

VHDL for Complex Designs

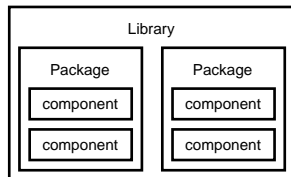
This lecture introduces some aspects of VHDL that are useful when designing complex logic circuits. After this lecture you should be able to:

- *make library packages visible*
- *save entities as components in packages*
- *instantiate components into an architecture*
- *declare `std_logic`, `std_logic_vector`, signed and unsigned signals*
- *declare enumerated types and subtypes of array types and save them in a package*
- *use conditional and selected signal assignments*
- *instantiate tri-state outputs*

Libraries, Packages and Components

When designing complex logic circuits it's desirable to decompose the 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 might also be possible to re-use them in future projects.

Re-use in VHDL is done by saving these parts (called "components") in "packages". A package typically contains a set of components for a particular application. Packages are stored in "libraries":



To make the components in a package available ("visible") in another design, we use `library` statements to specify the libraries to be searched and a `use` statement for each package we wish to use. The two most commonly used libraries are called `IEEE` and `WORK`.

In the Synopsys VHDL implementation a library is a directory and each package is a file in that directory. The package file is a database containing information about the components in the package (inputs, outputs, types, etc).

The `WORK` library is always available without having to use a library statement. It is simply the `WORK`

subdirectory into which the current design is placed.

`library` and `use` statements must be used before each entity that makes use of components found in those packages. 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: Why is there no `library` statement in the second example?

Creating Components

To create components we put component declarations within a package declaration. When we "analyze" the file (with the `read` command in `dc_shell`) the information about the components in the package are saved in a file with the package name in the `WORK` library.

These components can then be used in other designs by making them visible with a `use` statement. A component declaration is similar to an entity declaration and simply defines the input and output signals. Note that declaring a component does not create any hardware – only when the component is used

in an architecture is the hardware generated (“instantiated”).

For example, the following code creates a package called `flipflops` containing only one component called `rs` with inputs `r` and `s` and an output `q` when it is analyzed:

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

Exercise: If you analyzed this code, what file would be created? Where would it be?

Component Instantiation

Once a component has been placed in a package, it can be used (“instantiated”) in an architecture. A component instantiation simply describes how the component is “hooked up” to the other signals in the architecture. It is thus a *concurrent* statement like the `process` statement rather than a *sequential* statement. A component instantiation cannot be put inside a `process`.

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

```
-- xor gate

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

architecture rtl of xor2 is
begin
  process(a,b)
  begin
    x <= a xor b ;
  end process ;
end rtl ;

-- put xor2 component in a package

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

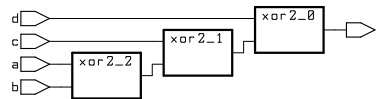
```
-- parity function built from xor gates

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

Type Declarations

It’s often useful to make up new types of signals for a project. We can do this in VHDL by including type declarations in packages. The two most common uses for defining new types are to declare arrays of given dimensions (e.g. a bus of a given width) and to declare types that can only have one of a set of possible values (called enumeration types).

The following example shows how a package called `dsp_types` that declares two new types is created:

```
package dsp_types is
  subtype sample is bit_vector (7 downto 0) ;
  type mode is (slow, medium, fast) ;
end dsp_types ;
```

std_logic Packages

In the IEEE library there are two packages that are often used. 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 a high-impedance value ('Z'). The `std_logic_1164` package also overloads the standard boolean operators so that they also work with `std_logic` signals.

The second package is called `std_logic_arith`¹ and defines the types `signed` and `unsigned`. These are subtypes of `std_logic_vector` with overloaded operators that allow them to be used as both vectors of logic values and as binary values (in two's complement or unsigned representations). Although the standard arithmetic operators (+, -, *, /, **) can be applied to signals of type `signed` or `unsigned`, it may not be practical or possible to synthesize complex operations such as multiplication, division or exponentiation.

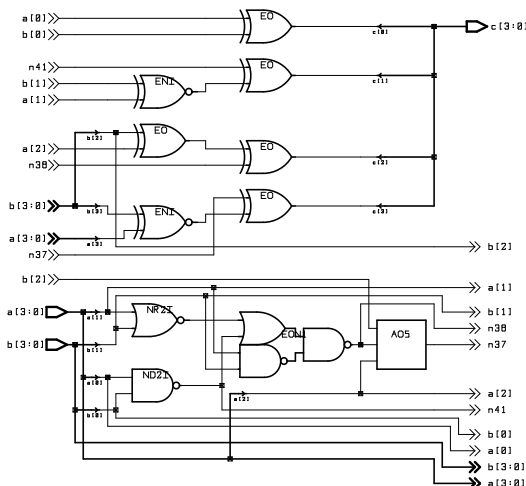
For example, we could generate the combinational logic to build a 4-bit adder using the following architecture:

```
library ieee ;
use ieee.std_logic_1164.all ;
use ieee.std_logic_arith.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 (rather messy) schematic is:



¹The IEEE standard is really `numeric_std` but it's not widely available yet.

Concurrent Assignment

A complex architecture often has many processes that only compute the value of a signal using an if/else or a case statement. The selected and conditional assignment statements are shorthand ways of describing such processes in VHDL. The conditional assignment is equivalent to a compound if/else statement embedded within a process; the selected assignment is equivalent to a case statement embedded within a process. These are *concurrent* statements.

For example, we can define an equivalence gate and a 4-bit 3-input multiplexer using a process as follows:

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

entity eqv is
  port ( a, b : in std_logic ; x : out std_logic ) ;
end eqv ;

architecture rtl of eqv is
begin
  process(a,b)
  begin
    if a = b then
      x <= '1' ;
    else
      x <= '0' ;
    end if ;
  end process ;
end rtl ;

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

```
entity mux3 is
  port (
    a, b, c : in std_logic_vector (3 downto 0) ;
    sel : in unsigned (1 downto 0) ;
    x : out std_logic_vector (3 downto 0) ) ;
end mux3 ;

architecture rtl of mux3 is
begin
  process(a,b,c,sel)
  begin
    case sel is
      when "00" => x <= a ;
      when "01" => x <= b ;
      when "10" => x <= c ;
      when others => x <= a ;
    end case ;
  end process ;
end rtl ;
```

But we can write equivalent architectures as follows:

The resulting schematic for the tbuf is:

```

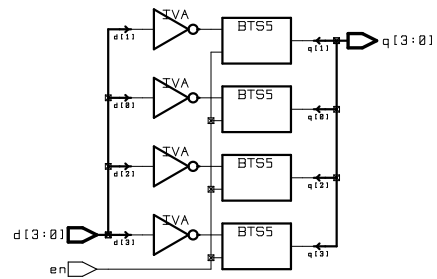
...

architecture rtl of eqv is
begin
  x <= '1' when a = b else '0' ;
end rtl ;

...

architecture rtl of mux3 is
begin
  with sel select x <=
    a when "00" ,
    b when "01" ,
    c when "10" ,
    a when others ;
end rtl ;

```



Tri-State Buses

A tri-state output can be set to the normal high and low logic levels as well as to and a high-impedance state. This type of output is often used in buses where different devices must drive the bus at different times. One way to specify that an output should be set to the high-impedance state is to use an `std_logic` type and assign it a value of 'Z'.

The following example shows an implementation of a 4-bit buffer with an enable output. When the enable is not asserted the output is in high-impedance mode :

```

library ieee ;
use ieee.std_logic_1164.all ;

entity tbuf is port (
  d : in std_logic_vector (3 downto 0) ;
  q : out std_logic_vector (3 downto 0) ;
  en : in std_logic
) ;
end tbuf ;

architecture rtl of tbuf is
begin
  process(d,en)
  begin
    if en = '1' then
      q <= d ;
    else
      q <= "ZZZZ" ;
    end if ;
  end process ;
end rtl ;

```