VHDL Packages: *numeric_std, std_logic_arith*

- The *numeric_std*, *std_logic_arith* packages
  - defines the *unsigned* and *signed* types based on the std_logic type
  - Defines numeric operations such as +, -, *, /, abs, > , < , etc. for these types
- std_logic_arith was developed before numeric_std, the numeric_std package seems to now be preferred.
- Use the numeric_std package when need to perform arithmetic operations (or synthesize arithmetic operators) on std_logic types

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
```

Either this
or this

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
```

Unsigned vs. Signed

- *Unsigned* is an unsigned binary integer with the the MSB as the left-most bit.
- *signed* is defined as a 2's complement value with the most significant bit as the left-most bit.
  - This means the MSB of a:unsigned(7 downto 0) is a(7)
  - The means the MSB of a:unsigned(0 to 7) is a(0)
- Need signed,unsigned types because arithmetic results of operations can be different depending on the types.

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

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

Type Conversions (*std_logic_arith*)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.std_logic_arith.all;

ENTITY tb IS
end tb;

architecture A of tb is
  signal a,b,c: std_logic_vector(7 downto 0);
  signal a_sign, b_sign,c_sign: signed(7 downto 0);
  signal a_unsign, b_unsign,c_unsign: unsigned(7 downto 0);
begin
  -- does not compile, "+" operation unknown std_logic_types
  --c <= a + b;

  c_sign <= a_sign + b_sign;
  c_unsign <= a_unsign + b_unsign;

  a_sign <= signed(a);    -- std_logic_vector to signed;
  a_unsign <= unsigned(a);  -- std_logic_vector to unsigned;
```

```
-- signed to std_logic_vector
a <= conv_std_logic_vector(a_sign,a'length);
 -- unsigned to std_logic_vector
a <= conv_std_logic_vector(a_unsign,a'length);
```

differs from *numeric_std*

Type Conversions (*numeric_std*)

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
use ieee.std_numeric_std.all;

ENTITY tb IS
end tb;

architecture A of tb is
  signal a,b,c: std_logic_vector(7 downto 0);
  signal a_sign, b_sign,c_sign: signed(7 downto 0);
  signal a_unsign, b_unsign,c_unsign: unsigned(7 downto 0);
begin
  -- does not compile, "+" operation unknown std_logic_types
  --c <= a + b;

  c_sign <= a_sign + b_sign;
  c_unsign <= a_unsign + b_unsign;

  a_sign <= signed(a);    -- std_logic_vector to signed;
  a_unsign <= unsigned(a);  -- std_logic_vector to unsigned;
```

```
-- signed to std_logic_vector
a <= std_logic_vector(a_sign);
 -- unsigned to std_logic_vector
a <= std_logic_vector(a_unsign);
```

differs from

std_logic_arith

Supported Operations (numeric_std)

- Abs, unary –
- +, -, *, / (division), rem, mod
- >, <, <=, >=, =, /=
- Shift_left, shift_right, rotate_left, rotate_right
- Sll, srl, rol, ror
- Resize
- To_integer, to_unsigned, to_signed
- Not, and,or,nand,nor,xor,xnor
- Std_match
- To_01

Metalogical and ‘Z’ Values (numeric_std)

- A *metalogical* value is defined as ‘X’, ‘W’, ‘U’, or ‘-’
- A high impedance value is ‘Z’
- If any bit in an operand to a numeric_std function contains a metalogical or high impedance value (‘Z’), the result is returned with all bits set to ‘X’.
  - One exception, the ‘std_match’ function
- A value is *well-defined* if it contains no metalogical or high impedance values.

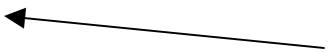
Conversions

Std_ulogic_vector, std_logic_vector, unsigned, signed are all closely related types (subtypes of std_ulogic).

Use explicit *type casts* when assigning one type to another

```
signal a_us,b_us: unsigned(7 downto 0);
signal a_s,b_s: signed(7 downto 0);
signal a,b: std_logic_vector( 7 downto 0);

a <= std_logic_vector(a_s);
a_s <= signed(a_us);
a_us <= unsigned(a);
b_s <= signed(a);
```



Type cast specifies name of target type.

Integer Conversions

```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL;
function TO_INTEGER (ARG: SIGNED) return INTEGER;
function TO_UNSIGNED (ARG, SIZE: NATURAL) return UNSIGNED;
function TO_SIGNED (ARG: INTEGER; SIZE: NATURAL) return SIGNED;
```

Basically the same functions as in the std_logic_1164 package.

Different forms of '+'

```
function "+" (L, R: UNSIGNED) return UNSIGNED;
function "+" (L, R: SIGNED) return SIGNED;
function "+" (L: UNSIGNED; R: NATURAL) return UNSIGNED;
function "+" (L: NATURAL; R: UNSIGNED) return UNSIGNED;
function "+" (L: INTEGER; R: SIGNED) return SIGNED;
function "+" (L: SIGNED; R: INTEGER) return SIGNED;
```

Note different combinations of allowable operands.

For synthesis, there is a problem – do not have access to carry-in, or carry-out which would be very useful. Would have to use operands with 2-extra bits to get access to both carry-in and carry-out.

Mixed Signed/Unsigned Operands

- Note that the defined forms of '+' do not have mixed unsigned/signed operands
- Must do explicit conversions if want to do mix unsigned/signed operands
- This way the user decides how the sign bit should be handled.

Multiplication, Division

function "*" (L, R: UNSIGNED) return UNSIGNED;

-- Result subtype: UNSIGNED((L'LENGTH+R'LENGTH-1) downto 0)

function "/" (L, R: UNSIGNED) return UNSIGNED;

-- Result subtype: UNSIGNED(L'LENGTH-1 downto 0)

Note that N*N returns 2*N bit result

Resize

function RESIZE (ARG: SIGNED; NEW_SIZE: NATURAL)
return SIGNED;

-- Result subtype: SIGNED(NEW_SIZE-1 downto 0)

function RESIZE (ARG: UNSIGNED; NEW_SIZE: NATURAL)
return UNSIGNED;

-- Result subtype: UNSIGNED(NEW_SIZE-1 downto 0)

Change size of input vector. If new size is larger than old, then sign extend the operand for signed numbers, else fill with zeros.

If size decreases, for signed case keep sign bit but drop leftmost part. For unsigned case, just drop leftmost part.

Std_match versus '='

- The *std_match* function compares two operands, with relaxed matching over '='
- Compares each bit. Matching function for each bit returns boolean TRUE if
  - Both values are well-defined (not 'X' or 'U') and the values are the same, or
  - One value is '0' and the other is 'L', or
  - One value is '1' and the other is 'H', or
  - At least one of the values is the don't care value ('-').
- The '=' function first converts both operands to integers, then compares.
  - Uses 'To_01' to map to '0','1' values, then integer convert, then compare.

Real Number/Complex Number computation

- IEEE standard 1076.2-1996 defines MATH_REAL and MATH_COMPLEX
  - Useful for models that do significant floating point computation
  - Not intended for synthesis purposes
  - MATH_REAL packages defines mathematical constants (PI, e, sqrt(2), etc.) and functions (trig functions, square root, X^Y, etc) for REAL number computations.
  - MATH_COMPLEX defines a complex number type and functions dealing with complex numbers
- Before this standard, homebrew packages were used.

VHDL Synthesizeable Subset

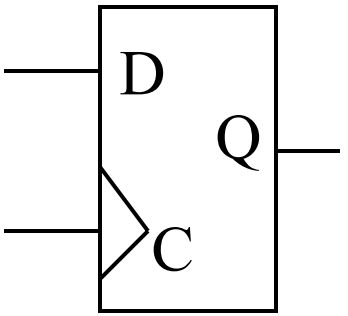
- The subset of VHDL that is synthesizable is limited
  - Unfortunately, not all synthesis tools agree on what statements are ‘synthesizeable’
- IEEE std 1076.6-1999 defines the RTL synthesizeable subset of the language and how language statements should be interpreted for synthesis purposes.
  - This is important as it enhances the portability of RTL models across synthesis tools from different vendors.
  - Much intellectual property of standard bus interfaces and building blocks (I.e, PCI, Firewire, USB, CPUs) are distributed in RTL form so want the RTL to be as portable as possible.

An Example of Coding for Simulation vs. Synthesis

```
A:  process (clk)
   if (clk = '1') then
        q < = d;

   end if;
```

For simulation, do not need *clk'event* because we know that a clk'event triggers the process.




```
B:  process (clk)
   if (clk'event and clk = '1') then
        q < = d;

   end if;
```

Processes A and B both simulate exactly the same.

The synthesis tool needs *clk'event* to infer that an edge-triggered device should be synthesized, not a latch!!

pragmas

- A pragma is an element of a language that is intended be interpreted as a command that controls the action of the compiler or synthesis tools
- The RTL synthesis standard defines two types of pragmas
  - Attributes
  - Metacomments
- A metacomment is a pragma embedded in a comment statement. Two meta comments are defined

```
-- RTL_SYNTHESIS OFF
-- RTL_SYNTHESIS ON
```

This causes statements bracketed by these metacomments to be ignored by the synthesis tool.

pragmas (cont.)

- Only one attribute is defined as having a synthesis specific interpretation
- The ENUM_ENCODING attribute is used specify the encoding for a enumerated type
  - One use is to specify encoding for FSM states

```
type mystate is (ST0,ST1,ST2,ST3);
 attribute ENUM_ENCODING of mystate:
    type is "0001 0010 0100 1000";
```

Defines encoding for ST0 as "001", ST1 as "0010", etc.

RTL Synthesizeable standard

- The standard is basically the LRM with statements marked out as being not recognized by a synthesis tool
- Most things are obvious
  - ie., file operations, dynamic memory allocation are not synthesizable
  - Floating point types are not synthesizeable
  - Delays on signals are ignored
- Arrays and Record types are fine as long as each field specifies a synthesizeable type
- Look at the standard for more detail