Knihovny (Library)

Jazyk VHDL definuje dvě třídy knihoven – pracovní (work) a zdrojové.

Pracovní knihovna může být připojena pouze jedna a jsou do ní automaticky ukládány překládané objekty.

V knihovně se nacházejí tzv. primární a sekundární návrhové jednotky - mezi primární patří **entity**, **package** a **configuration**, mezi sekundární **architecture** a **package body** (configuration nemá sekundární jednotku).

Primární a jim odpovídající sekundární jednotky se musí nacházet ve stejné knihovně.

V knihovně jsou nejčastěji **package** (knihovní balíky, slohy), příp. package body.

Položky LIBRARY a USE

- V systému VHDL je standardně přístupná pouze knihovna „Std“ s package „standard“ - není třeba připojovat (bývá přístupná i knihovna „work“); ostatní knihovny je třeba připojit (zviditelnit) příkazem LIBRARY;
- package se připojují příkazem USE (zviditelňuje specifikované položky v daném package);
- obě položky nutno v návrhu opakovat pro každou entitu a package;
- pokud je třeba zpřístupnit více package z jedné knihovny, užijeme příkaz USE několikrát za sebou.

Syntaxe:

LIBRARY jméno_knihovny ;

USE jméno_knihovny.jméno_package.položka ; -- příp. all

Package (knihovní balík, sloha)

- hierarchicky nad entitou a architekturou;
- položky deklarované v package jsou viditelné v celém návrhu (v package nelze uvést deklaraci entity nebo architektury);
- Package obsahuje dvě části:
  - deklarační část (příkaz **PACKAGE**) – pro deklarace hlaviček funkcí a procedur, typů a podtypů, konstant a signálů, komponent (vlastní popis musí být mimo sekci package), atributů, aj.
  - vlastní tělo (příkaz **PACKAGE BODY**) – pro definice podprogramů, funkcí, aj.).

Package - příklad

```
PACKAGE system IS
    CONSTANT pocet : INTEGER := 2;
    PROCEDURE add (SIGNAL a, b : IN BIT; suma : OUT BIT );
END system;

PACKAGE BODY system IS
    PROCEDURE add (SIGNAL a, b : IN BIT; suma : OUT BIT );
    BEGIN
            suma <= a  XOR  b;
    END add;
END system;
```

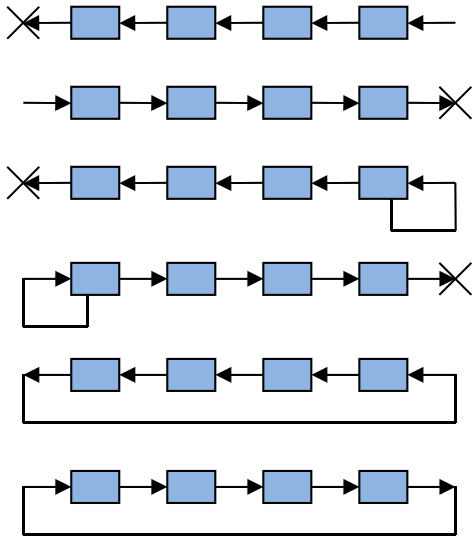
Standardizované package IEEE

Jsou uloženy v knihovně IEEE;

- **std_logic_1164** – definice vícehodnotovou logiku, vybrané konverzní funkce a logické operátory s std logikou;
- **numeric_bit, numeric_std** – definice aritmetických a logických operací nad typy unsigned a signed (nahrazují starší **package std_logic_arith, std_logic_unsigned a std_logic_signed**);
- **math_real** – definuje operace v plovoucí řádové čárce;
- **fixed_pkg, fphdl_pkg** – připravují se syntetizovatelné balíky pro podporu matematických operací v pevné a plovoucí řádové čárce (Quartus nepodporuje).

Operátory

- logické: **NOT, AND, OR, NAND, NOR, XOR, XNOR**
- relační: **=, /=, >, <, >=, <=**
- aritmetické: **+, -, *, /, mod** (Modulus), **rem** (Remainder), **abs** (Absolute Value), ** (exponent)
- posuvu a rotace
  - **SLL** (Shift Left Logical)
  - **SRL** (Shift Right Logical)
  - **SLA** (Shift Left Arithmetic)
  - **SRA** (Shift Right Arithmetic)
  - **ROL** (Rotate Left Logical)
  - **ROR** (Rotate Right Logical)
- slučující operátor &



Operátory - priorita

1) **, ABS, NOT -- nejvyšší priorita
2) *, /, MOD, REM
3) znaménka +, -
4) +, -, &
5) SLL, SRL, SLA, SRA, ROL, ROR
6) =, /=, <, <=, >, >=
7) AND, OR, NAND, NOR, XOR, XNOR -- nejnižší priorita

Prioritu operátorů lze měnit oblými závorkami, jinak se vyhodnocují zleva doprava (pozor na neasociativní operátory NAND a NOR)

Operátory - poznámky

- operandy operátorů musí být stejného typu;
- operandy u logických operátorů musí mít stejnou délku;
- porovnáváme-li u relačních operací operátory různé délky, operátory se porovnávají zleva doprava (zarovnávají se vlevo):

  bit_vector (“1000”) > bit_vector (“111”)

  -- výsledkem je FALSE, protože 1000 < 1110
- logické operátory lze používat jen s datovými typy bit, boolean, std_logic, bit_vector, std_logic_vector;
- relační operátory lze používat se skalárními datovými typy a jednorozměrnými poli – vrací hodnotu true a false (u datových typů record pouze = a /=).

Operace posuvu a rotace

Jsou definovány pro jednorozměrná pole typu bit a boolean, pravý operand musí být typu integer (může být i záporný)

Příklad:

```
signal a : bit_vector (7 downto 0) := “10000001”;
signal b, c, d, e, f, g : bit_vector (7 downto 0);

b <= a sll 1;        -- výsledek “00000010”
c <= a srl 1;        -- výsledek “01000000”
d <= a sla 2;        -- výsledek “00000111”
e <= a sra 2;        -- výsledek “11100000”
f <= a rol 1;        -- výsledek “00000011”
g <= a ror 1;        -- výsledek “11000000”
```

Slučující operátor (concatenation) &

Slučuje jednorozměrná pole (včetně řetězců):

Příklad – použití pro sloučení dvou signálů:

```
a, b : IN bit_vector (1 DOWNTO 0);
c : OUT bit_vector (3 DOWNTO 0);
c <= a & b;
-- c(3) <= a(1);    c(2) <= a(0);    c(1) <= b(1);    c(0) <= b(0);
```

Příklad – použití při definici posuvného registru:

```
VARIABLE shifted, shiftin : bit_vector (0 TO 3);
shifted := shiftin (1 TO 3) & '0';
```

Příklad – použití pro násobení a dělení (Q = C / 16 + C * 4):

```
SIGNAL C, Q : std_logic_vector (7 DOWNTO 0);
Q <= „0000“ & C(7 DOWNTO 4) + C(5 DOWNTO 0) & „00“;
```

Konverzní funkce

Knihovna ieee.std_logic_1164:

TO_BIT(arg)	z **std_logic** na **bit**
TO_BITVECTOR(arg)	z **std_logic_vector** na **bit_vector**
TO_STDLOGICVECTOR(arg)	z **bit_vector** na **std_logic_vector**

Knihovna ieee.std_logic_arith:

CONV_INTEGER(arg)	z **(un)signed** na **integer**
CONV_UNSIGNED(arg, b)	z **integer** a **signed** na **unsigned**
CONV_SIGNED(arg, b)	z **integer** a **unsigned** na **signed**
UNSIGNED(arg)	z **std_logic_vector** na **unsigned**
SIGNED(arg)	z **std_logic_vector** na **signed**
STD_LOGIC_VECTOR(arg)	z **(un)signed** na **std_logic_vector**
CONV_STD_LOGIC_VECTOR(arg, b)	z **integer** a **(un)signed** na **std_logic_vector**

(druhý parametr udává počet bitů výsledku)

Konverzní funkce (pokračování)

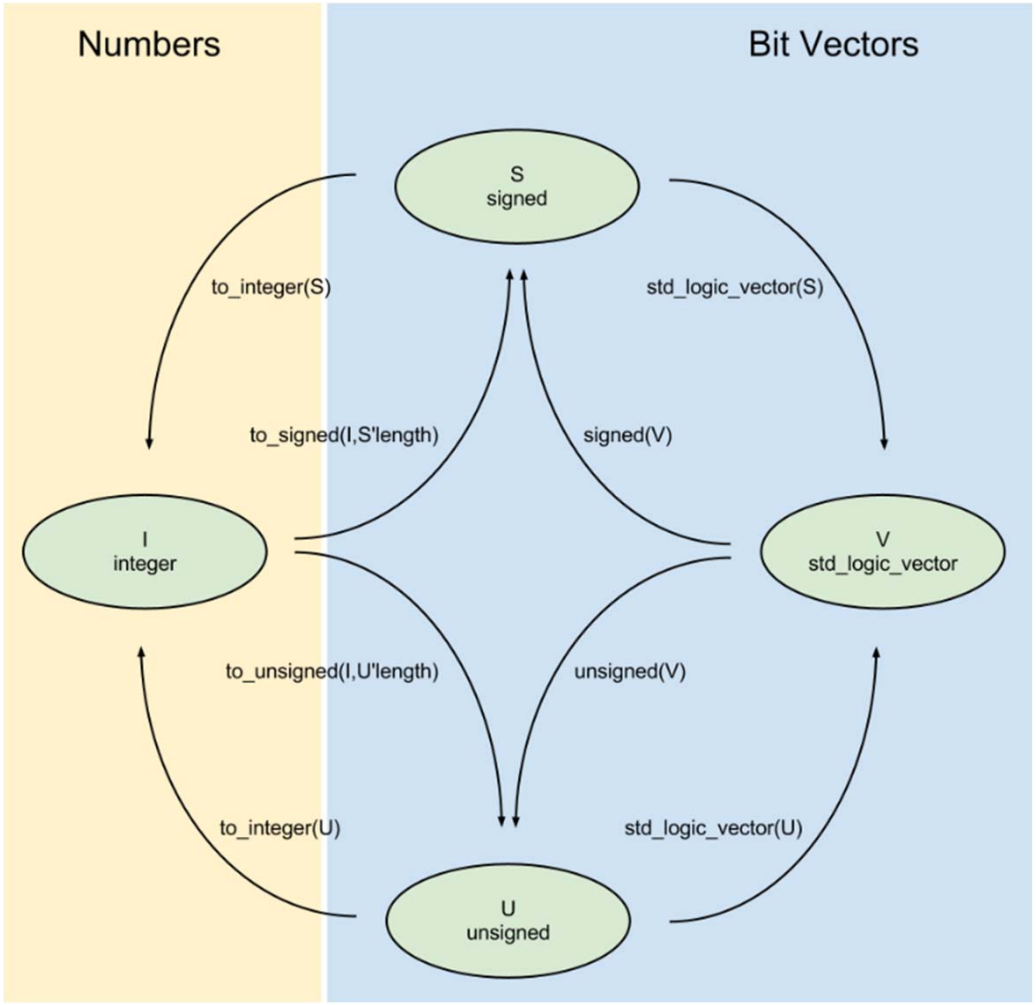
Knihovna ieee.numeric_std (ieee.numeric_bit):
(obsahuje funkce obdobné jako *std_logic_arith*, některé mají jiný název)

TO_INTEGER(arg)	z **(un)signed** na **integer**
TO_SIGNED(arg, b)	z **integer** a **unsigned** na **signed**
TO_UNSIGNED(arg, b)	z **integer** a **signed** na **unsigned**

Příklad:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
…
subtype byte is bit_vector (7 downto 0);
type mem_16x8 is array (0 to 15) of byte;
signal RAM : mem_16x8;
signal data : std_logic_vector (7 downto 0);
signal addr : bit_vector (0 to 3);
…
data <= to_stdlogicvector (RAM (conv_integer (to_stdlogicvector (addr))));
RAM (conv_integer (to_stdlogicvector (addr))) <= to_bitvector (data);
```

Konverzní funkce (pokračování)



Atributy

- Atributy jsou v podstatě speciální funkce, které poskytují dodatečnou informaci jejich nositelích;
- nositeli mohou být typy, signály, proměnné, vektory, pole, bloky, architektury, jednotky, …

Atributy: *předdefinované* (typy, pole, signály, objekty);
definované uživatelem (nepříliš standardizované).

Syntaxe:
PREFIX'jméno_atributu[(výraz)]

Předdef. atributy – typy (podtypy)

Celkem 14 funkcí k získávání informací o typech nebo podtypech, nejčastější jsou:

T'low	-- nejnižší mez typu T
T'high	-- nejvyšší mez typu T
T'left	-- hodnota levé krajní meze typu T
T'right	-- hodnota pravé krajní meze typu T
T'leftof(X)	-- hodnota na pozici vlevo od X
T'rightof(X)	-- hodnota na pozici vpravo od X
T'image(X)	-- převede výraz X na textový řetězec
T'value(X)	-- převede textový řetězec X na hodnotu

Předdef. atributy – typy (příklady)

TYPE word_ab IS range 31 downto 0;

-- word_ab'left = 31 word_ab'right = 0

-- word_ab'high = 31 word_ab'low = 0

-- word_ab'leftof(20) = 21 word_ab'rightof(20) = 19

SUBTYPE shorter IS integer range 0 to 100 ;

-- shorter'high = 100 shorter'left = 0

Předdefinované atributy - pole

Celkem 8 funkcí k získávání informací o polích:

A'low(N) -- nejnižší hodnota indexu N-té dimenze pole A

A'high(N) -- nejvyšší hodnota indexu N-té dimenze pole A

A'left(N) -- hodnota indexu levé krajní meze N-té dimenze pole A

A'right(N) -- hodnota indexu pravé krajní meze N-té dimenze pole A

A'length(N) -- počet prvků N-té dimenze pole A

A'range(N) -- rozsah indexů N-té dimenze pole od A'left do A'right

A'reverse_range(N) -- obrácený rozsah indexů N-té dimenze pole A (zamění *to* za *downto* či naopak)

A'ascending(N) -- booleovská hodnota *true*, pokud má N-tá dimenze pole vzestupný (*to*) rozsah, jinak *false*

(parametr N je možné u jednorozměrného pole vynechat)

Předdef. atributy – pole (příklady)

```
SIGNAL ab : bit_vector (7 downto 0);
--  ab'length = 8          ab'left = 7          ab'low = 0

TYPE pole8x4 IS array (8 downto 1, 0 to 3) of boolean;
--  pole8x4'left(1) = 8                  pole8x4'left(2) = 0
--  pole8x4'high(1) = 8                  pole8x4'right(2) = 3
--  pole8x4'low(1) = 1                   pole8x4'length(2) = 4
--  pole8x4'ascending(1) = false         pole8x4'range(2) = 0 to 3
--  pole8x4'reverse_range(1) = 1 to 8
```

Předdefinované atributy - signály

Celkem 11 funkcí k získání informací o signálech, nejpoužívanější jsou:

S'stable(T) -- pravdivý, pokud za čas T nenastala událost na S
S'event -- pravdivé, pokud událost na S právě nastala
S'last_value -- předcházející hodnota před změnou S
S'last_event -- čas mezi současností a minulou událostí na S
S'quiet(T) -- pravdivý, pokud za čas T nenastala transakce na S
S'active -- pravdivé, pokud transakce na S právě nastala
S'delayed(T) -- vytvoří kopii signálu zpožděnou o čas T (není-li čas udán, je signál zpožděn o tzv. delta zpoždění).

Transakce je přepočet (update) konkrétní hodnoty signálu – nemusí znamenat změnu hodnoty (transakce musí nastat vždy, když nastane událost).
Událost je transakce, jejíž výsledkem je změna hodnoty.
Signál je množina událostí (změna hodnoty v konkrétní čas).

Předdef. atributy – signály (příklady)

```
IF (clk'event and clk='1') THEN out1 <= data;  END IF;
    -- nastala-li na clk vzestupná hrana


WAIT FOR 30 ns;
data <= '1' after 30 ns;
WAIT FOR 10 ns;
a := data'stable(20 ns); -- true (na data 20 ns zpátky nenastala událost)
WAIT FOR 30 ns;
a := data'stable(20 ns);  -- false (před 10 ns nastala na data událost)
```

Atributy definované uživatelem

- slouží např. pro ovládání chodu syntezátoru nebo simulátoru (někde řešeno pomocí komentářů), závislé na konkrétním výrobci

Syntaxe:

ATTRIBUTE attribute_name : string;

ATTRIBUTE attribute_name **OF**
{component_name|label_name
|entity_name|signal_name|variable_name|type_name}:
{**component|label|entity|signal|variable|type**} **IS**
attribute_value;

Atributy def. uživatelem (příklady)

```
ATTRIBUTE loc : string;              -- přiřazení pinů
ATTRIBUTE loc OF out_B: SIGNAL IS “P14  P15  P16”; -- piny 14, 15, 16

ATTRIBUTE io_types : string;       -- druh V/V pinů
ATTRIBUTE io_types OF port_F: SIGNAL IS “LVCMOS33, 20”;  -- 20 mA

ATTRIBUTE pull : string;             -- připojení pull-up a pull-down odporů
ATTRIBUTE pull OF enabl_C: SIGNAL IS “DOWN”;   -- neaktivní v log. 0

ATTRIBUTE enum_encoding : string;
ATTRIBUTE enum_encoding OF barvy : TYPE IS “000 100 010 001 101”;
                    -- definice kódování hodnot výčtových datových typů
```

Příkaz LOOP (typy WHILE a FOR)

- příkaz lze používat pouze uvnitř procesu
- spouští opakovaně sekvence příkazů (smyčka)
- sekvenční příkaz

Syntaxe:

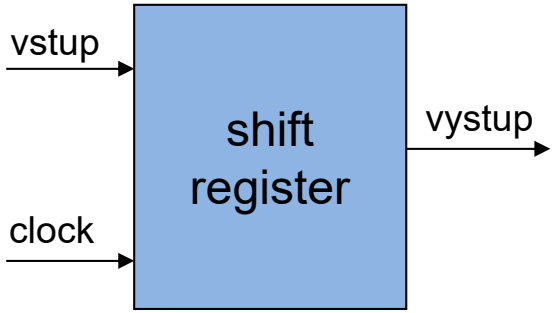
[návěstí] : **WHILE** podmínka **LOOP**
sekvence příkazů
END LOOP [návěstí];

[návěstí] : **FOR** parameter **IN** rozsah **LOOP**
sekvence příkazů
END LOOP [návěstí];

Příklad 4bit. posuvného registru

```
ENTITY shift_register IS
    PORT ( vstup, clock : IN BIT;
           vystup : OUT BIT );
END shift_register;

ARCHITECTURE a_shreg OF shift_register IS
SIGNAL pom : BIT_VECTOR ( 3 DOWNTO  0 ) := "0000";
BEGIN
    PROCESS ( clock ) BEGIN
      IF ( clock'event) AND ( clock = '1' ) THEN
          vystup <= pom (3);
          FOR  i  IN 3  DOWNTO 1 LOOP
              pom (i) <= pom (i-1);
          END LOOP;
          pom (0) <= vstup;
      END IF;
    END PROCESS;
END a_shreg;              -- „i" není nutné deklarovat, inicializovat, inkrementovat
```



Smyčka typu WHILE - příklad

- proměnnou „i“ je nutné deklarovat, inicializovat a inkrementovat

```
PROCESS ( clk, Rst, D_in )
VARIABLE i : integer;
BEGIN
    i := 0;
    IF Rst = ‘1’ THEN
    WHILE i < 7 LOOP
        Reg(i) <= (OTHERS => ‘0’); -- asynchronní nulování
        i := i + 1;
    END LOOP;
    ELSIF ( clock’event) AND ( clock = ‘1’ ) THEN
    …                                   -- popis synchronní funkce
    END IF;
END PROCESS;
```

Příkaz GENERATE

- pro popis pravidelných hardwarových struktur
- souběžný příkaz (nepoužívat v procesu)

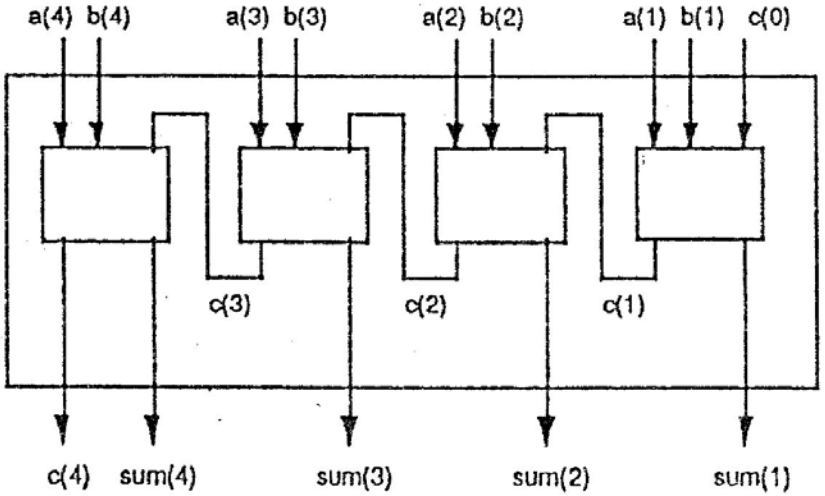
Syntaxe:

label : **FOR** parametr **IN** rozsah **GENERATE**
[{ deklarace }
BEGIN]
{ souběžné příkazy }
END GENERATE [label];

label : **IF** podmínka **GENERATE** -- nelze použít else
[{ deklarace }
BEGIN]
{ souběžné příkazy }
END GENERATE [label];

Příkaz GENERATE - příklad

Label : **FOR** i **IN** 1 to 4 **GENERATE**
jma PORT (a(i), b(i), c(i-1), c(i), sum(i));
END GENERATE;



Ekvivalentní příkazům:

U1: jma **PORT MAP** (a(1), b(1), c(0), c(1), sum(1));
U2: jma **PORT MAP** (a(2), b(2), c(1), c(2), sum(2));
U3: jma **PORT MAP** (a(3), b(3), c(2), c(3), sum(3));
U4: jma **PORT MAP** (a(4), b(4), c(3), c(4), sum(4));