Integrated Microelectronics Engineering, Module 1

Introduction to VHDL

Part 1: VHDL for Design

Ed Casas

VHDL is a Very-complex[1] Hardware Description Language used to
(2) design and simulate digital logic circuits.
(3)

Introduction

Let's start with a simple example – a type of circuit called an example1 that has one output signal (`c`) which is the AND function of two input signals (`a` and `b`). The file `example1.vhd` contains the following VHDL
(5) description:

```
-- An AND gate

library ieee ;
use ieee.std_logic_1164.all;

entity example1 is port (
   a, b: in std_logic ;
   c: out std_logic ) ;
end example1 ;

architecture rtl of example1 is
begin
      c <= a and b ;
end rtl ;
```

First some observations on VHDL syntax:

- VHDL is case-insensitive. There are many capitalization styles. I prefer all lower-case. Whichever style you use, be consistent.
- Everything following two dashes "`--`" on a line is a comment and is ignored.
- Statements can be split across any number of lines. A semicolon ends each statement. Indentation styles vary but an "end" should be indented the same as its corresponding "begin"
- Entity and signal names begin with a letter followed by letters, digits or underscore ("_") characters.

[1] Actually, the V stands for VHSIC and VHSIC stands for Very High Speed IC.

- the `library` and `use` statements need to be placed before each entity/architecture pair (more on this later).

6

A VHDL description has two parts: an entity part and an architecture part.

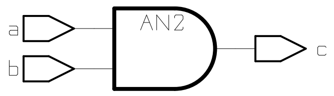
The entity part defines the input and output signals for the device or "entity" being designed while the architecture part describes the behaviour of the entity. Although it's possible to define several architectures for each entity, we'll only define one and call it `rtl`.

Each architecture is made up of one or more statements, all of which "execute[2]" at the same time (*concurrently*) and continuously. Concurrent execution allows us to describe the behaviour of hardware (which operates concurrently by nature). 7

The single statement in this example is a signal assignment that assigns the value of an expression to the output signal `c`. Expressions involving `std_logic` signals can use the logical operators `and`, `or`, `nand`, `nor`, and `xor`, all of which have equal precedence and `not` which has the highest precedence. Parentheses can be used to force evaluation in a different order.

From this VHDL description a program called a logic synthesizer can generate a circuit that behaves the same way as the VHDL description. In this case it's not too surprising that the synthesizer outputs the following circuit: 8

9



Exercise: Write the VHDL description of a half-adder, a circuit that computes the sum, `sum`, and carry, `carry`, of two one-bit numbers, a and b.

Vectors and Buses

We can use signals of type `std_logic_vector`, one-dimensional arrays of `std_logic`, to model buses. The declaration includes the allowed range of indices, for example:

```
a : std_logic_vector (3 downto 0)
```

The indices of `bit_vector`s are usually declared to have decreasing (`downto`) rather than increasing (`to`) values so that the first (leftmost) bit is the most significant one. `std_logic_vector` constants are formed

[2]The resulting hardware doesn't actually "execute" but this point of view is useful when using VHDL for simulation.

by enclosing an ordered sequence of `std_logic` values in double quotes (e.g. `"0010"`). (10)

Substrings ("slices") of vectors can be extracted by specifying a range in the index expression (e.g. `a(3 downto 2)`) and vectors can be concatenated using the & operator (e.g. `a & b`). The logical operators (e.g. `and`) can be applied to `bit_vector`s and operate on a bit-by-bit basis. (11)

(12) Exercise: Write a VHDL expression that shifts x, an 8-bit `std_logic_vector` declared as `x : std_logic_vector (7 downto 0) ;`, left by one bit and sets the least-significant bit to zero.

Selected Assignment

The selected assignment statement models the operation of a multiplexer – the value assigned is selected from several possible values by a controlling expression. Using `std_logic_vector`s and a selected assignment we can convert a truth table into a VHDL description. The following example describes a 3-to-8 decoder: (13)

(14)

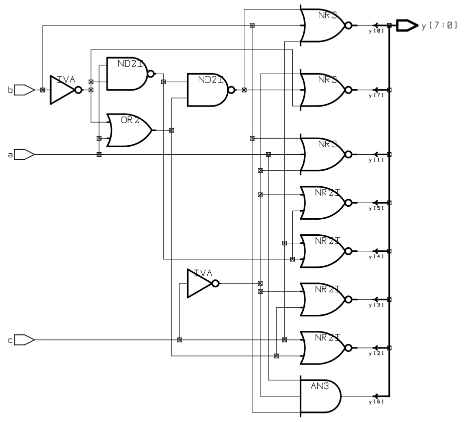
```
-- 3-to-8 decoder

library ieee ;
use ieee.std_logic_1164.all;

entity decoder is
  port (
  a, b, c : in std_logic ;
  y : out  std_logic_vector (7 downto 0) ) ;
end decoder ;
```

(15)

```
architecture rtl of decoder is
  signal abc : std_logic_vector (2 downto 0) ;
begin
  abc <= a & b & c ;

  with abc select y <=
     "00000001" when "000",
     "00000010" when "001",
     "00000100" when "010",
     "00001000" when "011",
     "00010000" when "100",
     "00100000" when "101",
     "01000000" when "110",
     "10000000" when others ;
end rtl ;
```

The keyword `others` indicates the default value to assign when none of the other values matches the selection expression. Always include an `others` clause.

Note that local signals can be declared between `architecture` and `begin`.
The VHDL description synthesizes to: (16)



Conditional Assignment

A conditional assignment statement behaves like if/else statement in a conventional programming language but, like the selected assignment statement, it is also a *concurrent* statement. Unlike the selected assignment, the expressions are tested in order and only the first value whose controlling expression is true is assigned. For example, the following circuit is a 4-to-3 priority encoder: (17)

(18)

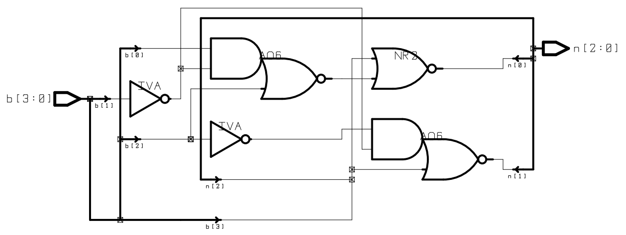
```
-- 4-to-3 encoder

library ieee ;
use ieee.std_logic_1164.all ;

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

(19)

```
architecture rtl of encoder 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 ;
```

and the resulting schematic is: (20)

21



Exercise: If we had used a selected assignment statement, how many lines would have been required in the selected assignment?

Sequential Circuits

So far we have only covered combinational logic circuits – circuits without memory. A `process` statement is used to describe sequential circuits. The `process` statement is a concurrent statement that contains *sequential* statements. Sequential statements execute one after the other as in conventional programming languages. The implications of this will be covered in detail later – for design purposes we only need to know how to use a process to generate flip-flops and registers. 22

23 As an example, we can describe a D flip-flop in VHDL as follows:

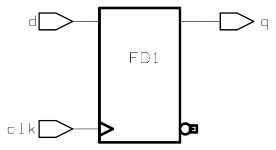
```
-- D Flip-Flop

library ieee ;
use ieee.std_logic_1164.all;

entity dff is
   port (
   d, clk : in std_logic ;
   q : out std_logic ) ;
end dff ;
```

24

```
architecture rtl of dff is
begin
process(clk)
begin
   if clk'event and clk = '1' then
      q <= d ;
   end if ;
end process ;
end rtl ;
```

The expression `clk'event` (pronounced "clock tick event") is true when the value of `clk` has changed since the last time the process was executed. The signal `q` is only assigned a value when `clk` changes and the new value is 1. Otherwise the output retains its previous value.

Not surprisingly, the result of synthesizing this description is a flip-flop (25) that is loaded on the rising edge of the clock input: (26)

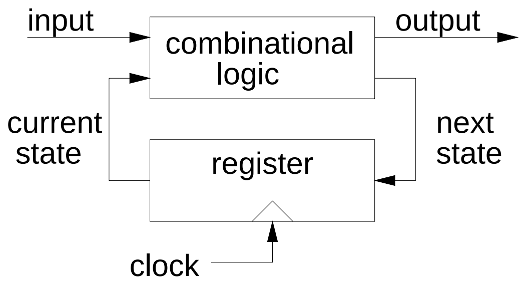


Exercise: What would we get if we replaced d and q with signals of type `std_logic_vector`?

State Machines

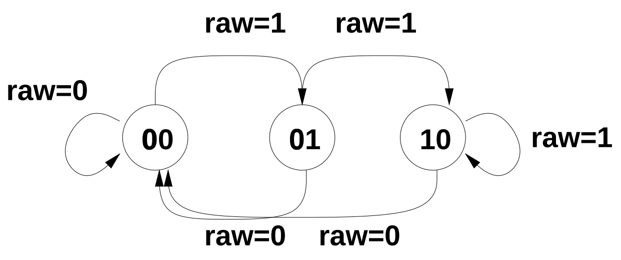
(27)

Sequential logic circuits are examples of state machines. Recall that a state machine computes its output and changes state based on its current state and its inputs: (28)



State machines in VHDL are implemented by using selected or conditional assignments to compute output and next state from the current state and the inputs. A process is used to implement the register that defines the current state.

For example, consider a VHDL description for a switch debouncer with the following state transition diagram: (29)



state	output
00	0
01	0
10	1

30 This can be described as follows:

```
-- Switch Debouncer

library ieee ;
use ieee.std_logic_1164.all ;

entity debounce is
   port (
      raw : in std_logic ;
      clean : out std_logic ;
      clk : in std_logic ) ;
end debounce ;
```

31

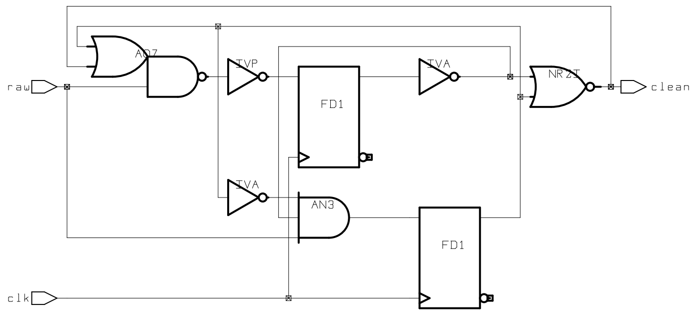
```
architecture rtl of debounce is
   signal currents, nexts :
       std_logic_vector (1 downto 0) ;
begin

   -- combinational logic for next state
   nexts <=
      "00" when raw = ’0’ else
      "01" when currents = "00" else
      "10" ;

   -- combinational logic for output
   clean <= ’1’ when currents = "10" else ’0’ ;
```

32

```
   -- sequential logic
   process(clk)
   begin
      if clk’event and clk = ’1’ then
         currents <= nexts ;
      end if ;
   end process ;

end rtl ;
```

The synthesized circuit is: 33

34



Exercise: Identify the components in the schematic that were created (“*instantiated*”) by different parts of the VHDL code.

Signed and Unsigned Types

We often want to treat vectors of logic values as binary values in two’s-complement or unsigned representations. By declaring signals to be of type `signed` or `unsigned` we can use the standard arithmetic (`+, -, *, /`) and comparison (`>, >, <=, >=,=, /=`) operators. However, only combinational logic will be generated and it may not be possible to synthesize multiplication or division. 35

We can use `unsigned` signals to build a state machine that implements a counter: 36

```
-- 3-bit Counter

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

entity counter is
   port (
      count_out : out unsigned (2 downto 0) ;
      clk : in std_logic ) ;
end counter ;
```

37

```
architecture rtl of counter is
   signal count, nextcount : unsigned (2 downto 0) ;
begin
   nextcount <= count + 1 ;
```

```
    process(clk)
    begin
       if clk'event and clk='1' then
          count <= nextcount ;
       end if ;
    end process ;

    count_out <= count ;
 end rtl ;
```

We used both `count` and `count_out` because the value of a signal of type `out` can't be "read" inside the architecture. 38

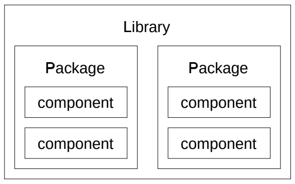
39 Exercise: Write the architecture for a 16-bit adder with two `signed` inputs, a and b and a `signed` output c.

Components, Packages and Libraries

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. 40

Re-use in VHDL is done by creating "components." A component declaration, like an entity declaration specifies the component's inputs and outputs.

Although components can be declared within an architecture, component declarations are usually saved in "packages". A package typically contains a set of components for a particular application. Packages are themselves stored in "libraries":



To save component declarations we put them within a `package` declaration. When this file is compiled ("analyzed") the information about the components in the package is saved in a file. 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 defines the input and output signals. Note that a component declaration does not create hardware – only when the component is used in an architecture will hardware be 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 compiled 41

- package name is `flipflops` - declares the `rs` component:

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

- compiling this file creates the package

To make the objects 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 most VHDL implementations a library is a directory and each package is a file in that directory. The package file is a database containing various things including 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 and is the library into which design entities are placed as they are compiled.

`library` and `use` statements must be used before *each* design unit (entity or architecture) that makes use of components found in those packages. For example, if you wanted to use the `DSP` package in the `ALTERA` library you would use: 42

```
library altera ;
use altera.dsp.all ;
```

Using Components

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 a *concurrent* statement like the `process` statement rather than a *sequential* statement and so a component instantiation cannot be put inside a `process`. 43

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 describes the `xor2` component (although a typical package defines more than one component): 44

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

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

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

45 the second file describes the `parity` entity that uses the `xor2` component:

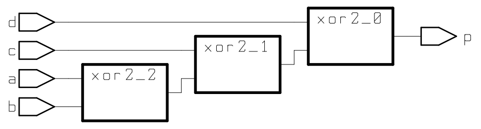
```
-- Parity function built from xor gates

library ieee ;
use ieee.std_logic_1164.all ;

use work.xor_pkg.all ;

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

46

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

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

48



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

Note that a component need not have been created in VHDL – it could be an ASIC standard-cell or FPGA configuration information.

Type Declarations

It's often useful to make up new types of signals for a project. We can do this in VHDL by using type declarations. 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). 49

The following example shows how a package called `dsp_types` that declares two new types is created: 50

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

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

Tri-State Buses

A tri-state output can be set to the normal high and low logic levels as well as to a high-impedance state. This type of output is often used where different devices must drive a bus at different times. One way to specify to the VHDL synthesizer that an output should be set to the high-impedance state is to use a signal of `std_logic` type and assign it a value of 'Z'. 51

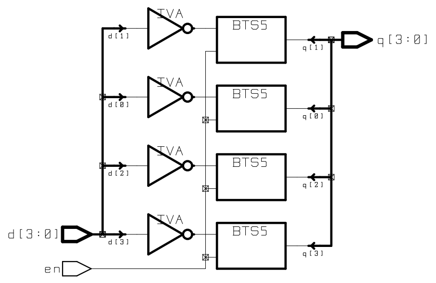
The following example shows an implementation of a 4-bit buffer with an enable output. When the enable is not asserted the output is in high-impedance mode : 52

```
-- Tri-State Buffer

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 ;
```

53

```
architecture rtl of tbuf is
begin
   q <=
      d when en = '1' else
      "ZZZZ" ;
end rtl ;
```

The resulting schematic for the `tbuf` is: 54



FPGA Configuration Demonstration

To demonstrate the use of VHDL we will design a "light chaser" by combining the counter and decoder described earlier. The `decoder` and `counter` entities are placed in a `democom` package (not shown) and instantiated in the architecture of the `demo` entity.

An internal signal and a type-conversion function are used to convert counter's `unsigned` output to the three `std_logic` inputs required by the decoder. A second internal signal is used to convert the active-high decoder output to the required active-low LED drive signals.

The FPGA demonstration board is configured so that the `clk` signal comes from a pushbutton and the `led` outputs drive the individual segments of a 7-segment LED display. 55

```
-- demonstration design light chaser

library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use work.democom.all ;

entity demo is
  port (
  clk : in std_logic ;
  led : out std_logic_vector (7 downto 0) ) ;
end demo;
```

56

```
architecture rtl of demo is
  signal count : unsigned (2 downto 0) ;
  signal scount : std_logic_vector (2 downto 0) ;
  signal ledN : std_logic_vector(7 downto 0) ;
begin
  c1: counter port map ( count, clk ) ;
  scount <= std_logic_vector(count);
  d1: decoder port map ( scount(2), scount(1), scount(0), ledN ) ;
  led <= not ledN ;
end rtl ;
```