Hierarchical Design with VHDL

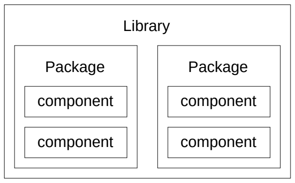
This chapter covers some features of VHDL that are useful for logic synthesis. You should learn to: make library packages visible; declare components in architectures and packages; declare constants; instantiate components into an architecture, declare std_logic, std_logic_vector, signed and unsigned signals; declare enumerated types and subtypes of array types; in architectures and packages use conditional signal assignments; and convert between std_logic_vector, unsigned and integer types.

Libraries, Packages and Components

When designing complex logic circuits it helps to decompose a design into simpler parts. Each of these parts can be written and tested separately, perhaps by different people. If the parts are sufficiently general then it's often possible to re-use them in future projects. In VHDL, design re-use is done by using "components." A component can be a general-purpose building-block (e.g. an adder or a counter), or it can be sub-system of your design.

Before we use a component, we first need to declare it. A component declaration is very similar to an entity declaration — it defines the input and output signals, not the functionality.

In order to avoid declaring each component in every architecture where it is used, we typically place component declarations in "packages." A package is a database containing information about the components in the package (each component's inputs, outputs, types, etc). A package typically contains a set of component declarations for a particular application. Packages are themselves stored in "libraries":



To use a component in a design, we use `library` statements to specify the libraries to be searched and a `use` statement for each package we need to use. The two most commonly used libraries are called `IEEE` and `WORK`.

The `WORK` library is specific to a project and is available without having to use a library statement.

`library` and `use` statements must be used before *each* design unit (entity or architecture) that uses those packages[1]. For example, if you wanted to use the `numeric_bit` package in the `ieee` library you would use:

```
library ieee ;
use ieee.numeric_bit.all ;
```

and if you wanted to use the `dsp` package in the `WORK` library you would use:

```
use work.dsp.all ;
```

Exercise 1: Why is there no library statement in the second example?

Note that a component defines an interface to another device. That device may not have been designed using VHDL so there may not necessarily be a corresponding entity declaration.

Creating Components

A component declaration is similar to an entity declaration and defines the input and output signals.

Component declarations can be placed in an architecture before the `begin`. But it's usually more convenient to put component declarations within a `package` declaration to avoid duplicating the declaration. When we compile (or "analyze") the package declaration the information about the components in the package is saved in the WORK library. The components in the packages can then be used in an architecture (in that same file or in other files) by using the appropriate `use` statements.

For example, the following code declares a package called `flipflops`. This package contains only one component, `rs`, with inputs `r` and `s` and an output `q`:

[1] An exception: when an architecture immediately follows its entity you need not repeat the `library` and `use` statements.

```
package flipflops is
   component rs
      port ( r, s : in bit ; q : out bit ) ;
   end component ;
end flipflops ;
```

Exercise 2: If this code was analyzed, what information would be saved? Where?

Component Instantiation

Once a component has been declared, it can be used ("instantiated") in an architecture. A component instantiation describes how the component's ports are connected to signals in the architecture. It is a *concurrent* statement (as is a selected assignment statement).

The following example shows how three 2-input exclusive-or gates can be used to build a 4-input parity-check circuit using component instantiation. This type of description is called *structural* as opposed to *behavioural* VHDL because we are defining the structure rather than the behaviour of the circuit.

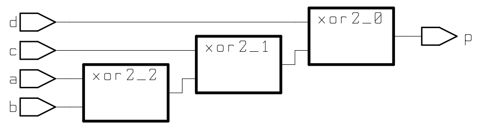
In this case we have put the component declaration into the file mypackage.vhd. The xor_pkg contains the xor2 component (although a typical package defines more than one component):

```
-- define an xor2 component in a package

package xor_pkg is
   component xor2
      port ( a, b : in bit ; x : out bit ) ;
   end component ;
end xor_pkg ;
```

A second file, parity.vhd, describes the parity entity that uses the xor2 component:

```
-- parity function built from xor gates

use work.xor_pkg.all ;

entity parity is
   port ( a, b, c, d : in bit ; p : out bit ) ;
end parity ;

architecture rtl of parity is
   -- internal signals
   signal x, y : bit  ;
begin
   x1: xor2 port map ( a, b, x ) ;
   x2: xor2 port map ( c, x, y ) ;
   x3: xor2 port map ( d, y, p ) ;
end rtl ;
```

The resulting top-level schematic for the parity entity is:



Exercise 3: Label the connections within the parity generator schematic with the signal names used in the architecture.

When the parity.vhd file is analyzed ("compiled"), the synthesizer will search the (WORK) library for the xor_pkg package.

We could also have put the xor_pkg package declaration in the parity.vhd file (the package would then be recreated every time we analyzed parity.vhd).

Although components don't necessarily have to be created using VHDL, we could have done so by using the following entity/architecture pair in file called xor2.vhd:

```
-- xor gate

entity xor2 is
   port ( a, b : in bit ; x : out bit ) ;
end xor2 ;

architecture rtl of xor2 is
begin
   x <= a xor b ;
end rtl ;
```

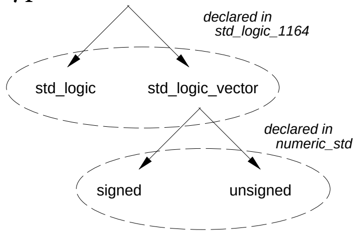
std_logic Packages

The IEEE library contains two useful packages. These packages define alternatives to the bit and bit_vector types for logic design.

The first package, std_logic_1164, defines the types std_logic (similar to bit) and std_logic_vector (similar to bit_vector). The advantage of the std_logic types is that they can have values other than '0' and '1'. For example, an std_logic signal can also have undefined ('X', useful for simulation), high-impedance values ('Z' useful for implementing tri-state outputs)[2]. The std_logic_1164 package also redefines ("overloads") the standard boolean operators (and, or, not, etc.) so that they work with std_logic signals.

[2]Don't care, '-', can sometimes be used for synthesis.

The second package, `numeric_std`[3] defines the types `signed` and `unsigned`. These are subtypes of `std_logic_vector` with overloaded operators that allow them to be used both as vectors of logic values and as as binary numbers (in signed two's complement or unsigned representations). The hierarchy of these logic types could be drawn as follows:

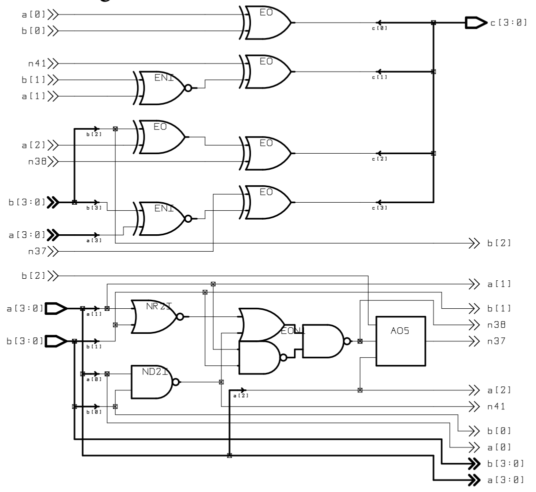


The standard arithmetic operators (+, −, *, /, **, >, <, <=, >=, =, /=) can be applied to signals of type `signed` or `unsigned`. Note that it may not be practical or possible to synthesize complex operators such as multiplication, division or exponentiation.

For example, we could generate the combinational logic to build a 4-bit adder as follows:

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

entity adder4 is
   port (
   a, b : in unsigned (3 downto 0) ;
   c : out unsigned (3 downto 0) ) ;
end adder4 ;

architecture rtl of adder4 is
begin
   c <= a + b ;
end rtl ;
```

The resulting schematic is:



[3] Another package, `std_logic_arith` is widely used.

Constants

We can declare symbolic constants in the same way as signals. For example:

```
constant zero_bits : unsigned (3 downto 0) := "0000" ;
```

A constant declared in a package is available to all design units (packages, entities and architectures) that use that package. You should use symbolic constants for any values that are likely to change or if it makes your code easier to read or easier to modify.

Integers

VHDL also includes an `integer` type which is useful for specifying small constants (e.g. `next_x <= x + 1 ;`). However, *signals* should be declared `std_logic` or one of its subtypes, *not* `integer`.

Type Conversion Functions

VHDL is a strongly-typed language – each operator must be supplied arguments of exactly the right type or the synthesizer will give an error message. Although many functions and operators (e.g. and) are overloaded so that you can use the same function/operator with more than one type, in many cases you will need to use type conversion functions.

The following type conversion functions are found in the the `std_logic_1164` package in the `ieee` library:

from	to	function
lv	bv	`to_bitvector(x)`
bv	lv	`to_stdlogicvector(x)`

The following type conversion functions are found in the the `numeric_std` package in the `ieee` library.

from	to	function
un,lv	un	`signed(x)`
sg,lv	un	`unsigned(x)`
un,sg	lv	`std_logic_vector(x)`
un,sg	in	`to_integer(x)`
in	un	`to_unsigned(x,len)`
in	sg	`to_signed(x,len)`

The abbreviations bv, lv, un and in are used for bit_vector, std_logic_vector, unsigned and integer respectively.

Functions in the numeric_std package "overload" most of the arithmetic and comparison operators (e.g. +, =) so that they take integer as well as unsigned operands. Note that when converting an integer you must explicitly specify the number of bits in the result (len).

For example:

```
 constant awidth : integer := 24 ;
 constant dwidth : integer :=  8 ;
 constant r1addr : std_logic_vector (awidth-1 downto 0)
              := std_logic_vector(X"1A_0002") ;
 signal abus : unsigned (awidth-1 downto 0) ;
 signal r1, d : std_logic_vector (dwidth-1 downto 0) ;
...
 r1 <=
    d when abus = to_unsigned(r1addr,awidth) else
    b"0000_0000" ;
```

Exercise 4: What is the type of the constant X"1A_0002"? What is the purpose of the to_unsigned() function in the last line of the above example? What conversion function(s) would you need to use if r1addr was declared to be of type bit_vector?

Conditional Assignment

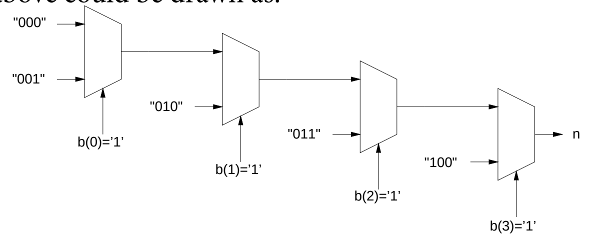
In the same way that a selected assignment statement models a case statement in a sequential programming language, a conditional assignment statement models an if/else statement. Like the selected assignment statement, it is also a *concurrent* statement.

For example, the following circuit outputs the position of the left-most '1' bit in the input:

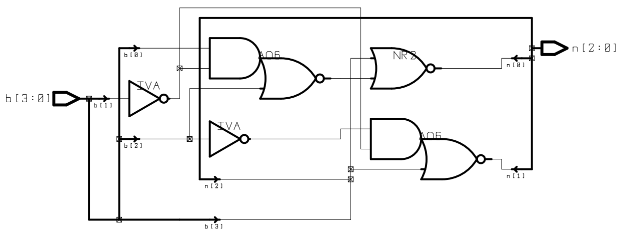
```
library ieee ;
use ieee.std_logic_1164.all ;

entity nbits is port (
   b : in std_logic_vector (3 downto 0) ;
   n : out std_logic_vector (2 downto 0) ) ;
end nbits ;

architecture rtl of nbits is
begin
   n <=
      "100" when b(3) = '1' else
      "011" when b(2) = '1' else
      "010" when b(1) = '1' else
      "001" when b(0) = '1' else
      "000" ;
end rtl ;
```

Note that the conditions are tested in the order that they appear in the statement and only the first value whose controlling expression is true is assigned.

In the same way that we can view a selected assignment statement as the VHDL model for a ROM or lookup table, a conditional assignment statement can be viewed the VHDL description of a tree of multiplexers. For example, the structure of the example above could be drawn as:



Synthesizing the above description results in:



Exercise 5: Write a conditional assignment that models a 2-to-1 multiplexer. Use an array x as the input, a signal sel to select the input and a signal y as the output. Repeat for a 4-to-1 multiplexer (sel is now an array).

The choice of selected or conditional assignments can affect the logic that is generated. A conditional assignment implies an ordered sequence of two-way decisions which results in the multiplexer tree as shown above. A selected assignment implies a logic circuit that evaluates all possible inputs simultaneously. This implies a single-stage sum-of-products (or equivalent) circuit. The circuit generated by a selected assignment will typically require less logic but will incur a longer propagation delay.

However the logic synthesizer may need to optimize the original circuit to meet either speed or space constraints. The final circuit may not match either of the above models.

Reserved Words

Note that there are 97 fairly common words (e.g. next, new, open) that are reserved words and cannot be used as VHDL identifiers.