TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair of VLSI Design, Diagnostics & Architecture

VHDL – TYPES AND SIGNALS

Dr. Thomas B. Preußer

thomas.preusser@tu-dresden.de

Dresden, July 1, 2014

Goals of this Lecture

- Introduction of VHDL Data Types and Operators.
- Understanding Signals.

General Aspects

VHDL has:

- a strict type system built on very few fundamental constructs,
- subtypes as *more constrained* specialization of their base type,
  This is different from subclasses in OOP!

Primitive Data Types

Discrete Enumerations

```
type boolean   is (false, true);
type bit       is ('0', '1');
type character is (NUL, ..., '0', '1', ..., 'A', 'B', ...);
```

Numerical Ranges (integral and floating-point)

```
type integer is range –2147483648 to 2147483647; -- or larger
type real    is range       –1E38 to       +1E38; -- or larger
```

Subtypes as Constrained Ranges

```
subtype natural  is integer range 0 to integer'high;
subtype positive is integer range 1 to integer'high;
```

Compound Data Types: Arrays

Unconstrained Array Types

```
type bit_vector is array(natural  range <>) of bit;
type string     is array(positive range <>) of character;
```

Constrained Array Types (custom example)

```
-- Immediately constrained Array: byte *is not* a bit_vector
type    byte is array(7 downto 0) of bit;

-- Constrained Subtype: byte *is* a bit_vector
subtype byte is bit_vector(7 downto 0);
```

Multidimensional Arrays (custom example)

```
type matrix is
array(natural range <>, natural range <>) of integer;
```

- are different from arrays of arrays!

Compound Data Types: Structures

Records (custom example)

```
type packet is record
                valid : bit;
                data  : byte;
              end record;
```

Operators

In increasing priority:

Logical Binary	**and or xor nand nor xnor**	– use parentheses
Comparison	= /= < <= > >=	– note the *not equals*
Shift / Rotate	**sll srl sla sra rol ror**	– avoid in favor of . . .
Addition / Concatenation	+ − &	. . . concatenation
Sign	+ −	
Multiplication	∗ / **mod rem**	– beware of . . .
Exponentiation	∗∗	. . . hardware complexity
Numeric Unaries	**not abs**	

Binary operators with equal priority are left-associative.

9-valued Logic (IEEE-1164)

use IEEE.std_logic_1164.**all**;
type std_ulogic **is** (...);

'U'	Uninitialized – default value
'X'	Forcing Unknown (often indicates an error in simulations!)
'0'	Forcing 0
'1'	Forcing 1
'Z'	High Impedance (e.g. tristate buffer, open-collector/open-drain)
'W'	Weak Unknown
'L'	Weak 0 (e.g. pull-down resistor)
'H'	Weak 1 (e.g. pull-up resistor)
'–'	Don't Care

- Intuitive meta-values for debugging purposes.
- Description of optimization opportunities: **else** '–'.
- Description of transmission gates, e.g. in bus drivers: ('Z').
- Modeling of external pull-up or pull-down resistors: ('H', 'L').

Hints

Meta-values have a great relevance for the circuit analysis by simulation:

- 'U' indicates signals that have not been evaluated yet and their dependents.
- '-' describes and *documents* disinterest:
  $\rightarrow$ Use, whenever possible!
- 'X' signals:
  - the evaluation of undetermined values ('X', 'Z', 'W', '-'), or
  - the driving of a bus with different values $\rightarrow$ **error!**

Industry Standard Type: `std_logic`

use IEEE.std_logic_1164.**all**;
subtype std_logic **is** resolved std_ulogic;
Multiple sources on one signal are resolved according to:

	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

Similar tables are used for the computations of **and**, **or**, ...

Hardware-Oriented Arithmetic Data Types

use IEEE.numeric_std.**all**;
For the representation of unsigned and signed numbers in two's complement with fixed but *arbitrary* bit length:

- Definitions:
  unsigned: **type** unsigned **is array**(natural **range** <>) **of** std_logic;
  signed: **type** signed **is array**(natural **range** <>) **of** std_logic;
- . . . only differ in the implementation of sign-dependent arithmetic like $*$ and comparisons like < or >=.
- . . . permit a direct addition of integer:
  Count <= Count + 1;
- . . . enable:
  - a secure control over data widths, and
  - a simple description of bit (slicing) operations.

Type Juggling

- . . . serves the strict enforcement of the type concept,
- . . . does usually *not* induce hardware,
- . . . is implemented by VHDL functions:

```
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;

-- Pseudo-functions for "Related Types":
function signed  (arg : std_logic_vector) return signed;
function unsigned(arg : std_logic_vector) return unsigned;
function signed  (arg : unsigned)         return signed;
...
```

Signals

- Signals serve the communication between processes and modules.
- A signal represents *one* typed physical data node.
- Signal values are updated through events scheduled for the *future* (no sooner than one delta cycle later):
  → Signal values remain *constant* during a delta cycle.

Updating Signals

Updates of one and the same signal:

- can only be scheduled to occur in the future:
  ```
  a <= ... ;          – at t + Δ
  a <= ... after d;   – at ⌊t⌋ + d
  ```
- may be grouped into one statement:
  ```
  a <= ... , ... after d, ... after ...;
  ```

Careful:

```
a <= '0';
wait for 5 ns;
a <= not a;
```

differs from

```
a <= '0',
     not a after 5 ns;
```

(Assume a was '1' and evaluate both code snippets.)

Delayed and grouped assignments are rare when designing for synthesis but quite common in testbenches for simulation.

Reacting on Signals

Processes can pause in anticipation of a signal update:

```
wait on a, b;          – Wait for any event on one of a or b
wait until a = '1';    – Wait for a boolean condition to turn true
```

wait clauses may be combined (*Avoid in real designs!*):

```
wait on a until b = '1' for 50 ns;
  – Waits for an event to occur on a while b is '1' with a timeout of 50 ns.
```

If an **until** clause appears without an **on** clause, aka. the *sensitivity list*, the sensitivity list is inferred to contain all the signals occurring in the **until** clause.

Modeling Combinatorics

Combinatoric circuits compute a (delayed) result only from their current inputs:

```
process
begin
  c <= a and b;    -- The actual function realized, here: AND
  wait on a, b;    -- Re-compute whenever an input changes
end process;
```

Sensitivity list is moved into the **process** specification (a *must* for synthesis):

```
process(a, b)      -- Sensitivity list: all inputs
begin
  c <= a and b;
end process;
```

Single-statement processes may be simplified into a *concurrent statement*:

```
c <= a and b;    -- Sensitive on all signals on right side
```

Concurrent Statements

Concurrent statements do not provide a control flow (**if**, **case**, **for**) to model behavior but there are:

- Conditional Assignment

```
y <= '0' when clr = '1' else
     '1' when set = '1' else
      x;
```

  - Prioritized selection of *first* hit.
  - Final alternative necessary for combinatorics (to avoid state).

- Selective Assignment

```
with s select
  y <= a when "00",
       b when "01" | "10",
       c when others;
```

  - Balanced Multiplexer.
  - Exhaustive case listing may be completed by **others**.

Sequential Statements

Processes do have a control flow and use sequential statements:

```
-- Branches
if <cond1> then
  ...
elsif <cond2> then
  ...
end if;

-- Loops
for i in 0 to N loop
  a(i) <= '0';
end loop;
```

```
-- Case Selections
case s is
  when "00" =>
    ...
  when "01" | "10" =>
    ...
  when others =>
    ...
end case;
```

Lab

Waveforms and Combinatorics