

Úvod do jazyka VHDL

Návrh číslicových systémů

2007-2008

Jan Kořenek

korenek@fit.vutbr.cz

Jak popsat číslicový obvod

- Slovně

„Navrhněte (číslcový) obvod, který spočte sumu všech členů dané posloupnosti“

- slovní vyjádření toho co má obvod dělat je pro člověka přirozené, avšak vyrobit podle něj obvod není možné

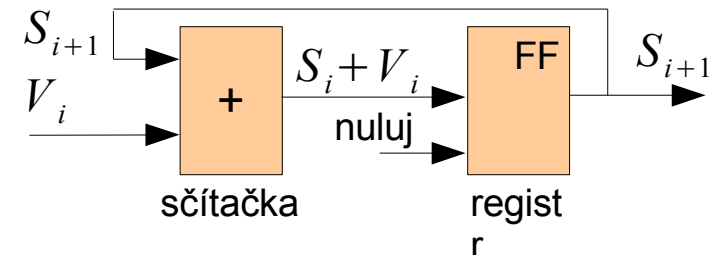
- Matematicky

- v současné době neexistují nástroje, které by umožnily automatizovaně bez úzké asistence člověka – návrháře fyzickou implementaci

$$S = \sum_{i=1}^N V_i$$

- Graficky pomocí schématu

- funkčních bloky a jejich propojení
- pro velké obvody pracné a nepřehledné



- Programovacím jazykem

- lze vytvořit popis chování obvodu v programovacím jazyku

HDL jazyky

- Na rozdíl od schematu návrhář popisuje funkci obvodu pomocí jazyka
 - Zařízení je možné **modelovat** a **simulovat**
 - **Proces syntézy umožňuje transformovat HDL popis do prvků cílové technologie** – syntéza je proces analogický kompilaci používané u programovacích jazyků
- V praxi se používají zejména jazyky **VHDL** a **Verilog**
 - Oba jazyky jsou mohou být vstupem procesu syntézy
 - VHDL dominuje v Evropě, Verilog v USA

Jazyk VHDL

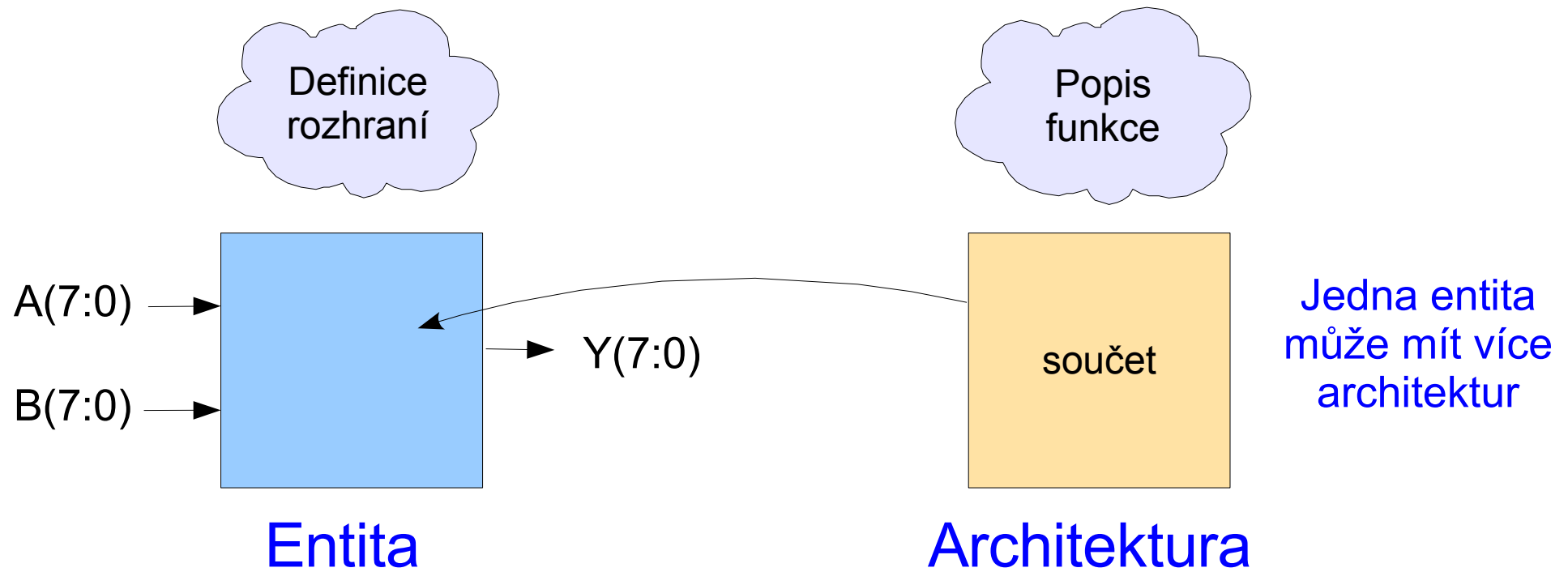
- Zkratka VHDL z akronymu:

VHDL = VHSIC Hardware Description Language

- VHSIC je zkratka pro Very High Speed Integrated Circuit
- Původně bylo VHDL vyvinuto pro vojenské účely ke specifikaci číslicových systémů, později se stalo IEEE standardem
- Jazyk VHDL není svázán s žádnou cílovou technologií
- Umožňuje tři základní úrovně popisu – behaviorální, strukturní a data flow popis
- Existuje spousta nástrojů umožňující syntézu nebo simulaci obvodů popsaných v jazyku VHDL

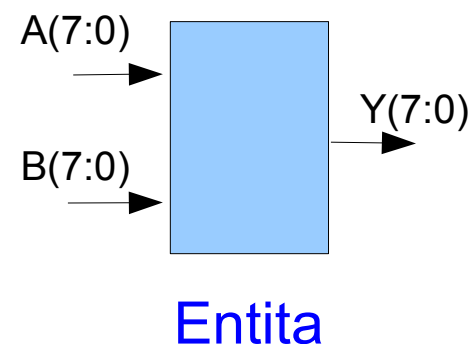
Číslicové zařízení ve VHDL

- VHDL popisuje číslicová zařízení a jednotlivé části zařízení pomocí **komponent**, které se popisují pomocí:
 - **Entita** definuje rozhraní komponenty
 - **Architektura** popisuje chování nebo strukturu komponenty



Entita

- Popisuje rozhraní mezi komponentou a okolím
- Rozhraní komponenty se skládá ze **signálů rozhraní** a **generických parametrů**
- Signály rozhraní mohou být podle směru v módu **IN**, **OUT** nebo **INOUT**



Příklad:

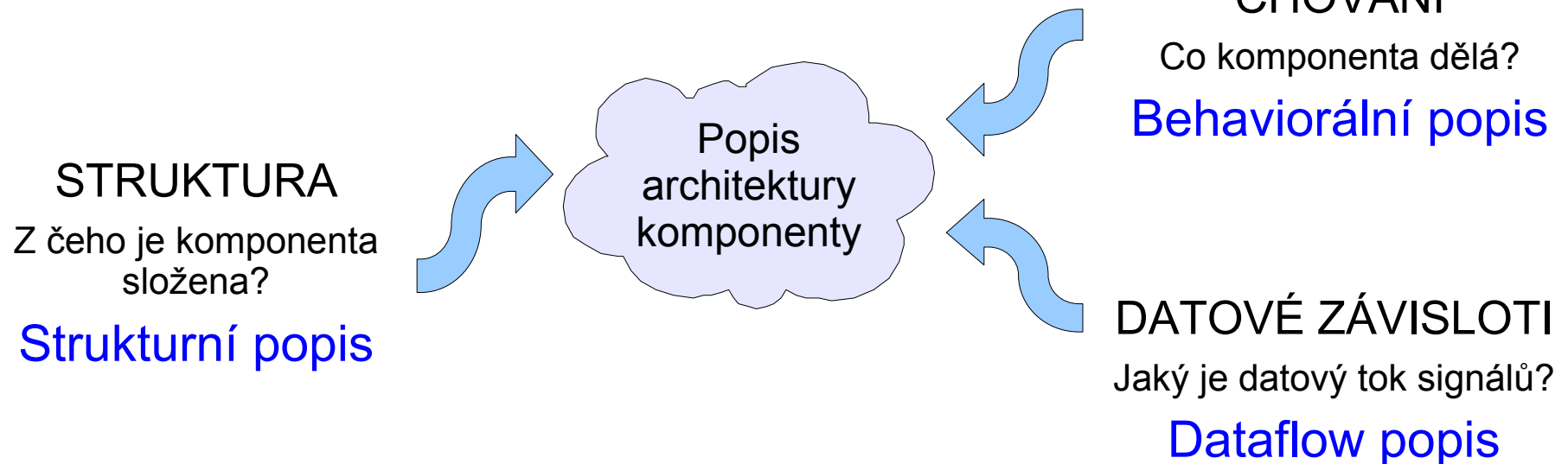
```
entity NAND is
  generic (DATA_WIDTH : integer :=8
  );
  port (A      : in  std_logic_vector(DATA_WIDTH-1 downto 0);
        B      : in  std_logic_vector(DATA_WIDTH-1 downto 0);
        Y      : out std_logic_vector(DATA_WIDTH-1 downto 0);
  );
end entity register;
```

parametry

signály rozhraní

Architektura

- Definuje chování nebo strukturu komponenty
- **Architektura je vždy svázána s entitou**, která definuje rozhraní s okolím
- Každá komponenta může být popsána na úrovni **struktury**, **chování** nebo **dataflow**
- Různé způsoby popisu je možné kombinovat



Architektura

Syntax:

```
Architecture name of entity_name is  
    Deklarační část  
begin  
    Sekce paralelních příkazů  
end architecture name;
```

Jméno entity

popisující rozhraní architektury.
Architekturu není možné použít s
jinou entitou, než je uvedena zde.

Deklarační část architektury

je vyhrazena pro deklaraci
signálů, konstant nebo typů
použitých uvnitř architektury.

Sekce paralelních příkazů

- Součástí sekce paralelních příkazů mohou být **instance komponent** nebo **procesy** vzájemně propojené signály
 - **Behaviorální popis** – architektura je složena z jednoho nebo více procesů
 - **Strukturní popis** – architektura obsahuje pouze instance komponent
 - V praxi se často používají oba přístupy i v rámci jedné architektury

Behaviorální popis (jeden proces)

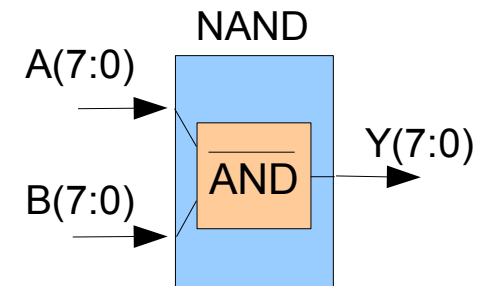
- Architektura složena z jednoho nebo více procesů
- Proces je popsán na úrovni **algoritmu** (příkazy, podmínky, cykly, atd.)
- Cílem je pouze popsat, jak se mění výstupy v závislosti na změnách vstupních signálů, **nemusí být zřejmá hardwarová realizace**

Příklad:

```
architecture behv of NAND is
begin

    nand_proc : process (A, B)
    begin
        Y <= not (A AND B);
    end process nand_proc;

end behv;
```



Proces `and_proc` přepočítá výstup **Y** vždy při změně signálu **A** nebo **B**

Dataflow popis

- Modeluje datové závislosti
- Zkrácený zápis chování pomocí paralelních příkazů uvnitř architektury:

- Přirazovací příkaz

```
Y <= NOT (A AND B);
```

- Podmíněný přirazovací příkaz

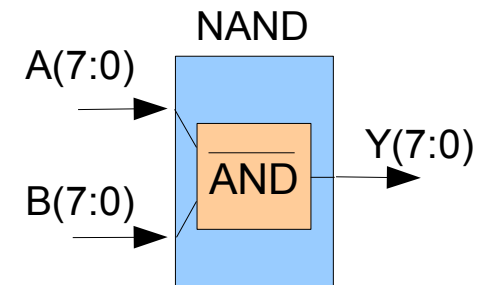
```
Y <= B when (A='1') else '0';
```

- Výběrový přirazovací příkaz

```
with S select    Y <= A when "0",  
                  B when "1",
```

Příklad:

```
architecture dataflow of NAND is  
begin  
    Y <= not (A AND B);  
end behv;
```



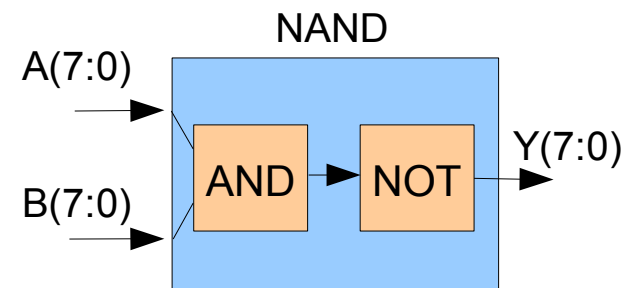
Behaviorální popis (více procesů)

- Architektura se může skládat z více procesů:
 - Procesy mohou číst/nastavovat vstupními/výstupními signály
 - Komunikace mezi procesy je realizována prostřednictvím signálů

Příklad:

```
architecture behv of NAND is
    signal ab_and : stdl_logic_vector(7 downto 0);
begin
    and_proc : process (A, B)
    begin
        ab_and <= A AND B;
    end process and_proc;

    not_proc : process (ab_and)
    begin
        Y <= not ab_and;
    end process not_proc;
end behv;
```



Procesy komunikují
prostřednictvím signálu
ab_and

Proces ve VHDL

Syntax:

```
name: process (sensitivity list)
    declarations
begin
    sequential statements
end process name;
```

Seznam citlivých proměnných

Kdykoliv se změní signál ze sensitivity listu, je spuštěn proces a vypočítají se nové hodnoty signálů

Deklarační část procesu
je vyhrazena pro deklaraci proměnných, konstant nebo typů použitých uvnitř procesu

Sekvenční příkazy

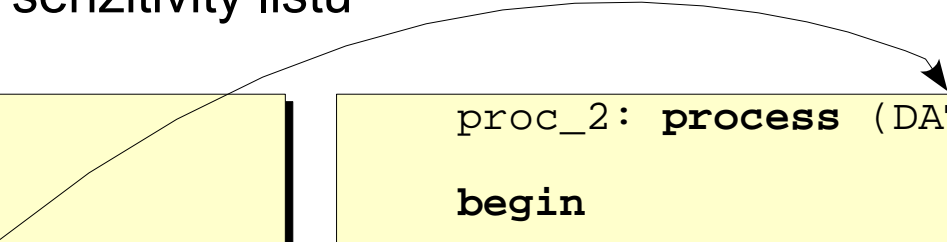
Program, který popisuje chování dané komponenty nebo její části. Na základě vstupních signálů a vnitřních proměnných program vypočítá hodnoty signálů

- Proces může popisovat chování celé komponenty nebo pouze její části
 - Architektury může obsahovat více procesů komunikujících vzájemně pomocí signálů

Senzitivity list procesu

- Senzitivity list procesu – proces je „spuštěn“ pokud dojde ke změně signálu uvedeného na senzitivity listu procesu
 - Po provedení sekvence příkazů je proces pozastaven a čeká se opět na změnu signálu ze senzitivity listu

```
proc_1: process
begin
    DATA <= "1010";
    wait for 10 ns;
    DATA <= "0101";
    wait for 10 ns;
end process proc_1;
```



```
proc_2: process (DATA)
begin
    statement1;
    statement2;
    statement3;
end process proc_2;
```

- Proces obsahující senzitivity list nemůže obsahovat příkaz WAIT
- Příkaz WAIT se používá zejména při tvorbě testbenche

Řídicí struktury v procesu

- Podmíněné vykonání příkazů (*if ... then ...*)

```
IF <condition> THEN <statements> END IF;
```

- Podmíněné vykonání příkazů s alternativou (*if ... then ... else ...* nebo *if ... then ... elsif ...*)

```
IF <condition> THEN <statements> [ELSIF <statements>]  
[ELSIF <statements>] END IF;
```

- Výběr více příkazů (*case ...*)

```
CASE <condition> IS WHEN <value> => <statements>  
[WHEN <value> => <statements>]  
END CASE;
```

Řídicí struktury v procesu – cykly

- Cykly umožňující opakované vykonání sekvence příkazů

- *while ... do ...*

```
WHILE <condition> LOOP <statements> END LOOP;
```

- *for ... loop ...*

```
FOR <range> LOOP <statements> END LOOP;
```

- Příkazy pro přerušení běhu smyčky

- **NEXT** – skok do další iterace

```
next when <condition>;
```

- **EXIT** – ukončení celé smyčky

```
exit when <condition>;
```

Příklad procesu pro součet jedniček

*Součet jedniček v bitovém vektoru **bus_in**:*

```
process (bus_in)
    variable count : std_logic_vector(3 downto 0);
begin
    count := "0000";
    for i in 0 to 15 loop
        if bus_in(i) = '1' then
            count := count + '1';
        end if;
    end loop;
    N_ONE <= count;
end process;
```

- Při každé změně signálu bus_in se vyvolá proces, spočítá se počet jedniček ve vektoru bus_in a výsledek se uloží do N_ONES.
- Pro akumulaci počtu jedniček je využita proměnná.

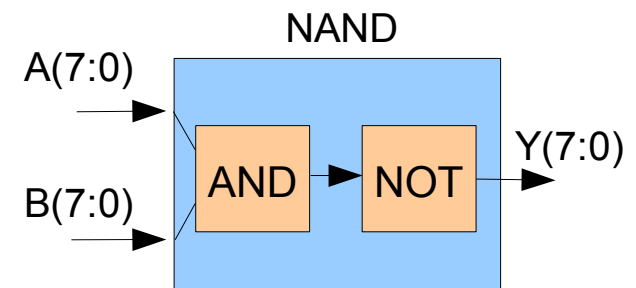
Strukturní popis

- Popisuje z čeho se daný systém (zařízení nebo komponenta) skládá – **jakou má obvod strukturu**
- Strukturní popis může mít více úrovní hierarchie
 - Každá dílčí komponenta může být popsána opět na úrovni struktury nebo na úrovni chování
 - Komponenty na nejnižší úrovni jsou vždy popsány behaviorálně

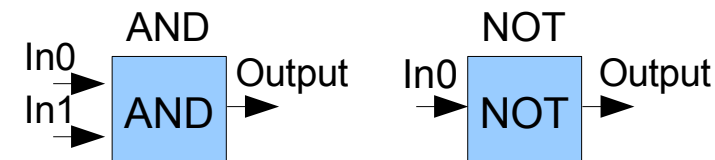
Příklad:

```
architecture struct of NAND is
    signal ab_and : stdl_logic_vector(7 downto 0);
begin
    and_i: entity work.AND
        port map ( In0=>A, In1=>B, Output=>ab_and);
    not_i: entity work.NOT
        port map ( In0=>ab_and, Output=>Y);
end behv;
```

**Komponenty komunikují prostřednictvím
signálu **ab_and****

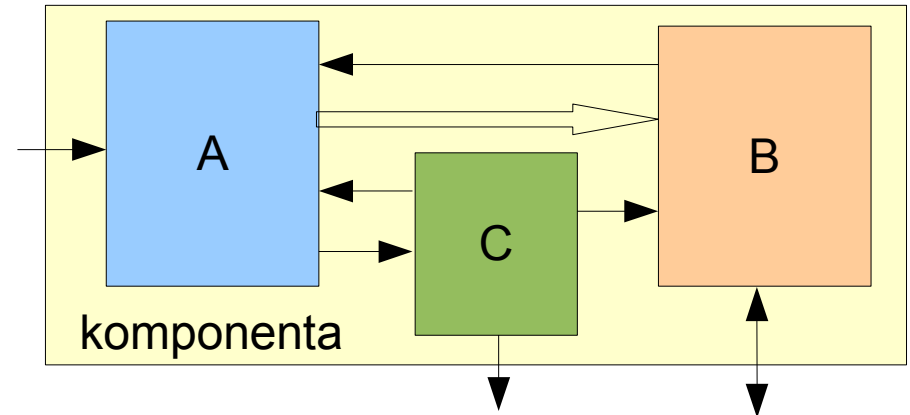


Použité komponenty



Signály ve VHDL

- Signály slouží pro komunikaci mezi komponenty nebo procesy
- Mohou být implementovány pomocí **vodiče** nebo **sběrnice** (více vodičů)
- Signálu je možné přiřadit **libovolný datový typ**, který definuje charakter přenášených hodnot
 - Pro reprezentace vodiče se používá typ `std_logic`
 - Pro sběrnice se používá typ `std_logic_vector()`. Šířka sběrnice je definována šířkou pole.
- Význam bitů sběrnice je dán rozsahem pole



```
std_logic_vector(7 downto 0)
```

```
std_logic_vecotr(0 to 7)
```

7 is MSB bit

0 is MSB bit

Deklarace signálů

- Vnitřní signály komponenty jsou deklarovány v deklarční části architektury – lze použít pouze v rámci architektury

```
Architecture name of processor is
```

```
    Deklarační část
```

```
begin
```

```
    Sekce paralelních příkazů
```

```
end architecture name;
```

Zde je možné deklarovat
vnitřní signály
architektury

Signál nelze deklarovat uvnitř
procesu!!!

- Syntax deklarace:

```
signal <jmeno> : <typ> [:= imp_hodnota];
```

- Jméno jednoznačně identifikuje signál, typ definuje charakter přenášených hodnot.
- Signálu je možné nastavit počáteční (implicitní) hodnotu

Koncept nastavení signálu

- Nastavení signálu se provádí až v okamžiku pozastavení procesu
- V procesu je přiřazena hodnota do signálu vícekrát => provedeno je až poslední nastavení

```
example : process (A, B)
begin
  C <= A;
  C <= B+1;
end process example;
```

První přiřazení signálu je ignorováno

- U každého signálu je v procesu možné nastavit sekvenci jedné nebo více transakcí
- Každá transakce se skládá z nové hodnoty signálu a času přiřazení

```
some_signal <= '0' after 10 ns,
               '1' after 30 ns,
               '0' after 40 ns,
               '1' after 80 ns;
```

Naplánování čtyř transakcí

Atributy signálů

- Simulátor ukládá historii signálu – transakce spojené se signálem
- Historie signálů je ve VHDL přístupná pomocí **atributů** – informací připojených k signálu

- Syntaxe použití atributu:

```
<signal_name>'<attribute name>
```

- Příklady používaných atributů:

- **transaction** – boolean – atribut je hodnoty true v případě, že je právě aktivována na signálu transakce
- **event** – *boolean – atribut je hodnoty true v případě změny hodnoty signálu*
- **last_value** – hodnota signálu před poslední změnou hodnoty

- Příklad detekce náběžné hrany hodin

```
clk'event AND clk='1'
```

Signály v procesu

- Signály slouží pro komunikaci mezi procesy – připojení **vstupů a výstupů procesů**
- Vlastnosti signálů
 - Signály **nemůžou být deklarovány uvnitř procesů**
 - **Přiřazení signálů je fyzicky provedeno až v okamžiku pozastavení procesu.** Dokud není proces pozastaven, má signál původní hodnotu
 - **V procesu je provedeno pouze poslední přiřazení signálů.** Ostatní přiřazení jsou ignorovány
 - Je možné **definovat hodnotu zpoždění pro přiřazení signálu**
- V procesu je možné přechodně uchovat hodnotu pomocí proměnných – ***variable***

Proměnné

- Proměnné mohou být deklarovány a použity pouze v procesu
- Příklad deklarace a použití proměnné

```
Process (b)
variable a : integer := 3;
begin
    a := a + 2;
    output <= a * b;
end process;
```

Deklarace může obsahovat **jméno**, **typ** a **implicitní hodnotu**

Signál output je nastaven již podle aktualizované proměnné **a**
(output <= 5 * 4)

Přiřazení do proměnné se provede okamžitě
(a = 3 + 2 = 5)

- Hodnota proměnné zůstává zachována i v případě pozastavení a opětovném spuštění procesu

Signál vs. Proměnné v procesu

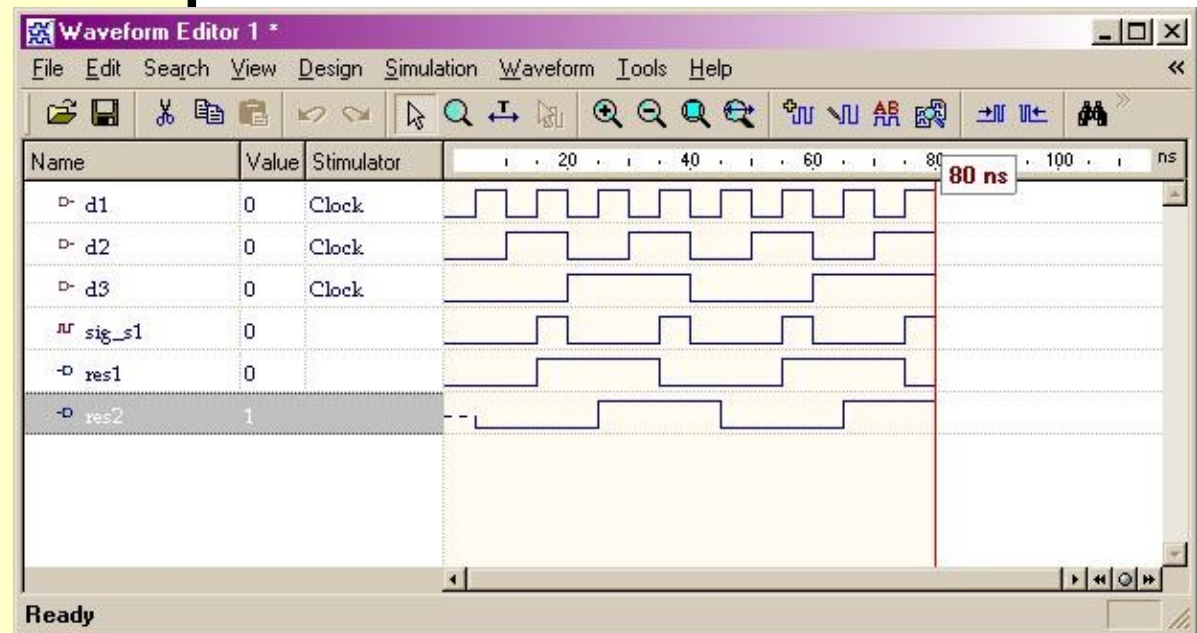
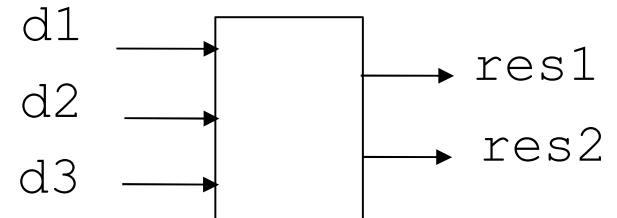
```
entity sig_var is
port(d1, d2, d3: in std_logic;
     res1, res2: out std_logic);
end sig_var;
```

```
architecture behv of sig_var is
```

```
signal sig_s1: std_logic;
begin
```

```
  proc1: process(d1, d2, d3)
    variable var_s1: std_logic;
  begin
    var_s1 := d1 and d2;
    res1 <= var_s1 xor d3;
  end process;
```

```
  proc2: process(d1, d2, d3)
  begin
    sig_s1 <= d1 and d2;
    res2 <= sig_s1 xor d3;
  end process;
end behv;
```



Sledujte rozdíl mezi res1 a res2.

Komentáře, znaky, řetězce, ...

- Komentář – uvozen dvojicí znaků --

```
-- Toto je komentář
```

- Znak nebo bit – vkládá se do apostrofů '1'

```
sig_bit <= '1';
```

- Řetězec nebo bitový vektor – vkládá se do uvozovek

```
sig_bit_vector <= "0001";
```

- Identifikátory – podobná pravidla jako u jiných jazyků

Příklady rozšiřujících datových typů

- Výčtový typ

```
TYPE muj_stav IS (reset, idle, rw, io);  
signal stav: muj_stav;  
stav <= reset;    -- nelze stav <="00";
```

- Pole

```
TYPE data_bus IS ARRAY (0 TO 1) OF BIT;  
variable x: data_bus;  
variable y: bit;  
y := x(12);
```

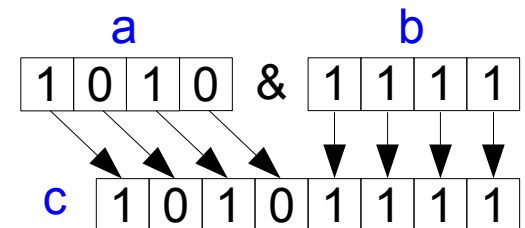
Práce na úrovni bitů

Uvažujme deklarace signálů:

```
signal a, b: bit_vector (3 downto 0);  
signal c    : bit_vector (7 downto 0);  
signal clk : bit;
```

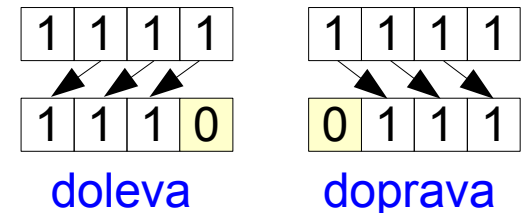
Konkatenace signálů:

```
c <= a & b;
```



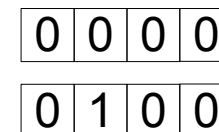
Posun doleva a doprava:

```
b <= b(2 downto 0) & '0'; -- posun doleva  
b <= '0' & b(3 downto 1); -- posun doprava
```



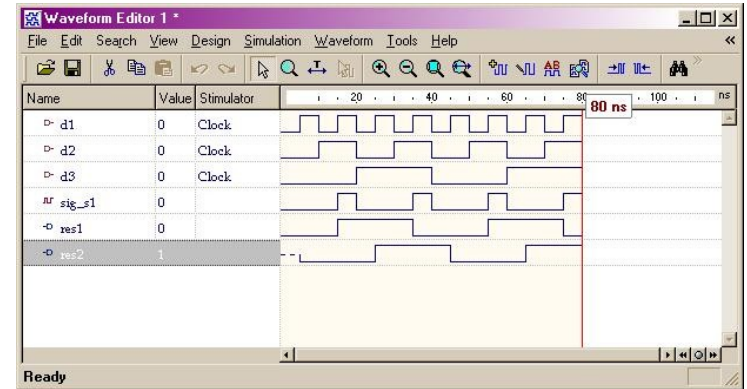
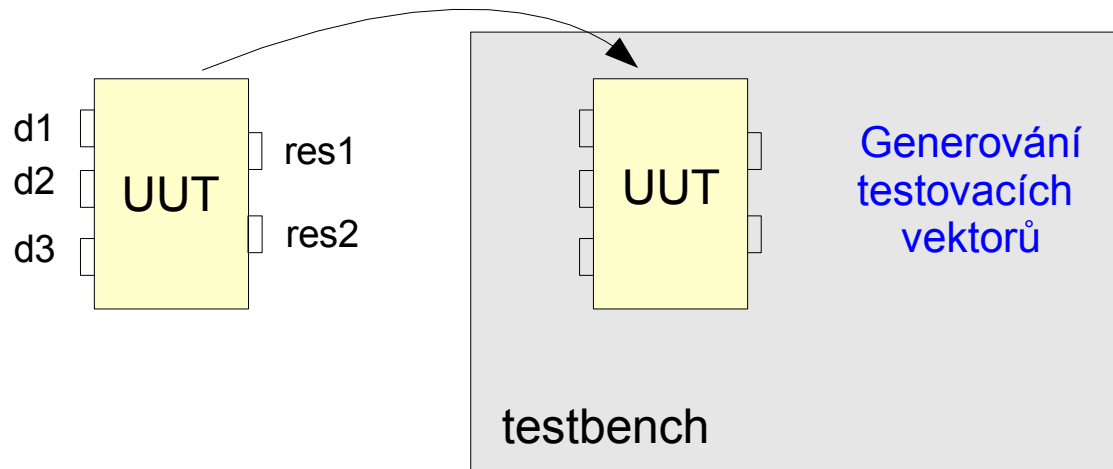
Agregace:

```
a <= (others => '0'); -- vše do nuly  
a <= ('0', '1', others=>'0'); -- MSB = "01"
```



Testbench

- Testování VHDL komponent v prostředí VHDL



- Testbench obvykle obsahuje:
 - Instanci vyvíjené komponenty označenou jako UUT (Unit Under Test)
 - Generátor testovacích vektorů
 - Monitorování a ověřování reakcí UUT

Základní prvky testbench

```
entity testbench is  
end entity testbench;
```

Testbench Entity

```
architecture testbench_arch of testbench is
```

```
    signal a, b : std_logic;  
    signal ...
```

*Testovací vektory
připojené k UUT*

```
begin
```

```
    UUT : entity work.and_gate  
    port map( ...  
              ...  
            );
```

*Instance UUT
(napojení testovacích
vektorů)*

```
    test : process  
    begin  
        a <= ...;  
        b <= ...;  
        ...  
        wait for ...;  
        ...  
        wait for ...;  
        ...  
  
        wait;  
    end process test;
```

*Přikládání
testovacích vektorů*

```
end architecture testbench_arch;
```

Příklad Testbench

```
entity and_gate is
  port (
    A : in  std_logic;
    B : in  std_logic;
    Y : out std_logic
  );
end entity and_gate;

architecture behavioral of and_gate is
begin
  p_and : process (A, B)
  begin
    Y <= A and B;
  end process p_and;
end architecture behavioral;
```

Test všech kombinací na vstupu
hradla AND



```
entity testbench is
end entity testbench;

architecture tb_arch of testbench is
  signal a, b, y : std_logic;
begin

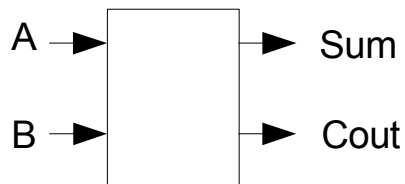
  UUT : entity work.and_gate
  port map( A => a,
            B => b,
            Y => y
  );

  test : process
  begin
    a <= 0; b <= 0;
    wait for 10 ns;
    a <= 0; b <= 1;
    wait for 10 ns;
    a <= 1; b <= 0;
    wait for 10 ns;
    a <= 1; b <= 1;
    wait;
  end process test;

end architecture tb_arch;
```

Příklad – Sčítačka

ADDER



```
entity ADDER is
    port( A, B      : in  std_logic;
          Sum, Cout: out std_logic);
end entity ADDER;
```

Pravdivostní tabulka:

A	B	S	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

	A	
B	+	
	0	1
	1	1
		10

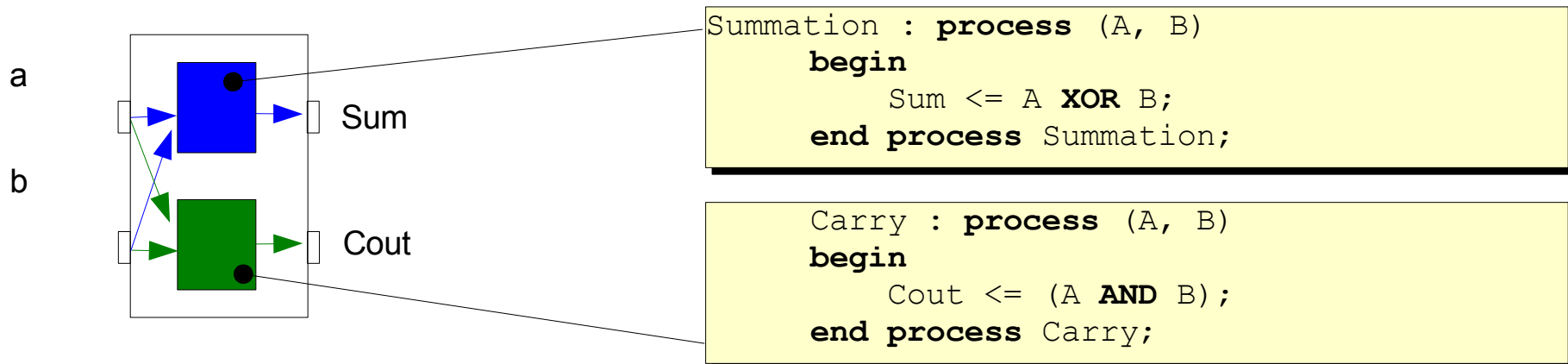
Cout
Přetečení
do vyššího řádu

S
Součet



$$Cout = A \cdot B$$

Příklad – sčítačka



- Popis s využitím procesů

```
ARCHITECTURE behav of ADDER IS
BEGIN
    Summation : process (A, B)
    begin
        Sum <= A XOR B;
    end process Summation;

    Carry : process (A, B)
    begin
        Cout <= (A AND B);
    end process Carry;
END behav;
```

- Dataflow popis

```
ARCHITECTURE dataflow of ADDER IS
BEGIN
    Sum <= A XOR B XOR Cin
    Cout <= (A AND B) OR (A AND Cin) OR
            (B AND Cin);
END dataflow;
```


Dekodér

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

entity dec3to8 is
    port (
        addr: in  std_logic_vector(2 downto 0);
        y    : out std_logic_vector(7 downto 0)
    );
end dec3to8;

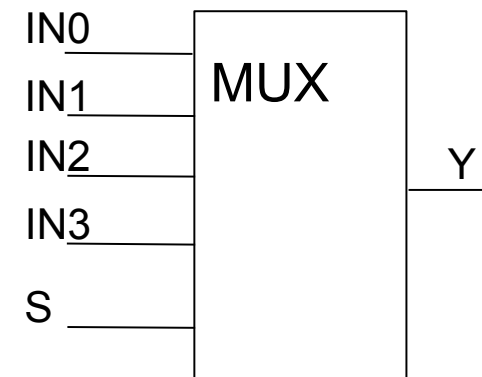
architecture dec3to8arch of dec3to8 is
begin
    with addr select
        y <= "10000000" when "111",
            "01000000" when "110",
            "00100000" when "101",
            "00010000" when "100",
            "00001000" when "011",
            "00000100" when "010",
            "00000010" when "001",
            "00000001" when others;
end dec3to8arch;
```



ADDR (address) – binární adresa
Y – výstupní hodnoty dekodéru. V tomto případě čísla v kódu 1 z n

Multiplexor

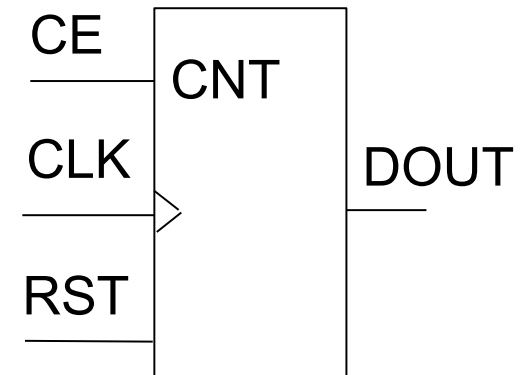
```
library ieee;
use ieee.std_logic_1164.all;
entity Mux is
port( I3: in  std_logic_vector(2 downto 0);
      I2: in  std_logic_vector(2 downto 0);
      I1: in  std_logic_vector(2 downto 0);
      I0: in  std_logic_vector(2 downto 0);
      S : in  std_logic_vector(1 downto 0);
      O : out std_logic_vector(2 downto 0));
end Mux;
architecture behv1 of Mux is
begin
  process (I3, I2, I1, I0, S)
  begin
    case S is
      when "00" => O <= I0;
      when "01" => O <= I1;
      when "10" => O <= I2;
      when "11" => O <= I3;
      when others => O <= "ZZZ";
    end case;
  end process;
end behv1;
```



IN0, IN1,
IN2, IN3 – přepínané vstupy
S – řídicí hradlo
Y – výstup multiplexoru

Čítač

```
library IEEE;
use IEEE.std_logic_1164.all;
entity counter is
port (
    CLK    : in  std_logic;
    RST    : in  std_logic;
    CE     : in  std_logic;
    DOUT   : out std_logic_vector(7 downto 0)
end counter;
architecture behav of counter is
begin
    process (CLK,RST,CE)
    begin
        if (RST = '1') then
            DOUT <= (others => '0');
        elsif (CLK'event and CLK = '1') then
            if CE='1' then
                DOUT <= DOUT + '1';
            end if;
        end if;
    end process;
end behav;
```



CLK (clock) – hodinový vstup

RST(reset) – reset

CE (count enable) – povolení čítání

DOUT (data output) – hodnoty čítače