# ENCS 533
# Lecture 2
# Introduction to VHDL

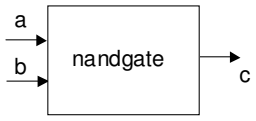
Instructor: Abdellatif Abu-Issa[1]

## 1 Introduction

In this lecture we will look at how to do simple designs in VHDL. In the first lab session you will get the opportunity to get some experience of writing and simulating your own descriptions.

## 2 A simple example

We'll start off with an extremely simple example: we will describe a NAND gate. The first thing that we have to do is to say what the device looks like to the outside world. This basically means describing its *port map*, i.e. the signals that flow in and out of it the device.




To describe this in VHDL, we use an *entity* declaration.

```
ENTITY nandgate IS
      PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC );
END;
```

Everything in uppercase is a VHDL keyword, i.e. part of the language. Everything in lower case is a name that I have chosen for the parts of my design. The entity has to be given a name (we've chosen nandgate, but you could have chosen any other name). Each of the signals in the port map is declared as having a *mode* and a *type*. The mode can be IN or OUT, and simply says whether the signal is an input or an output. The type STD_LOGIC represents a signal that bit can a value of '0', '1', 'X' or 'U'. ('X' means unknown. 'U' means uninitialized, i.e. a signal that has not yet been assigned any valid logical value.) STD_LOGIC is the normal way to describe logic signals that appear at the input or output of gates, or at wires in between them.

Now that we have described the inputs and outputs, we need to say what the device does, i.e. how its outputs respond to its inputs. This is done in an *architecture*:

```
ARCHITECTURE simple OF nandgate IS
BEGIN
      c <= a NAND b;
END;
```

[1] These lectures are prepared by Dr Steven Quigley, the Lecturer of Advanced Digital Design at the University of Birmingham - UK. After taking his permission, I used them in the ENCS 533 course at Birzeit University – Palestine. Dr Quigley is my supervisor in the PhD Thesis.

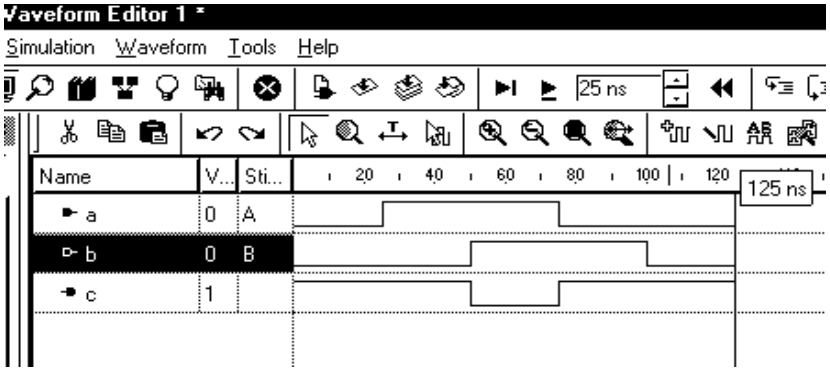
The ARCHITECTURE statement says that we are producing a description of what goes on inside *nandgate*. It is possible (and indeed quite common) for us to try out many different designs for what goes on inside *nandgate*. We have to give each different design a name, so that we can tell VHDL which version we want to use. I have chosen the name *simple* for this particular design. After the ARCHITECTURE statement comes the word BEGIN. This introduces the main body of the architecture, which explains how the outputs relate to the inputs. At the end of the body comes the END statement, which says that we have reached the end of the body.

How the outputs relate to the inputs is described by the statement

```
c <= a NAND b;
```

The symbol <= (which is meant to look like a left-pointing arrow) is pronounced "gets". It means that the signal c gets the value of a NANDed together with the value of b. Whenever a or b change their value, this statement causes the value of c to be updated.

If we want to check that our description is functioning correctly, we can feed it into a simulator, a program that predicts how the outputs would change in response to changes in the input. Here is the sort of thing we get if we run this code through a simulator




The horizontal axis is time, ranging from 0 to 125 ns. Traces are shown for the signals a, b and c. Whenever a or b changes its value, c receives a new value. In order to carry out the simulation, we need to tell the simulator what we want each of the inputs a and b to do (in this case we have toggled each from 0 1to 1 and then back to 0). The simulator then works out what the output c would do in response. You can see that c is carrying out the logic function a NAND b, so the design is correct.

VHDL knows the following logical connectives: NOT, AND, OR, NAND, NOR, XOR

### 2.1 Another architecture

As mentioned earlier, we are allowed to give many different descriptions of the way that the input related to the output. So, for example, here is a second version of the architecture of the *nandgate*.

```
ARCHITECTURE complicated OF nandgate IS
BEGIN
      c <= NOT ( a AND b );
END;
```

This achieves exactly the same function as the first description, but does it in a different way.

### 2.2 BEGIN and END statements

If you are used to C or Java, you will know that sometimes you want to consider a group of statements to be considered as one block. C and Java use curly braces { and } to indicated the beginning and end of a block respectively. So, for example, a loop in C or Java might look like this:

```
for (i=1; i<=n; i++)
{
    a[i]=i;
    b[i]=a[i]*a[i];
}
```

The braces show that the two statements should be considered collectively as a block that makes up the body of the loop.

VHDL uses the keywords BEGIN and END. So in VHDL the loop would look like this

```
FOR i IN ( 1 TO N ) LOOP
BEGIN
    a(i) = i;
    b(i) = a(i) * a(i);
END LOOP;
```

Note that indentation of the block is used to make it clearer where the block starts and ends.

### 2.3 Semicolons

Like C or Java, VHDL uses the semicolon to indicate the end of a statement.

Statements that "open up" a block don't take semicolons. So in C these would be wrong:

```
for (i=1; i<=n; i++); /* WRONG: shouldn't be a semicolon here */
{;                    /* ALSO WRONG: don't want a semicolon here */
    a[i]=i;
    b[i]=a[i]*a[i];
}
```

Similarly in VHDL these would be wrong

```
FOR i IN ( 1 TO N ) LOOP;  --WRONG: shouldn't be a semicolon here
BEGIN;                     --ALSO WRONG: don't want a semicolon here
    a(i) = i;
    b(i) = a(i) * a(i);
END LOOP;
```

Let's have another look at our simple example:

```
ENTITY nandgate IS
      PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC );
END;

ARCHITECTURE simple OF nandgate IS
BEGIN
      c <= a NAND b;
END;
```

The keyword IS is "opening up" a block of statements, and therefore does not need a semicolon. However, note that VHDL is a little inconsistent as to whether IS needs to be followed by a BEGIN. In an ENTITY, the BEGIN is implied, and the END statement is answering the IS. By contrast, in an ARCHITECTURE the word BEGIN must also be there, and the END is answering the BEGIN.

## 3 Stylistic issues

### 3.1 Case

VHDL is not case sensitive. All three of these are identical in meaning, and you'll see all three styles in textbooks and design magazines:

```
ENTITY nandgate IS
      PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END;

entity NANDGATE is
      port ( A, B: in std_logic; C: out std_logic);
end;

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

It used to be considered good style to write all the keywords of VHDL in one case, and all the names that we have chosen for our design in the other case. This makes it easier to figure out what is going on in the design.

Originally it was fashionable to use uppercase for all keywords of VHDL, as in the first listing above. Then the fashion changed to make all keywords lower case and all names chosen by the designer uppercase, as in the second listing.

Nowadays the fashion is to put everything in lowercase. Modern VHDL editors are context sensitive, and can figure out which words are part of the VHDL language and show them in a particular colour. So for the editors we use in labs, VHDL keywords are automatically displayed in blue, and the names chosen by us for signals, entities and architectures are shown in black.

In lectures we will show all keywords in uppercase to make it clearer to you what is part of the VHDL language, and what is just a name that I have chosen.

### 3.2 Spaces and indents

You can put as many spaces as you like between words. So, for example, these are both the same

```
ENTITY nandgate IS
PORT (a,b: IN STD_LOGIC; c: OUT STD_LOGIC);
END;

ENTITY nandgate IS
      PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END;
```

### 3.3 Returns

Putting in a carriage return makes no difference to the function of your code. So the following two are identical in function

```
ENTITY nandgate IS
      PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END;

ENTITY nandgate IS
      PORT ( a, b: IN STD_LOGIC;
                c: OUT STD_LOGIC);
END;
```

You can use whichever you feel is clearest.

### 3.4 Annotating END statements

In a long description, it can be easy to lose track of how many BEGIN and END pairs you have in the code. To help you keep track, you can put the name of what you think you are ending after the END statement. So, for example, you can write

```
ENTITY nandgate IS
      PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY nandgate;

ARCHITECTURE simple OF nandgate IS
BEGIN
      c <= a NAND b;
END ARCHITECTURE simple;
```

When you run the compiler, the code will be checked, and if there is a mismatch between what you say you are ENDing and what VHDL thinks you are ending, then this will be flagged as an error.

Although the annotation of END statements is normally optional, it is considered to be good style. It is a useful safety precaution, which can save you from bugs that are difficult and time consuming to find.

### 3.5 Comments

Comments are introduced by two dashes:

```
-- This is a comment
```

## 4 The IEEE library

The listings shown so far in this lecture have been incomplete, and if you try to use them, then the compiler will give an error message something like "Cannot recognise type STD_LOGIC". A large number of features and extensions to the capabilities of the VHDL language are bundled into a library called "IEEE". The definitions used for STD_LOGIC are held in this library. In order to use the features of this library, a design must open the library and say which features of the library it wishes to access.

### 4.1 Opening libraries

The IEEE library is opened by this statement:

```
LIBRARY IEEE;
```

The IEEE library contains many sub-libraries, which in turn contain many features. In order to say which features of which sub-libraries we wish to access, we use a statement that looks like this:

```
USE IEEE.XXXX.YYYY
```

Where XXXX is the name of the required sub-library, and YYYY is the name of the specific feature that is to be used. Rather than listing each specific feature that we want to use (which can be very tedious), often we will simply make all features within a sub-library visible by using the VHDL keyword ALL:

```
USE IEEE.XXXX.ALL
```

This opens up all features in the XXXX sub-library of the IEEE library so that they can be used by our design.

### 4.1 Using STD_LOGIC

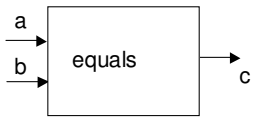
The standard logic definitions are held in a sub-library called std_logic_1164[2]. So here is a full listing for the NAND, that opens up the library to access the features of STD_LOGIC type.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY nandgate IS
      PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC );
END ENTITY nandgate;

ARCHITECTURE simple OF nandgate IS
BEGIN
      c <= a NAND b;
END ARCHITECTURE simple;
```

## 5 Conditionals

Sometimes we want to assign a signal in a way that is conditional on something else happening. Here is an example.

[2] 1164 is simply the number of the IEEE standards document that defined the Standard Logic type.



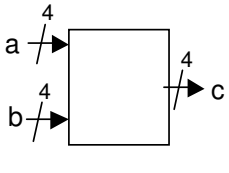

This device is called *equals*. It has two inputs *a* and *b* and one output *c*. If the two inputs are equal then the output is 1. If the two inputs are unequal, then the output is 0.

Here is a VHDL description of this device

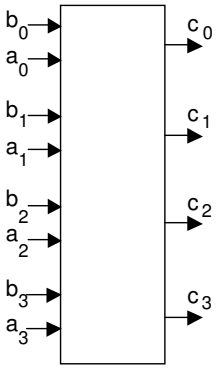
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY equals IS
      PORT ( a, b: IN STD_LOGIC;
             c: OUT STD_LOGIC);
END ENTITY equals;

ARCHITECTURE number1 OF equals IS
BEGIN
      c <= '1' WHEN a=b ELSE '0';
END ARCHITECTURE equals;
```

## 6 Handling signals that are more than 1 bit wide

Most interesting design have inputs that are more than just a single bit. For example, lets consider a device that has twp 4-bit inputs a and b, and a 4-bit output c.




If we expand this out, it looks like this




### 6.1 STD_LOGIC_VECTORs

In VHDL, quantities such as a, b and c are represented called STD_LOGIC_VECTORs. If you are familiar with arrays in computer programming,

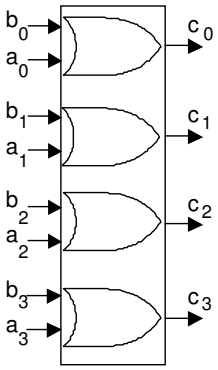
you can think of a STD_LOGIC_VECTOR as being an array of STD_LOGIC signals. So the input a would be declared as being

```
STD_LOGIC_VECTOR(0 TO 3)
```

Now a contains four members a(0), a(1), a(2) and a(3). Each of these four members is of type STD_LOGIC.

### 6.2 An example

Imagine that we wanted to represent a device like this:




The entity declaration would look like this.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY orgate IS
      PORT ( a, b: IN STD_LOGIC_VECTOR(0 TO 3);
               c: OUT STD_LOGIC_VECTOR(0 TO 3));
END ENTITY orgate;
```

There are several ways that we could write the architecture. One way to describe it would be like this, explicitly listing what happens for each bits:

```
ARCHITECTURE number1 OF orgate IS
BEGIN
      C(0) <= a(0) OR b(0);
      C(1) <= a(1) OR b(1);
      C(2) <= a(2) OR b(2);
      C(3) <= a(3) OR b(3);
END ARCHITECTURE number1;
```

Alternatively, we could just write this, which would be simpler and would mean exactly the same thing

```
ARCHITECTURE number2 OF orgate IS
BEGIN
      c <= a OR b;
END ARCHITECTURE number2;
```

VHDL knows that a, b and c are four bits wide, and will do the appropriate operation for each of the bit positions.

Or, if we preferred, we could write this

```
ARCHITECTURE number3 OF orgate IS
BEGIN
      C(0 TO 3) <= a(0 TO 3) OR b(0 TO 3);
END ARCHITECTURE number3;
```

This is effectively a loop, which tells VHDL to make four assignments, one for each of the four bit positions 0, 1, 2 and 3.

### 6.3 STD_LOGIC_VECTOR values

The value of an STD_LOGIC is indicated by a string of values enclosed in *double* quotes. So if a is a single bit, assignment looks like this:

```
a <= '1';
```

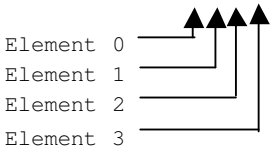
If a is 4-bits wide assignment looks like this:

```
a <= "1110";
```

### 6.4 Direction of numbering

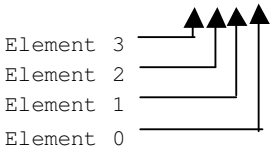
In the examples given above, the elements were numbered from 0 to 3

```
a: STD_LOGIC_VECTOR(0 TO 3)

a <= "1110";
```




This feels normal and intuitive (indeed in the programming languages that you may know, e.g. C or Java, this is the only way that you are allowed to do it). However, in VHDL you also have the option to have arrays where the index counts *downwards*:

```
a: STD_LOGIC_VECTOR(3 DOWNTO 0)

a <= "1110";
```

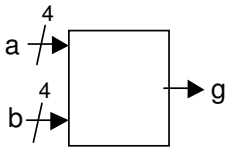


In digital logic design, the normal numbering convention is that bit 0 is the least significant bit (lsb). This is accomplished by having the index run *downwards*. The upward numbering scheme gets this wrong.

So unlike most programming languages, in VHDL *it is normal for arrays to be numbered downwards*. You can use upward-numbering if you want, but this often leads to confusion that creates awkward bugs in your code.

### 6.5 Arithmetic on STD_LOGIC_VECTORs

Now let's return to the comparator example thate we used in the last lecture




It has two inputs, a and b, both of which represent four-bit binary numbers. There is a single one-bit output g, which represents the "greater than" condition. When a>b then g='1'; otherwise g='0'.

But as we have already seen, this can't be interpreted unless we know whether a and b are signed (2's complement) or unsigned numbers. Suppose a=1111 and b=0001. If the numbers are unsigned then a is fifteen and b is one. So a>b and g=1. But if the numbers are signed then a is minus one and b is plus one; a<b and g='0'.

The way that VHDL handles this is by having two different versions of the arithmetic operators +,-,>,< etc. one for unsigned numbers and one for signed. The signed library is called STD_LOGIC_SIGNED. The unsigned is STD_LOGIC_UNSIGNED. We have to import one or the other library to tell VHDL how we want the standard logic vectors to be interpreted. So if we want the signed version it looks like this

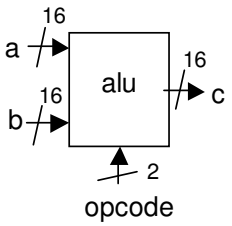
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY comp IS
      PORT ( a, b: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
             g: OUT STD_LOGIC);
END ENTITY comp;

ARCHITECTURE simple OF comp IS
BEGIN
      g <= '1' WHEN a>b ELSE '0';
END ARCHITECTURE simple;
```

If we wanted to use unsigned arithmetic, then line 3 would be changed to

```
USE ieee.std_logic_unsigned.ALL;
```

### 6.6 A more advanced example

All this may seem like a lot of hard work to produce designs that would have been easier using old fashioned methods. But now let's take on a more complicated design that really illustrates the power of VHDL.



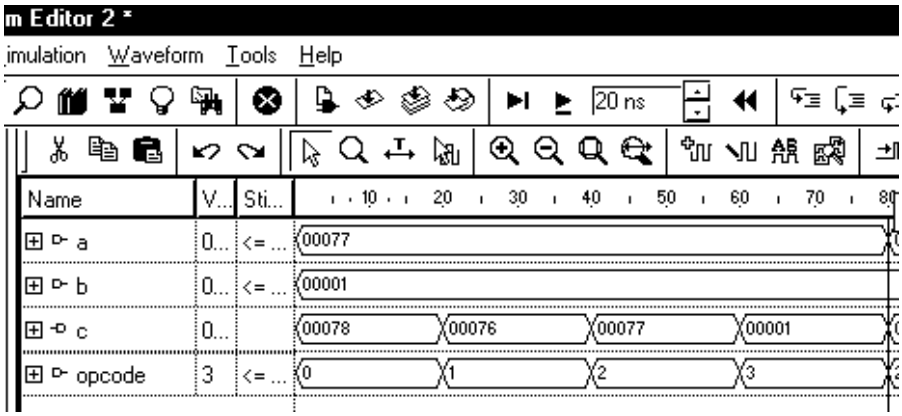

This is an arithmetic logic unit. It has two inputs, a and b, each of which are 16 bits wide. The 16-bit output c is produced by some arithmetic or logical operation on the two inputs. The operation that will be performed on a and b to produce c is defined by the opcode as follows:-

| Opcode | Operation |
|---|---|
| 00 | a + b |
| 01 | a - b |
| 10 | a or c |
| 11 | a and c |

To design one of these using traditional methods is a non-trivial task. However, in VHDL it's easy. Here is the listing:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;

ENTITY alu IS
      PORT ( a, b: IN STD_LOGIC_VECTOR(16 DOWNTO 0);
             opcode: IN STD_LOGIC_VECTOR(1 DOWNTO 0);
             c: OUT STD_LOGIC_VECTOR(16 DOWNTO 0) );
END ENTITY alu;

ARCHITECTURE simple OF alu IS
BEGIN
      c <= a + b WHEN opcode="00"
      ELSE a - b WHEN opcode="01"
      ELSE a OR b WHEN opcode="10"
      ELSE a AND b WHEN opcode="11";
END ARCHITECTURE simple;
```

In order to test whether our design does what we expect, we can feed it into a simulator tool. This is a program which allows us to apply inputs to the design, and to see what outputs would be produced by the design. Here is an example simulation:




The horizontal axis is time, ranging from 0 to 80 ns.
a has been given the value “0000000001110111”, which is 0077 in Hex.
b has been given the value “0000000000000001”, which is 0001 in Hex.
As the opcode changes through the sequence “00”,”01”,”10”,”11” (or 0,1,2,3 in Hex), the output c gets a+b, then a-b, then a OR b then a AND c.

Once we have simulated the description thoroughly and are sure that it correctly gives the behaviour we want, the code can then be fed into a synthesis tool, a computer program which will automatically generate a gate-level design.

## 7 Summary

We’ve looked at the basic features of VHDL, and seen some simple examples.

Once a VHDL description of a design has been produced, there are two things we can do it:

- Simulate it to see if the design really does what we want
- Synthesise it to automatically generate a hardware implementation.

**You should now know...**

The meaning of the following:

- Entity
- Port map
- Architecture
- Standard Logic (STD_LOGIC)
- The logic values ‘0’, ‘1’, ‘X’ and ‘U’
- Standard Logic Vector (STD_LOGIC_VECTOR)
- How to use a WHEN statement to carry out conditional assignment