

VHDL Primer

Tutorial #3

Mike Goldsmith

Feb 10, 2004, ~ 2 hr duration

Outline

- New Types and Subtypes
- Composite Data Types
- Type Attributes
- Signal Attributes
- Generic parameters
- Configuration block

New Types and Subtypes

- Make your own signal types
- Must specify all possible signal values

```
type engine_states is (off, starting, running, braking);  
signal foo : engine_states := off;
```

```
if foo = running then...
```

New Types and Subtypes

- Subtypes create a narrower *range*
- Must specify range

subtype natural is integer **range** 0 to *highest_integer*;

subtype X01 is std_logic **range** 'X' to '1';

signal a, b : natural := -5; --invalid, -5 is not in range

signal c : X01 := '0';

New Types and Subtypes

- Types can also have **units**, referred to as *physical types*

type resistance **is range** 0 to 1E9

units

ohm;

kohm = 1000 ohm;

Mohm = 1000 kohm;

end units resistance;

signal r1,r2,r3 : resistance;

if (r1 / 2.5 = 9 kohm) **or** (r2/r3 = 2) **then**

Composite Types

- Arrays and Records
 - Array: a ‘collection of signals’
 - Size is determined by the *range*
 - Can be specified as increasing or decreasing
- type point is array (1 to 2) of real;**
- type plane is array (1 to 2, 1 to 2) of real;**

Composite Types

- **Constrained Arrays:** all instances have the same range

type word **is array** (31 **downto** 0) **of** bit;

signal foo : word := X"FE24_CD58"; --X denotes hex

- **Unconstrained Arrays:** each instance must specify the range

type std_logic_vector **is array** (natural **range** $\langle \rangle$) **of** std_logic;

signal foo : std_logic_vector (15 **downto** 0); -- little endian

signal oof : std_logic_vector (0 **to** 15); -- big endian

Composite Types

- Can assign *subrange* of one vector to another (or to a signal)

```
signal alpha, beta : std_logic_vector(7 downto 0);
```

```
signal gamma : std_logic;
```

```
beta(3 downto 0) <= alpha(7 downto 4);
```

```
beta(7 downto 4) <= "1--1" –use double quotes instead
```

```
gamma <= alpha(2);
```


Composite Types

- Can use aggregates to assign initial values to arrays

- Positional aggregate:

```
signal x,y : point := (0.0,0.0);
```

- Named aggregate:

```
type engines is array (1 to 16) of engine_states;
```

```
signal theEngines : engines :=(1|5|7 =>running, 2 => braking,  
  others => off);
```

Composite Types

- Records: like structs... group signals of possibly different types

type timestamp **is record**

hours : integer **range 0 to 23**;

minutes : integer **range 0 to 59**;

seconds : integer **range 0 to 59**;

end record timestamp;

signal current_time : timestamp;

current_time.seconds <= clock **mod 60**;

Composite Types

- Record aggregates: assigning initial values
 - Basically named aggregates

signal current_time : timestamp := (hours=>0, minutes=>30, seconds=>0);

- You can have constrained arrays within records, and can have arrays *of* records, both of which can make aggregates tricky

signal a : radialarc := (start=> (x=>0.0, y=>0.0), end=> (x=>3.0, y=>5.0), radius=>1.5);

Type Attributes

- For any signal of type ‘T’

Attribute	Description	negative	std_logic
T'left	First/ Leftmost value	<i>int_low</i>	‘U’
T'right	Last/ Rightmost value	-1	‘-’
T'low	Lowest Value	<i>int_low</i>	‘U’
T'high	Highest Value	-1	‘-’
T'ascending	Boolean, Range Ascending	false	true

Type Attributes

- For finite-ranged and physical types ‘T’

Attribute	Description	engine
T’ pos (x)	Position number of ‘x’ in T	engine’ pos (off) = 0
T’ val (n)	Value of ‘n’ in T	engine’ val (2) = running

- For any subtype ‘U’ of type ‘T’, there is an additional attribute: **base**

U’**base** = T; U’**base**’**right** = T’**right**, but maybe not U’**right**

Signal Attributes

- For any signal 'S'

Attribute	Description
S'event	Boolean, True if there is an event on S in the current simulation cycle
S'stable(T)	Boolean, True if there has been no change in S within time T (before now)
S'delayed(T)	A clone of S delayed by T
S'active	Boolean, true if there is a transaction on S in the current simulation cycle
S'last_event	Time since last event on S
S'last_value	Value of S before last event
S'range	The range of S (if an array)

Generic parameters

- A parameter used within the architecture that can be set upon instantiation of the module
- Declared in the entity block

```
entity reg_gen is
```

```
  generic( WIDTH : positive := 8);
```

```
  port(      clk : in std_logic;
```

```
           reset_n: in std_logic;
```

```
           d: in std_logic_vector (WIDTH-1 downto 0);
```

```
           q: out std_logic_vector (WIDTH-1 downto 0)
```

```
  );
```

```
end entity reg_gen;
```

Generic parameters

```
architecture rtl of reg_gen is
    signal FF : std_logic_vector (WIDTH-1 downto 0);
begin
    process (clk, reset_n, d) is
        begin
            if reset_n = '0' then
                FF <= (FF'range => '0'); --use of aggregate
            else
                if rising_edge(clk) then
                    FF <= d; --even though the size isn't fixed
                end if;                --they're still the same size...
            end if;
        end process;
        q <= FF;
    end architecture rtl;
```


Generic parameters

- Instantiating a component with generics

architecture foo of bar is

component reg_gen is

generic(WIDTH : positive := 8);

port(clk : **in** std_logic;

reset_n: **in** std_logic;

d: **in** std_logic_vector (WIDTH-1 **downto** 0);

q: **out** std_logic_vector (WIDTH-1 **downto** 0)

);

end component reg_gen;

...

Generic parameters

...

```
signal clk, reset_n : std_logic;
```

```
signal a,b : std_logic_vector(15 downto 0);
```

```
signal c,d : std_logic_vector(31 downto 0);
```

```
begin
```

```
reg16: reg_gen generic map( WIDTH =>16) --NO SEMICOLON
```

```
    port map(clk=>clk, reset_n=>reset_n, d=>a, q=>b);
```

```
reg32: reg_gen generic map( WIDTH =>32)
```

```
    port map(clk=>clk, reset_n=>reset_n, d=>c, q=>d);
```

```
end architecture foo;
```

- Two different sized registers; same code

Configuration blocks

- **Disclaimer:** I lied. Component blocks aren't *needed* to instantiate a module
- Alternative construct:

architecture foo of bar is

begin

reg16: entity work.reg_gen(rtl)

generic map(WIDTH=>16) **port map** (...);

end architecture foo;

- Choice of architecture upon instantiation is **fixed**

Configuration blocks

- Reasons that components are *useful* is that you can use the construct of the **configuration** block to choose a specific *architecture* for specific *instances*, based on which *architecture* of the corresponding *entity* is chosen
- Wow, that was a mouthful, but what does it mean?

Configuration blocks

- Syntax:

configuration *identifier of entity* **is**

for *architecture*

for *component_id: component_type*

[**use entity** *entity_name(architecture)*];

[**use configuration** *config_name*];

[nest based on component architecture]

end for;

end for;

[alternate architecture]

end configuration *identifier*;

Configuration blocks

```
configuration full of foo is
  for bar
    for reg16: reg_gen
      use entity work.reg_gen(rtl);
    end for;
    for others: reg_gen
      use entity work.reg_gen(behavioural);
      for behavioural
        for all: widgets
          use configuration work.widget_cfg;
        end for;
      end for;
    end for;
  end for;
end configuration full;
```

Configuration blocks

- Using common mappings to **reduce typing**
configuration full of foo is

for bar

for all: reg_gen

use entity work.reg_gen(rtl)

generic map (rt=>20 ns, ft=>30 ns, WIDTH=>WIDTH)

port map (clk=>clk, reset_n=>reset_n, d=>open, q=>open);

end for;

end for;

end configuration full;

- Allows for **back-annotation** of parameters

Configuration blocks

- Because of configuration, we only have to map **open** ports and **unmet** generics

architecture foo of bar is

begin

```
reg16: reg_gen generic map( WIDTH =>16)  
    port map(d=>a, q=>b);
```

```
reg32: reg_gen generic map( WIDTH =>32)  
    port map(d=>c, q=>d);
```

end architecture foo;