VHDL for Complex Designs

This lecture covers VHDL features that are useful when designing complex logic circuits. After this lecture you should be able to:

- *make library packages visible*
- *declare components and save them 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 packages*
- *use conditional signal assignments*
- *convert between std_logic_vector, unsigned and integer types*
- *instantiate tri-state outputs*
- *model RAM using arrays*

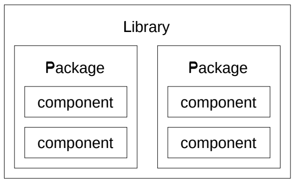
Libraries, Packages and Components

As in the design of any complex system, in the design of complex logic circuits it's often a good idea 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.

A similar principle holds for software. We break down complex programs into smaller subroutines or functions, each of which is easier to write, test, and re-use.

In VHDL, we can re-use an entity in another entity's architecture by declaring the first entity as a "component." A component declaration is very similar to its corresponding entity declaration — it only declares the input and output signals.

In order to avoid declaring each component in every architecture where it is used, we typically place component declarations in "packages." A package typically contains a set of component declarations for a particular application. Packages are themselves 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 Design Compiler[1] and Max+PlusII VHDL implementations 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 (the component inputs, outputs, types, etc).

The `WORK` library is always available without having to use a library statement. In Design Compiler the `WORK` library is a subdirectory of the current directory called `WORK` while in Max+PlusII it is the current project directory.

`library` and `use` statements must be used before *each* design unit (entity or architecture) that makes use of components found in those packages[2]. 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:

[1]The logic synthesizer used to create the schematics in these lecture notes.

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

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

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

Creating Components

To create components, we usually put component declarations within a `package` declaration. When we compile (or "analyze") the file that contains the package and component declarations the information about the components in the package will be saved in a file with the name of the package (and, in Max+PlusII, the extension `.vhdlview`) in the WORK library. The components in the package 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 a component declaration does not create hardware – only when the component is used in an architecture is the hardware generated ("instantiated").

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` when it is compiled:

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

Exercise: If you this code was stored in a file called `ff.vhd`, what file would be created? Where would it be placed?

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 and 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.

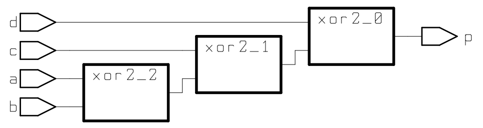
There are two files: the first file, `mypackage.vhd`, describes the `xor2` component (although a typical package defines more than one component):

```
-- 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 ;
```

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

When the `parity.vhd` file is compiled, the compiler will search the (`WORK`) directory for the `xor_pkg` package.

Although components don't necessarily have to be created using VHDL descriptions (they may have been created using design methods other than 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 ;
```

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 most common uses for defining new types are to create signals of of a given width (i.e. a bus) 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 three new types is created:

```
package dsp_types is
   type mode is (slow, medium, fast) ;
   subtype word is bit_vector (15 downto 0) ;
end dsp_types ;
```

Note that we need to use a `subtype` declaration in the second example because the `bit_vector` type is already defined. Type declarations are often placed in packages to make them available to multiple design units.

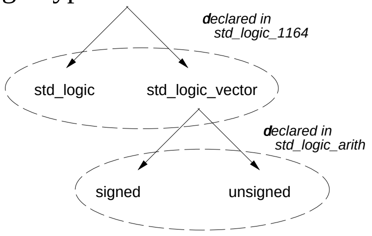
Exercise: Write a declaration for a signal that controls whether the value in a register should be loaded, incremented, decremented, or held. Write the declaration for an 8-bit signal type called `byte`.

`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 redefines ("overloads") the standard boolean operators so that they also work with `std_logic` signals.

The second package is called `std_logic_arith`[3] 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 a binary values (in two's complement or unsigned representations). The hierarchy of these logic types could be drawn as follows:



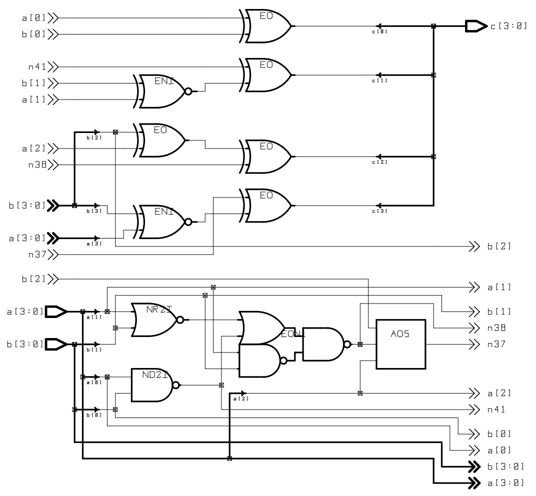
Although the standard arithmetic operators (`+, -, *, /, **`) can be applied to signals of type `signed` or `unsigned`, it may not be practical or possible to synthesize complex operators such as multiplication, division or exponentiation.

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:

[3]The IEEE standard is really `numeric_std` but it's not widely used yet.



Conditional Assignment

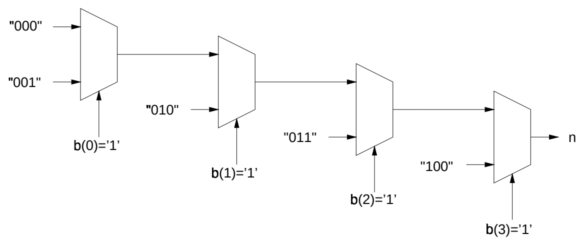
In the same way that a selected assignment statement models a case statement in a sequential programming language, a conditional assignment statement models an if/else statement. Like the selected assignment statement, it is also a *concurrent* statement.

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

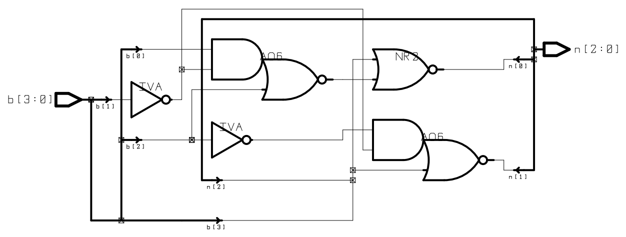
```
library ieee ;
use ieee.std_logic_1164.all ;

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 was 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 one or more multiplexers. For example, the structure of the example above could be drawn as:



Synthesizing the above description results in:



Exercise: 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).

Type Conversion Functions

VHDL is a strongly-typed language – each function or operator must be supplied arguments of exactly the right type or the compiler will give an error message. Although many functions/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 table shows the type conversion functions found in the the `std_logic_arith` package in the `ieee` library. The abbreviations `lv`, `un` and `in` are used for `std_logic_vector`, `unsigned` and `integer` respectively.

from	to	function
lv	un	`unsigned(x)`
lv	in	`conv_integer(x)`
un	lv	`std_logic_vector(x)`
un	in	`conv_integer(x)`
in	un	`conv_unsigned(x,len)`
in	lv	`conv_std_logic_vector(x,len)`

Note that when converting an `integer` you must explicitly specify the number of bits in the result (`len`).

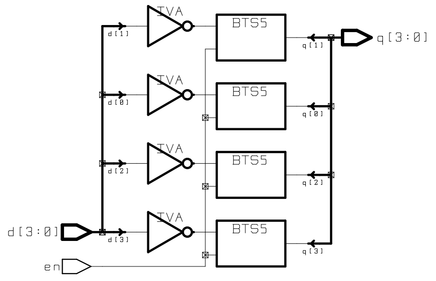
Tri-State Buses

A tri-state output can be set to the normal high and low logic levels as well as to a third state: high-impedance. This type of output is often used where different devices drive the same bus at different times (e.g. a data bus). The most common way to specify that an output should be set to the high-impedance state is to use a signal of type `std_logic` 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
   q <=
      d when en = '1' else
      "ZZZZ" ;
end rtl ;
```

The resulting schematic for the `tbuf` is:



Tri-state outputs are used primarily to implement bidirectional bus signals. Such signals are declared of type `inout` rather than `in` or `out` and their values can be used within the architecture (unlike signals of type `out`). When the bus is to act as an input, the bidirectional bus signals are driven to the high-impedance state and in this case it's the value of signals outside the entity that determine the signal's value.

RAM Models

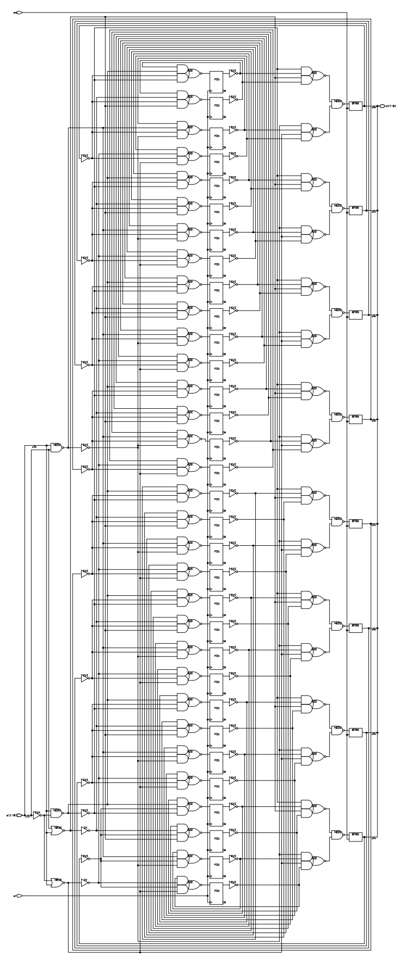
VHDL also allows the use of arrays with signal indices to model random-access memory (RAM). The following example demonstrates the use of VHDL arrays as well as bi-directional buses. We must use the type-conversion function `conv_integer` because the address input, `a`, is of type `unsigned` while the array index must be of type `integer`.

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

entity ram is   port (
   -- bi-directional data signal
   d : inout std_logic_vector (7 downto 0) ;
   -- address input
   a : in unsigned (1 downto 0) ;
   -- output enable and write strobe (clock)
   oe, wr : in std_logic ) ;
end ram ;
```

```
architecture rtl of ram is
   subtype byte is std_logic_vector (7 downto 0) ;
   type byte_array is array (0 to 3) of byte ;
   signal ram : byte_array ;
begin
   -- output value is the indexed array element
   d <=
      ram(conv_integer(a)) when oe = '1' else
      "ZZZZZZZZ" ;

    -- register the indexed array element
   process(wr)
   begin
      if wr'event and wr = '1' then
         ram(conv_integer(a)) <= d ;
      end if ;
   end process ;
end rtl ;
```

Exercise: Modify the design above to create a 16-element, 4-bit wide RAM with separate input and output signals.

The result of synthesizing this description is:



For many implementation technologies (FPGAs, gate arrays, or standard-cell ASICs) there are usually vendor-specific ways of implementing memory arrays that give better results. However, using a VHDL-only model with “random logic” as shown above is more portable.

Exercise: Why is portability desirable?