Datové typy

Datový typ určuje formát dat, který může daný port, signál, proměnná či konstanta přenášet nebo s nimi pracovat. Datové typy mohou být ve VHDL předdefinované (implicitně viditelné ve všech VHDL modelech) nebo se musí deklarovat pomocí příkazu **TYPE**.

Datové typy lze dělit (sdružovat) do skupin, z nichž nejdůležitější jsou:

- *Číselné* – např. integer, real, natural, positive, signed, unsigned;
- *Logické* – bit, boolean, std_logic, std_ulogic;
- *Znakové* – character, string;
- *Složené* – více elementů v jednom objektu (vector, array, record), zápis bitových polí: b”11001001”, X”C3F”, O”754”;
- *Fyzikální* – vyžadují připojení fyzikální jednotky (např. time);

Datové typy - předdefinované

- **charakter**, **string** – pole 256 znaků – ISO 8859-1 (Latin1) (lze užívat i některé speciální znaky, např. ! # $ % & ’ / > á Ö), jsou case sensitive, znaky zapisujeme s apostrofy, stringy s uvozovkami (např.: ”bit” /= ”Bit”);
- **bit** – ‘0‘, ‘1‘;
- **bit_vector** – např. ”1001”;
- **boolean** - false, true;
- **integer** – rozsah $\pm(2^{31}-1)$ + podtypy: **natural** (≥ 0) a **positive** (≥ 1);
- **real** – reálná čísla (nesyntetizovatelné, pouze pro simulace);
- **time** – čas v rozlišení fs, ps, ns, us, ms, s, m, hr (povinná mezera mezi hodnotou a jednotkou).

Deklarace datových typů

SIGNAL a : integer RANGE 0 TO 255;
u celočíselného typu uvádíme obvykle rozsah, který omezuje počet bitů (jinak 32bitové).

SIGNAL b: bit_vector(7 downto 0);

Datové typy, které nejsou implicitní, musí být deklarovány v package nebo v deklarační části architektury příkazem **TYPE,** případně **SUBTYPE**. Např.:

TYPE word IS ARRAY (31 DOWNTO 0) of BIT;

SIGNAL abc : word;

TYPE byte IS INTEGER RANGE 10 DOWNTO -10 ; -- někdy se slovo „INTEGER“ vynechává (v rozsahu jsou uvedena celá čísla)

SIGNAL c1 : byte;

Výčtový datový typ (enumerated)

- specifikuje se seznam hodnot, kterých může nabývat (jiné hodnoty nelze přiřazovat) – diskrétní typ;
- záleží na pořadí (lze určit předchozí a následující hodnotu);
- nejčastěji se v praxi používá k výčtu stavů stavového automatu.

```
TYPE muj_stav IS (res, id, rw, int) ;
SIGNAL stav : muj_stav ;
SIGNAL ekv : STD_LOGIC_VECTOR (0 TO 1) ;
stav <= res ;        -- nelze psát:  stav <= “00“ ;  stav <= ekv ;

TYPE prepinac IS ( ON, OFF );  -- výčtový typ
VARIABLE  a: prepinac;
a := ON;
```

Vícehodnotová logika

Definována v package „std_logic_1164“

std_logic a **std_ulogic** – výčtové typy definující 9 hodnot – všechny hodnoty jsou typu „character“ – ‘U’, ‘X’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘-’ (nutno psát velkými písmeny)

TYPE std_ulogic **IS (**

‘U’, -- neinicializováno (signál nebyl dosud buzen), implicitní
‘X’, -- neznámá hodnota (vzniká při konfliktu ‘0’ a ‘1’)
‘0’, -- log. 0 z tvrdého zdroje
‘1’, -- log. 1 z tvrdého zdroje
‘Z’, -- vysoká impedance
‘W’, -- neznámá hodnota (vzniká při konfliktu H a L)
‘L’, -- log. 0 z měkkého zdroje
‘H’, -- log. 1 z měkkého zdroje
‘-’) ; -- neurčená hodnota (don’t care), na hodnotě nezáleží.

Vícehodnotová logika (pokračování)

std_ulogic – nedisponuje tzv. vyhodnocovací funkcí (resolution function), a tedy na signál tohoto typu nelze připojit více budičů najednou.

std_logic je podtypem std_ulogic s definovanou vyhodnocovací funkcí (tabulkou):

	U	X	0	1	Z	W	L	H	-
U	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
X	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
0	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
1	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
Z	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
W	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
L	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
H	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
-	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

Datové typy signed a unsigned

- datové typy obdobné std_logic_vector (logické vektory), které jsou chápány jako čísla ve dvojkové soustavě;
- unsigned – kladná celá čísla bez znaménka;
- signed – čísla ve dvojkovém kódu se znaménkem;
- slouží pro práci s aritmetickými operátory;
- využívá se při volání package **numeric_std, numeric_bit**;

Příklad:

A má hodnotu “1010”, B má hodnotu “0100”

A a B jsou typu signed, potom A < B je “pravdivé”

A a B jsou typu unsigned, potom A < B je “nepravdivé”

Příkaz WAIT

- příkaz pro spuštění (pozastavení) procesu nebo procedury
- sekvenční příkaz

Syntaxe:

WAIT [**ON** seznam_signálů][**UNTIL** výraz][**FOR** čas] ;

Příklady:

```
WAIT ON s1, s2 ;        -- čekáme, dokud nenastane změna signálů
WAIT FOR 50 ns ;        -- čekáme daný čas (nesyntetizovatelné)
WAIT UNTIL enable = '1' ;       -- čekáme na pravdivost podmínky
WAIT ON a, b UNTIL clk = '1' ;  -- čekáme, dokud nenastane změna
                    jednoho ze signálů a nebo b, ale současně musí clk = 1.
```

Příkaz Process

- představuje nezávislý děj, který se provede při aktivaci;
- užívá se zejména pro popis sekvenčních dějů;
- příkazy uvnitř procesu se vykonávají sekvenčně, ale více procesů v architektuře se vykonává paralelně.

Syntaxe:

[jméno] : **PROCESS** [(clock, reset)] --seznam citlivých proměnných
-- deklarace procesu
BEGIN
-- příkazy procesu
END PROCESS [jméno];

- seznam citlivých proměnných je při syntéze ignorován, ale je významný pro správné provádění simulace.

Spuštění procesu

Každý proces je spuštěn automaticky na začátku simulace.
Další spouštění je možné jedním ze dvou možností:

```
PROCESS (a)     -- spouštění na základě citlivostního seznamu
BEGIN
        y <= a ;
END PROCESS ;

PROCESS         -- spouštění příkazem WAIT
BEGIN
        WAIT on a:
        y <= a ;
END PROCESS ;
```

Process – použití signálů

```
SIGNAL a, b, c : bit ;
PROCESS (b)
BEGIN
a <= b ;
c <= a ;
END PROCESS ;
```

Signál	Předešlý stav	Následující stav
b	1	0
a	1	0
c	1	**1 !**

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

Signál	Předešlý stav	Iterace č. 1	Iterace č. 2
b	1	0	0
a	1	0	0
c	1	1	0

Process – použití proměnné

```
SIGNAL b, c : bit ;
PROCESS (b)
VARIABLE a : bit ;
BEGIN
a := b ;
c <= a ;
END PROCESS ;
```

Signál	Předešlý stav	Následující stav
b	1	0
a	1	0
c	1	**0**

- přiřazování je nahrazeno **:=** **(**např.: y := a AND b ;)
- signály lze přiřazovat do proměnných a naopak
- proměnné deklarované uvnitř procesu si zachovávají svoji hodnotu při dalším průchodu procesem (na rozdíl od funkcí a procedur, kde se vždy inicializují).

Příkaz IF

- lze používat pouze uvnitř procesu (sekvenční příkaz);
- má charakter prioritního přiřazení => může vést na složitější obvodové zapojení (volit raději CASE – WHEN);
- podmínka je výraz vracející hodnotu typu boolean.

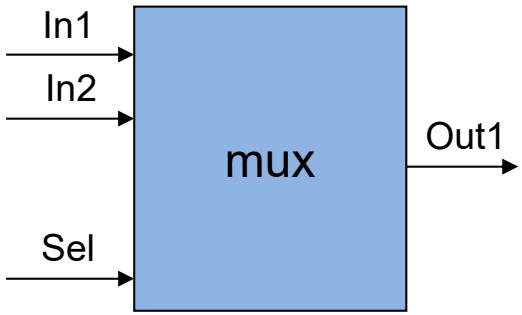
Syntaxe:

IF podmínka1 **THEN** {sekvence_příkazů1}
[{ ELSIF podmínka2 **THEN** {sekvence_příkazů2} }]
[ELSE {sekvence_příkazů}]
END IF ;

Příklad multiplexoru

```
ENTITY mux IS
    PORT ( In1, In2, Sel : IN BIT;
           Out1 : OUT BIT );
END mux ;

ARCHITECTURE multiplexer OF mux IS;
BEGIN
    PROCESS ( Sel, In1, In2 )
    BEGIN
       IF ( Sel = '1' ) THEN  Out1 <= In1;
       ELSE Out1 <= In2;            -- nutné, jinak vznikne latch!
       END IF;
    END PROCESS;
END multiplexer;
```



Příkaz CASE

- lze používat pouze uvnitř procesu (sekvenční příkaz);
- musí být specifikovány všechny možnosti výrazu;
- hodnoty výrazu se nesmí překrývat;
- nemá charakter prioritního přiřazení
  (obdoba příkazu WITH – SELECT – WHEN).

Syntaxe:

```
CASE výraz IS
        WHEN  hodnota_1 => příkaz;
        WHEN  hodnota_2 | hodnota_4 => příkaz;    -- nebo
        WHEN  hodnota_m TO hodnota_n => příkaz;
        WHEN OTHERS  => příkaz;  -- příp. NULL
END CASE;
```

Popis KLO procesem

- V seznamu citlivých proměnných uvést všechny vstupy,
- obvod nutno plně specifikovat.

Příklad KLO:

```
PROCESS (vstupy) BEGIN
  CASE vstupy IS
    WHEN "000" => segment <= "0000001";
    WHEN "001" => segment <= "1001111";
    WHEN "010" => segment <= "0010010";
    WHEN "011" => segment <= "0000110";
    WHEN "100" => segment <= "1001100";
    WHEN "101" => segment <= "0100100";
    WHEN OTHERS => segment <= "-------";  -- hardwarově výhodnější
  END CASE;                               -- než uvést konkrétní logické hodnoty
END PROCESS;
```

Náběžné a sestupné hrany

clk'event AND clk = '1' -- náběžná hrana
clk'event AND clk = '0' -- sestupná hrana

```
SIGNAL clock : boolean ;
IF NOT clock AND clock'event    --  sestupná hrana
```

v knihovně IEEE 1164 existují fce **rising_edge ( )** a **falling_edge ( )**
(knihovnu IEEE 1164 nutno deklarovat – příkazy LIBRARY a USE)

- tyto funkce nelze užívat v logických operacích, např.:
  IF NOT falling_edge (clk) …
- lze psát: enclk <= clk AND clk_en ;
  IF rising_edge(enclk) THEN …

```
WAIT UNTIL (clk'event AND clk = '1');
IF (num'event AND num = '0') THEN q <= c(5);  END IF;
```

Použití hod. signálu v procesu

```
PROCESS (hodinový_signál)     -- použitím citlivých proměnných
BEGIN
IF (podmínka_hrany_hod_signálu) THEN
výstupní_signál <= vstupní signál;
… další sekvenční příkazy …
END IF;
END PROCESS;
```

```
PROCESS                       -- použitím příkazu WAIT
BEGIN
WAIT ON (hodinový_signál) UNTIL (specifikace_hrany_hod_signálu);
výstupní_signál <= vstupní signál;
… další sekvenční příkazy …
END PROCESS;
```

Použití synchronního resetu (1)

```
PROCESS (hodinový_signál)        -- použitím citlivých proměnných
BEGIN
    IF (podmínka_hrany_hod_signálu) THEN
        IF (podmínka_resetu) THEN
        výstupní signál <= resetovací_hodnota;
        ELSE
        výstupní_signál <= vstupní signál;
        … další sekvenční příkazy …
        END IF;
    END IF;
END PROCESS;
```

- nepoužívat ELSE ve vnějším IF s podmínkou hrany hod. signálu;
- nepoužívat v procesu více příkazů IF (pouze vložené).

Použití synchronního resetu (2)

```
PROCESS                                          -- použitím příkazu WAIT
BEGIN
    WAIT ON (hodinový_signál) UNTIL (specifikace_hrany_hod_signálu);
        IF (podmínka_resetu) THEN
        výstupní_signál <= resetovací_hodnota;
        ELSE
        výstupní_signál <= vstupní signál;
        … další sekvenční příkazy …
        END IF;
END PROCESS;
```

Použití asynchronního resetu (1)

Použitím citlivých proměnných:

```
PROCESS (hodinový_signál, resetovací_signál)
BEGIN
        IF (resetovací_podmínka) THEN
        výstupní signál <= resetovací_hodnota;
        ELSIF (podmínka_hrany_hod_signálu) THEN
                výstupní_signál <= vstupní signál;
                … další sekvenční příkazy …
        END IF;
END PROCESS;
```

- nepoužívat příkaz ELSE po detekci hrany hodinového signálu;
- nepoužívat v procesu více příkazů IF (pouze vložené).

Použití asynchronního resetu (2)

Použitím příkazu WAIT:

```
PROCESS
BEGIN
        WAIT ON hodinový_signál, resetovací_signál;
        IF (resetovací_podmínka) THEN
        výstupní signál <= resetovací_hodnota;
        ELSIF (podmínka_hrany_hod_signálu) THEN
                    výstupní_signál <= vstupní signál;
                    … další sekvenční příkazy …
        END IF;
END PROCESS;
```

Použití asynchronního resetu a setu

Příklad klopného obvodu D s asynchronním resetem a setem:

```
PROCESS (clk, reset, set)
BEGIN
    IF (reset = '0') THEN
        q <= '0';
    ELSIF (set = '0') THEN
        q <= '1';
    ELSIF (clk'event AND clk = '1') THEN
        q <= data;
    END IF;
END PROCESS;
```

V citlivých proměnných uvedeny pouze hodiny a asynchronní signály!

Klopný obvod řízený hladinou (latch)

- v návrzích raději nepoužívat (vliv střídy hodinového signálu);
- v ASIC obvodech vedou na jednodušší obvodové zapojení;
- někdy se generují v důsledku špatného popisu kombinační logiky (chybějící ELSE, neúplný příkaz CASE);
- v seznamu citlivých proměnných nutno uvést i datové vstupy.

Příklad: (při clk = ‘0’ zachovává předchozí stav)

```
PROCESS (clk, a, b)
BEGIN
        IF (clk = ‘1’) THEN
            y <= a AND b ;   -- u KLO by bylo: ELSE y <= ‘0’ ;
        END IF ;
END PROCESS ;
```

Hranový klopný obvod

- nevytvářet hranově řízené klopné obvody z latchů tím, že neuvedeme úplný seznam citlivých proměnných;

```
PROCESS (clk)  -- nevhodné řešení (nereaguje na změnu ‘a’ nebo ‘b’
BEGIN          -- a tím se může tvářit jako KO řízený hranou)
    IF (clk = ‘1’) THEN
        y <= a AND b;
    END IF;
END PROCESS;

PROCESS (clk)     -- vhodné řešení
BEGIN
    IF rising_edge(clk) THEN
        y <= a AND b;
    END IF;
END PROCESS;
```

Více než jeden hodinový signál

- každý hodinový signál musí mít vlastní proces

```
PROCESS (clk1, clk2)
BEGIN
    IF riging_edge(clk1) THEN
        q1 <= a;
    END IF ;
    IF rising_edge(clk2) THEN      -- nelze
        q2 <= b;
    END IF;
END PROCESS;
```