UNIVERSITÀ
DEGLI STUDI DI TRIESTE

dipartimento
di ingegneria
e architettura

10 – Predefined and Standard Packages

A.Carini – Progettazione di sistemi elettronici

IEEE Standard Packages

- The IEEE has published standard packages that define some data types and common operations much used.
- The use of these packages can save user time.
- Moreover, many tool developers provide optimized implementations for the standard packages.
- We will see:
  - `Math_real`
  - `Math_complex`
  - `Std_logic_1164`
  - `Numeric_bit, numeric_std`
  - `Fixed_generic_pkg`
  - `(Float_generic_pkg)`

Math_real

TABLE 9.1 *Constants defined in the package* math_real

Constant	*Value*	*Constant*	*Value*
math_e	e	math_log_of_2	$\ln 2$
math_1_over_e	$1/e$	math_log_of_10	$\ln 10$
math_pi	π	math_log2_of_e	$\log_2 e$
math_2_pi	2π	math_log10_of_e	$\log_{10} e$
math_1_over_pi	$1/\pi$	math_sqrt_2	$\sqrt{2}$
math_pi_over_2	$\pi/2$	math_1_over_sqrt_2	$1/\sqrt{2}$
math_pi_over_3	$\pi/3$	math_sqrt_pi	$\sqrt{\pi}$
math_pi_over_4	$\pi/4$	math_deg_to_rad	$2\pi/360$
math_3_pi_over_2	$3\pi/2$	math_rad_to_deg	$360/2\pi$

Math_real

TABLE 9.2 *Functions defined in the package* math_real

Function	*Meaning*	*Function*	*Meaning*
ceil(x)	Ceiling of x (least integer $\geq x$)	sign(x)	Sign of x (–1.0, 0.0 or +1.0)
floor(x)	Floor of x (greatest integer $\leq x$)	"mod"(x, y)	Floating-point modulus of x /y
round(x)	x rounded to nearest integer value (ties rounded away from 0.0)	realmax(x, y)	Greater of x and y
trunc(x)	x truncated toward 0.0	realmin(x, y)	Lesser of x and y
sqrt(x)	$\sqrt{x}$	log(x)	$\ln x$
cbrt(x)	$\sqrt[3]{x}$	log2(x)	$\log_2 x$
"**"(n, y)	n^y	log10(x)	$\log_{10} x$
"**"(x, y)	x^y	log(x, y)	$\log_y x$
exp(x)	e^x		

Math_real

sin(x)	sin x (x in radians)	arcsin(x)	arcsin x
cos(x)	cos x (x in radians)	arccos(x)	arccos x
tan(x)	tan x (x in radians)	arctan(x)	arctan x
		arctan(y, x)	arctan of point (x, y)

Function	*Meaning*	*Function*	*Meaning*
sinh(x)	sinh x	arcsinh(x)	arcsinh x
cosh(x)	cosh x	arccosh(x)	arccosh x
tanh(x)	tanh x	arctanh(x)	arctanh x

```
procedure uniform ( variable seed1, seed2 : inout positive;
                    variable x : out real);
```

- The procedure generates values in [0, 1]. Seed1 and Seed2 are updated at each procedure call and are integers.
- Seed1 in [1 .. 2,147,483,562], Seed2 in [1 .. 2,147,483,398] (~=2^31)

Math_complex

```
type complex is record
    re : real;     -- Real part
    im : real;     -- Imaginary part
  end record;
```

```
subtype positive_real is real range 0.0 to real'high;
subtype principal_value is real range -math_pi to math_pi;

type complex_polar is record
    mag : positive_real;     -- Magnitude
    arg : principal_value;  -- Angle in radians; -math_pi is illegal
  end record;
```

Constants:

math_cbase_1	$1.0 + j0.0$
math_cbase_j	$0.0 + j1.0$
math_czero	$0.0 + j0.0$

Math_complex

TABLE 9.3 *Overloaded operators defined in* math_complex

Operator	*Operation*		*Left operand*	*Right operand*	*Result*
=	equality		complex_polar	complex_polar	boolean
/=	inequality		complex_polar	complex_polar	boolean
abs	magnitude			complex	positive_real
				complex_polar	positive_real
–	negation			complex	complex
				complex_polar	complex_polar
+	addition	{	complex	complex	complex
–	subtraction		real	complex	complex
*	multiplication		complex	real	complex
/	division		complex_polar	complex_polar	complex_polar
			real	complex_polar	complex_polar
			complex_polar	real	complex_polar

Math_complex

TABLE 9.4 *Functions defined in the package* math_complex

Function	*Result type*	*Meaning*
cmplx(x, y)	complex	$x + jy$
get_principal_value(x)	principal_value	$x + 2\pi k$ for some k, such that $-\pi <$ result $\leq \pi$
complex_to_polar(c)	complex_polar	c in polar form
polar_to_complex(p)	complex	p in Cartesian form
arg(z)	principal_value	arg(z) in radians
conj(z)	same as z	complex conjugate of z
sqrt(z)	same as z	$\sqrt{z}$
exp(z)	same as z	e^z
log(z)	same as z	$\ln z$
log2(z)	same as z	$\log_2 z$
log10(z)	same as z	$\log_{10} z$
log(z, y)	same as z	$\log_y z$
sin(z)	same as z	$\sin z$
cos(z)	same as z	$\cos z$
sinh(z)	same as z	$\sinh z$
cosh(z)	same as z	$\cosh z$

Std_Logic_1164 Multivalue Logic

- Defines types and overloaded operators.
- The types it defines:

std_ulogic	The basic multivalued enumeration type (
std_ulogic_vector	Array of std_ulogic elements (see page 96
std_logic	Resolved multivalued enumeration type (
std_logic_vector	Array of std_logic elements (see Section 1

- Moreover, it defines the subtypes X01, X01Z, UX01, UX01Z.

Std_Logic_1164 Multivalue Logic

```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '–');
```

- Three logic states: ‘0’, ‘1’, **unknown.**
- Three level of driver strength: ‘forced’, ‘weak’, ‘Z’.
    - ‘forced’ level: represents an active driver
      (a transistor in strong conduction or a switch).
    - ‘weak level’: represents a resistive driver
      (a pull-up, a pull-down, or a pass-transistor).
    - ‘z’ or high-impedance-state: represents a switched off driver.
- Moreover, there is an uninitialized state ‘U’ and a ‘don’t care state’, ‘-’.

Std_Logic_1164 Multivalue Logic

- It provides the overloaded version of the logic operators **AND, NAND, OR, NOR, XOR, XNOR, NOT** for scalar and vectors, and other functions, like:

To_bit(s,xmap)	Convert to a bit value
To_bitvector(s, xmap)	Convert to a bit vector

In these two functions, the parameter xmap is a bit value that is used in the result when a bit to be converted is other than '0', '1', 'L' or 'H'.

Std_Logic_1164 Multivalue Logic

To_StdULogic(b)	Convert to a standard logic value
To_StdLogicVector(x)	Convert to a std_logic_vector
To_StdULogicVector(x)	Convert to a std_ulogic_vector
To_X01(x)	Strip strength
To_X01Z(x)	Strip strength
To_UX01(x)	Strip strength

rising_edge(s)	True when there is a rising edge on s, false otherwise
falling_edge(s)	True when there is a falling edge on s, false otherwise
is_X(s)	True if s contains an unknown value, false otherwise

Standard integer numeric packages

- The packages *numeric_bit* and *numeric_std* provide arithmetic operations between integer represented by arrays of *bit* or *std_logic* elements, respectively.
- Each package defines two types: ***signed*** and ***unsigned***.
- In *numeric_bit:*

```
type unsigned is array ( natural range <> ) of bit;
type signed is array ( natural range <> ) of bit;
```

- In *numeric_std*:

```
type UNSIGNED is array ( NATURAL range <> ) of STD_LOGIC;
type SIGNED is array ( NATURAL range <> ) of STD_LOGIC;
```

Standard integer numeric packages

- In VHDL-2008, the definition changed as follows

```
type unresolved_unsigned is array (natural range <>) of std_ulogic;
type unresolved_signed   is array (natural range <>) of std_ulogic;

alias u_unsigned is unresolved_unsigned;
alias u_signed   is unresolved_signed;

subtype unsigned is (resolved) unresolved_unsigned;
subtype signed   is (resolved) unresolved_signed;
```

Standard integer numeric packages

- Whichever the package used, the leftmost element is the most-significant digit, the rightmost element is the least-significant digit.
- Example:

```
signal head_position : signed ( 0 to 15 );
subtype address is unsigned ( 31 downto 0 );
signal next_PC : address;
constant PC_increment : unsigned := X"4";
```

Overloaded operators on unsigned and signed

Operator	*Operation*	*Left operand*	*Right operand*	*Result*
abs	absolute value		signed	signed
–	negation			
+	addition	unsigned	unsigned	unsigned
–	subtraction	signed	signed	signed
*	multiplication	unsigned	natural	unsigned
/	division	natural	unsigned	unsigned
rem	remainder	signed	integer	signed
mod	modulo	integer	signed	signed
=	equality	unsigned	unsigned	boolean
/=	inequality	signed	signed	boolean
<	less than	unsigned	natural	boolean
<=	less than or equal	natural	unsigned	boolean
>	greater than	signed	integer	boolean
>=	greater than or equal	integer	signed	boolean
sll	shift-left logical	unsigned	integer	unsigned
srl	shift-right logical	signed	integer	signed
rol	rotate left			
ror	rotate right			
not	negation		unsigned	unsigned
			signed	signed
and	logical and	unsigned	unsigned	unsigned
or	logical or	signed	signed	signed
nand	negated logical and			
nor	negated logical or			
xor	exclusive or			
xnor	negated exclusive or			

The two operands could also have different length.

Other functions

Function	*First parameter*	*Second parameter*	*Result*
shift_left, shift_right, rotate_left, rotate_right	unsigned	natural	unsigned
	signed	natural	signed
resize	unsigned	natural	unsigned
	signed	natural	signed
to_integer	unsigned		natural
	signed		integer
to_unsigned	natural	natural	unsigned
to_signed	integer	natural	signed
rising_edge*, falling_edge*	**signal** bit		boolean
std_match†	unsigned	unsigned	boolean
	signed	signed	boolean
	std_ulogic	std_ulogic	boolean
	std_ulogic_vector	std_ulogic_vector	boolean
	std_logic_vector	std_logic_vector	boolean
to_01†	unsigned	⟦ std_logic ⟧	unsigned
	signed	⟦ std_logic ⟧	signed

* Provided in numeric_bit only
† Provided in numeric_std only.

Example:

- Addition with carry:

```
signal a, b, sum : unsigned(15 downto 0);
signal c_out : std_ulogic;

(c_out, sum) <= ('0' & a) + ('0' & b);
```

- Addition with carry in:

```
signal a, b : unsigned(15 downto 0);
signal sum  : unsigned(16 downto 0);
signal c_in;
...

sum <= ('0' & a) + ('0' & b) + c_in;
```

Std_logic_unsigned and std_logic_signed

- Many tools provide in IEEE library two “non-standard” packages called *std_logic_unsigned* and *std_logic_signed*.
- The package *std_logic_signed* provides signed numerical computation on type *std_logic_vector.*
- The package *std_logic_unsigned* provides unsigned numerical computation on type *std_logic_vector.*

Standard fixed point packages

- Many signal processing operations require operations on non integer data. These could be floating points operations, but they are costly. More often, fixed point operations are used.
- VHDL 2008 has introduced a certain number of packages for fixed point arithmetic, which is an integer arithmetic with a power of 2 scaling.
- Together with scaling, we must take into account the possibility of rounding, and of saturation in additions.
- The principal package for fixed-point arithmetic is `fixed_generic_pkg`.
- It allows to select the rounding and saturation most suitable for our application.
- It uses some generic constants to control the behavior.

fixed_round_style : fixed_round_style_type

This constant determines the rounding behavior for operations in the package. The type **fixed_round_style_type** is an enumeration type defined in the package **fixed_float_types**, also in library **ieee**. The values are **fixed_round**, if results are to be rounded to the nearest representable value; and **fixed_truncate**, if results are to be truncated toward zero to the next smallest representable value. The default is **fixed_round**.

fixed_overflow_style : fixed_overflow_style_type

This constant determines the behavior on overflow. The type **fixed_overlow_style_type** is defined in the package **fixed_float_types**. The values are **fixed_saturate**, if an overflowing result is to remain at the largest representable value; and **fixed_wrap**, if modulo-based behavior is required. The default is **fixed_saturate**.

fixed_guard_bits : natural

This constant specifies the number of extra bits of precision to use for division operations. The default is 3.

no_warning : boolean

This constant allows suppression of warning messages on conditions such as non-matching operand lengths and occurrence of metalogical values (values other than '0', 'L', '1' and 'H'). The default is **false**.

Standard fixed point packages

- The package defines types for unsigned and signed fixed point numbers.
- For unsigned:

```
type unresolved_ufixed is array (integer range <>) of std_ulogic;

alias u_ufixed is unresolved_ufixed;

subtype ufixed is (resolved) unresolved_ufixed;
```

- Indexes are always descending. The positive indexes are for the whole part, the negatives are for the fractional part.

```
signal A : ufixed(3 downto -3) := "0110100";
```
→ 0110,100 6.5

- But also:

```
variable X : ufixed(9 downto 2);
variable Y : ufixed(-5 downto -14);
```

Standard fixed point packages

- For signed:

```
type unresolved_sfixed is array (integer range <>) of std_ulogic;
alias u_sfixed is unresolved_sfixed;
subtype sfixed is (resolved) unresolved_sfixed;
```

- Same rules for indexes, leftmost bit is the sign bit.
- Operations are performed at maximum precision:

```
signal A4_2 : ufixed(3 downto -2);
signal B3_3 : ufixed(2 downto -3);
signal Y5_3 : ufixed(4 downto -3);
...
Y5_3 <= A4_2 + B3_3;
```

- The integer part is one bit larger that the largest integer part of operands, the fractional part is as large and the largest fractional part.

Standard fixed point packages

- Signal and variables of these types can be assigned using strings or the conversion functions **to_ufixed**, **to_sfixed**

```
signal A4 : ufixed(3 downto -3);
...
A4 <= "0110100";  -- string literal for 6.5
A4 <= to_ufixed(6.5, 3, -3);  -- pass indices
A4 <= to_ufixed(6.5, A4);  -- sized by A4
```

Standard fixed point packages

- The use of string literals in expressions is problematic, see workaround:

```
subtype ufixed4_3 is ufixed(3 downto -3);
signal A4, B4 : ufixed4_3;
signal Y5     : ufixed (4 downto -3);
...
Y5 <= A4 + "0110100";                    -- illegal
Y5 <= A4 + ufixed4_3'("0110100");
Y5 <= A4 + 6.5;                          -- overloading with real
Y5 <= A4 + 6;                            -- overloading with integer
```

Standard fixed point packages

- If we need to change size:

```
signal A4_3 : ufixed(3 downto -3);
signal Y7_3 : ufixed(6 downto -3);
...
Y7_3 <= Y7_3 + A4_3;    -- illegal, result too big

Y7_3 <= resize(arg           => Y7_3 + A4_3,
               size_res      => Y7_3,
               overflow_style => fixed_wrap,
               round_style   => fixed_truncate);


Y7_3 <= resize (arg          => Y7_3 + A4_3,
                left_index   => 7,
                right_index => -3);
```

Standard fixed point packages

- Note that some unexpected results we can be found in multiple operations:

```
signal A4, B4, C4, D4 : ufixed(3 downto 0);
signal Y6             : ufixed(5 downto 0);
signal Y7A, Y7B       : ufixed(6 downto 0);
...

Y6 <= (A4 + B4) + (C4 + D4);

Y7A <= ((A4 + B4) + C4) + D4;

Y7B <= A4 + B4 + C4 + D4;
```

See:

- Peter Ashenden, «The designers' guide to VHDL» Morgan Kaufmann,
  - Chapter 9: 9.1-9.2.4