu
BEGIN
    CASE state IS
        WHEN s0 => IF (x = '1') THEN nxt_state <= s1;
                                ELSE nxt_state <= s0;  END IF;
        WHEN s1 => IF (x = '1') THEN nxt_state <= s0;
                                ELSE nxt_state <= s1;  END IF;
    END CASE;
END PROCESS state_trans;
output: PROCESS (state)   -- Proces pro definici výstupu
BEGIN
    CASE state IS
        WHEN s0 => z <= '0';
        WHEN s1 => z <= '1';
    END CASE;
END PROCESS output;
END ar_moor;
```



Mealyho stavový automat - příklad

Kromě procesu definujícího výstupní hodnoty je vše stejné s Moorovým automatem

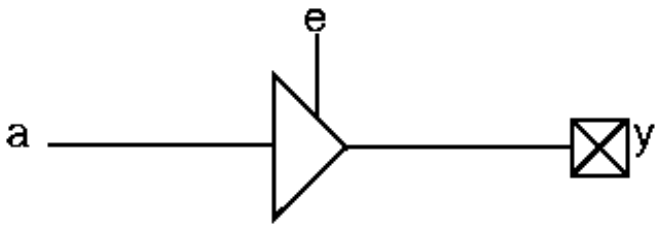
```
output: PROCESS (state, x)     -- Proces pro definici výstupu
BEGIN
    CASE state IS
        WHEN s0 => IF (x = '1') THEN  z <= '1';
                   ELSIF (x = '0') THEN  z <= '0';
                   ELSE z <= 'X';
                   END IF;
        WHEN s1 => IF (x = '1') THEN  z <= '0';
                   ELSIF (x = '0') THEN  z <= '1';
                   ELSE z <= 'X';
                   END IF;
    END CASE;
END PROCESS output;
```

Třístavové výstupy - příklad

```
LIBRARY  IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY tristate IS
PORT (e, a : IN std_logic;
          y : OUT std_logic);
END tristate;

ARCHITECTURE tri OF tristate IS
BEGIN
    PROCESS (e, a)
    BEGIN
          IF  e = '1'  THEN  y <= a;
          ELSE  y <= 'Z';  END IF;
    END PROCESS;
END tri;
```



Nesyntetizovatelné:

a <= b and 'Z';

if a = 'Z' then …

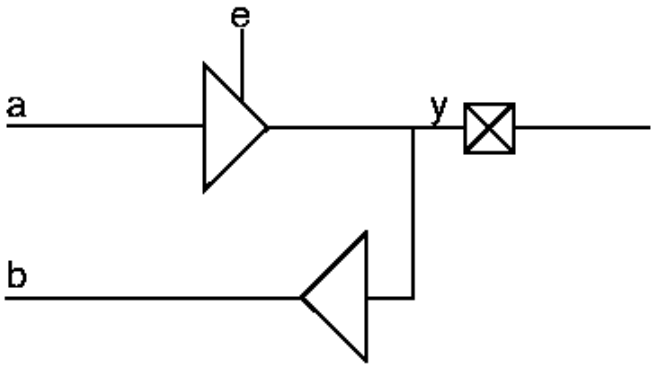
Proces je možné nahradit přiřazením: y <= a **WHEN** (e = '1') **ELSE** 'Z';

Obousměrná komunikace - příklad

```
LIBRARY  IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY bidir IS
PORT (    y      : INOUT  std_logic;
          e, a   : IN  std_logic;
          b      : OUT  std_logic );
END bidir;

ARCHITECTURE bi OF bidir IS
BEGIN
    PROCESS (e, a)  BEGIN
        CASE e IS
            WHEN '1' => y <= a;
            WHEN '0' => y <= 'Z';
            WHEN OTHERS => y <= 'X';
        END CASE;
    END PROCESS;
    b <= y;
END bi;
```



Zpoždění

- všechny přiřazovací příkazy ve VHDL představují určité zpoždění, lze ale i přesně specifikovat;
- zpoždění má jednu ze tří forem:
  - *Transport delay*
  - *Inertial delay*
  - *Delta delay* (standardní zpoždění, není-li specifikováno jinak)

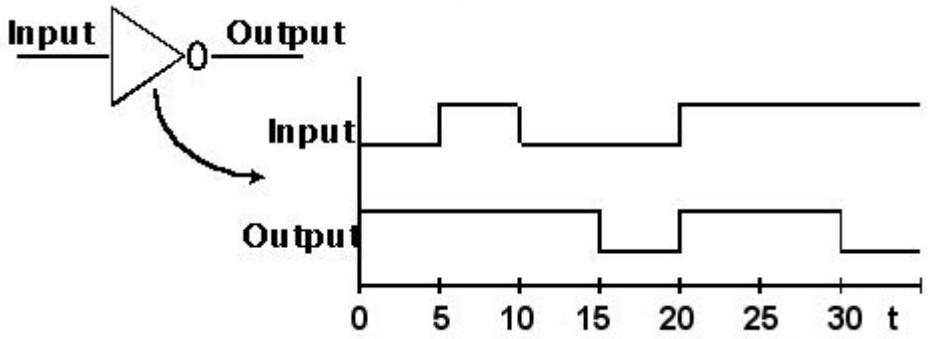
    output <= **NOT** input;
- zpoždění se uplatňuje u signálů (ne proměnných);
- zpoždění se nerespektuje při syntéze (pouze při simulaci).

Zpoždění typu Transport

Uvozeno klíčovým slovem TRANSPORT

- signál nabývá nové hodnoty po specifikovaném zpoždění

output <= **TRANSPORT** NOT input **AFTER** 10 ns;

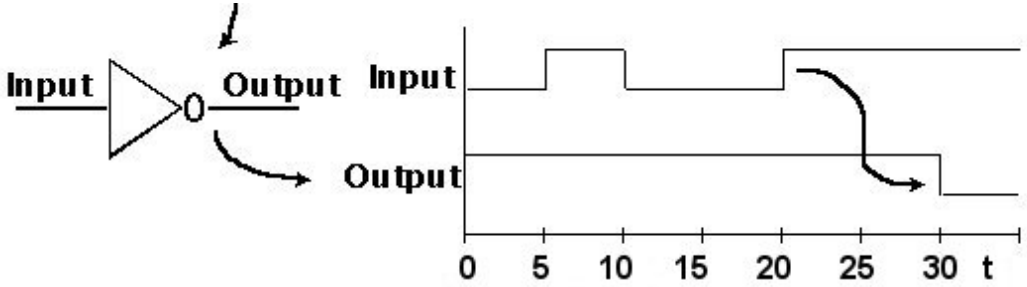


Zpoždění typu Inertial

INERTIAL zpoždění je defaultní, REJECT je volitelný

output <= **REJECT** 5 ns **INERTIAL** NOT input **AFTER** 10 ns;
- neprojde impuls kratší než 5 ns

output <= NOT input **AFTER** 10 ns;
- neprojde impuls kratší než 10 ns



Simulace ve VHDL (Testbench)

- Simulace na nejvyšších úrovních hierarchie (DTL – Design Top Level);
- na DTL aplikujeme budicí signály (stimuly) a sledujeme výstupní signály (response);
- stimuly píšeme ve VHDL
  - deklarace entity je prázdná,
  - deklarace komponenty a signálů je povinná,
  - u signálů definujeme časové průběhy;
- výhodou je přenositelnost mezi návrhovými systémy;
- některé simulátory vybaveny WYSIWYG prostředky pro editaci testbenchů;
- počáteční inicializace – lze specifikovat, implicitně - výčtový typ: první hodnota; integer: 0; real: 0.0; std_logic: ‘U’.

Testbench - příklad

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY test_mux4 IS END;                          -- prázdná entita
ARCHITECTURE bench OF test_mux4 IS
COMPONENT mux4                                    -- deklarace komponenty
     PORT ( sel :  IN  std_logic_vector(1 DOWNTO 0);
                a, b, c, d  : IN  std_logic;
                f : OUT std_logic );
END COMPONENT;
SIGNAL sel : std_logic_vector (1 DOWNTO 0);       -- vstupní a výstupní signály
SIGNAL a, b, c, d, f : std_logic;
BEGIN
  M : mux4 PORT MAP ( sel, a, b, c, d, f );       -- vložení instance komponenty
    sel <= "00", "01" AFTER 30 ns, "10" AFTER 60 ns,          --časové průběhy
    "11" AFTER 90 ns, "XX" AFTER 120 ns, "00" AFTER 130 ns;
    a <= 'X', '0' AFTER 10 ns, '1' AFTER 20 ns;
    b <= 'X', '0' AFTER 40 ns, '1' AFTER 50 ns;
    c <= 'X', '0' AFTER 70 ns, '1' AFTER 80 ns;
    d <= 'X', '0' AFTER 100 ns, '1' AFTER 110 ns;
END bench;
```