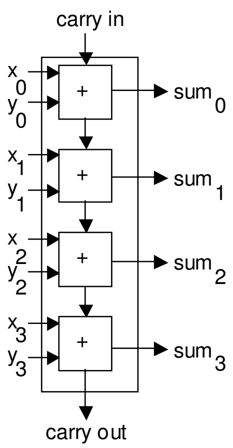
ENCS 533 - Advanced Digital Design
Lecture 3
Behavioural and Structural descriptions in VHDL

Introduction
In the last lecture we looked at a way of writing behavioural descriptions in VHDL, saying what we want the device to be able to do. This type of design can be simulated (feeding data to its inputs, to see what its outputs would do) in order to verify that the description has the required behaviour. Once we are sure that the description is correct, then it can be synthesised, i.e. run through a program that will automatically generate hardware that fulfils the description.

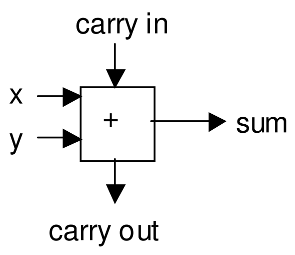
It is also possible to use VHDL to write structural descriptions, saying how we would connect together basic units (e.g. logic gates) to build our design. (The output of a synthesis tool, which builds a gate level netlist to implement our design, will often be of this form.)

In this lecture, we will look a little deeper at behavioural descriptions, and then move on to look at how to write structural descriptions.

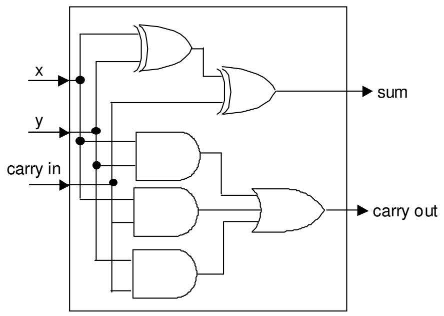
An example
Throughout this lecture we will be looking at the four-bit adder example introduced in lecture 1.



The four-bit adder is built up from four 1-bit full adders, which have the following behaviour:

	x	y	Carry in	Sum	Carry out
	0	0	0	0	0
	0	0	1	1	0
	0	1	0	1	0
	0	1	1	0	1
	1	0	0	1	0
	1	0	1	0	1
	1	1	0	0	1
	1	1	1	1	1

There are many ways to implement this. One possible way is shown below.



Behavioural description

Suppose we want to write a behavioural description of the full adder. In order to do this, we must explain how we want the outputs of the design to relate to the inputs. One way is simply to use the truth table to describe the device, like this:

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

ENTITY fulladd IS
    PORT ( x, y, cin: IN STD_LOGIC;
             sum, cout: OUT STD_LOGIC);
END ENTITY fulladd;

ARCHITECTURE tedious_but_easy OF fulladd IS
BEGIN
      sum <= '0' WHEN x='0' AND y='0' AND cin='0'
        ELSE '1' WHEN x='0' AND y='0' AND cin='1'
        ELSE '1' WHEN x='0' AND y='1' AND cin='0'
        ELSE '0' WHEN x='0' AND y='1' AND cin='1'
        ELSE '1' WHEN x='1' AND y='0' AND cin='0'
        ELSE '0' WHEN x='1' AND y='0' AND cin='1'
        ELSE '0' WHEN x='1' AND y='1' AND cin='0'
        ELSE '1' WHEN x='1' AND y='1' AND cin='1';
      cout <= '0' WHEN x='0' AND y='0' AND cin='0'
        ELSE '0' WHEN x='0' AND y='0' AND cin='1'
        ELSE '0' WHEN x='0' AND y='1' AND cin='0'
        ELSE '1' WHEN x='0' AND y='1' AND cin='1'
        ELSE '0' WHEN x='1' AND y='0' AND cin='0'
        ELSE '1' WHEN x='1' AND y='0' AND cin='1'
        ELSE '1' WHEN x='1' AND y='1' AND cin='0'
        ELSE '1' WHEN x='1' AND y='1' AND cin='1';
END tedious_but_easy;
```

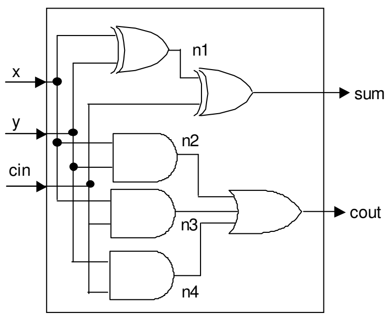
This is easy and obvious, but also tedious. There are many neater ways to describe the behaviour. We could take inspiration from the gate level design of the full adder, and write this

```
ARCHITECTURE simple OF fulladd IS
BEGIN
      sum <= cin XOR x XOR y;
      cout <= ( x AND y ) OR ( cin AND x ) OR ( y AND cin );
END ARCHITECTURE simple;
```

This is much neater and nicer, but requires us to think a bit harder about how the outputs relate to the inputs.

It's important to realise that as far as a synthesis tool is concerned, both descriptions are the same thing. They simply say how the outputs relate to the inputs. The second architecture is *not* ordering the synthesis tool to use two XOR gates, 3 AND gates and an OR gate. It's simply a shorthand for saying how the output relates to the inputs. The synthesis tool is free to do whatever it wants to find a circuit that has the same input-output relation.

Local signals
Now let's look at a slight modification of our description.

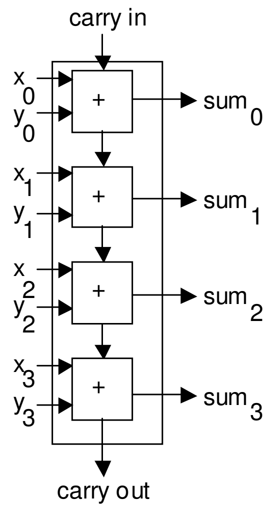


We have given names to the internal nodes of the circuit (n1, n2, n3, n4). Once we have given them names, we are free to use them in our description. So here is a slightly different description

```
ARCHITECTURE number3 OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
BEGIN
      n1 <= x XOR y;
      sum <= cin XOR n1;
      n2 <= x AND y;
      n3 <= cin AND x;
      n4 <= y AND cin;
      cout <= n2 OR n3 OR n4;
END ARCHITECTURE number3;
```

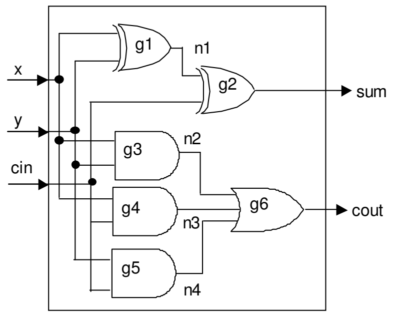
This is basically the same as the *simple* architecture of *fulladd*, but this time we have used the local signals n1, n2, n3 and n4 as part of the description. In order to use the names, we have to *declare* that they exist, that they are *signals*, and that they carry logic values (e.g. '1', '0', 'X' and 'U') which means that they are of type *STD_LOGIC*. The declaration of local signals takes place between the ARCHITECTURE statement and the first BEGIN.

The reason why they are called *local* signals is that they are "hidden" inside the full adder. Imagine that we place our full adder in a bigger system (e.g. the four bit adder):



The full adder's inputs and outputs (i.e. the signals in its port map) are visible to other devices in the system. However, the local signals n1, n2, n3, n4 are buried inside the full adder and cannot be accessed by other devices in the system.

How are statements processed?
Let's look again at the full adder circuit, and for the sake of clarity, we will now give names (g1…g6) to the gates.



Imagine that the signals x, y and cin are initially at zero. Looking through the circuit, we can see that n1, n2, n3, n4, sum and cout will all be at zero.

Now imagine that x changes its value from 0 to 1. Let's think through what happens next:

- x is the input to three gates: g1, g3 and g4. These gates are potentially affected by the change, so we need to re-compute their outputs n1, n2, n3.
- We also know that gates g2, g5 and g6, which don't have x as an input, can't be affected by this change, so there is no point to re-computing their outputs.
- The new value of n1 is 1 (i.e. it changed)
- The new value of n2 is 0 (i.e. it is unchanged)
- The new value of n3 is 0 (i.e. it is unchanged)
- n1 just changed, which means that any gate that has n1 as an input (i.e. g2) needs to have its output (sum) re-computed.
- n2 and n3 didn't change, so we don't need to bother to examine any consequences in gate g6, which has n2 and n3 as inputs.

- The new value of sum is 1.
- There are no more gates whose inputs have changed, so we can stop analysing the circuit now.

The way that VHDL processes statements during simulation tries to capture the above thought process.

```
      ARCHITECTURE number3 OF fulladd IS
          SIGNAL n1, n2, n3, n4: STD_LOGIC;
      BEGIN
1             n1 <= x XOR y;
2             sum <= cin XOR n1;
3             n2 <= x AND y;
4             n3 <= cin AND x;
5             n4 <= y AND cin;
6             cout <= n2 OR n3 OR n4;
      END ARCHITECTURE number3;
```

- All statements 1-6 are scanned simultaneously, waiting for a signal on the right hand side (RHS) to change. In the jargon of VHDL, a change to a signal is called an *event.*
- When x changes from 0 to 1 (in the jargon of VHDL *there is an event on x*), statements 1, 3 and 4 are triggered and run.
- Statement 1 computes a new value of n1. It changes from 0 to 1. There is an event on n1.
- Statement 3 re-computes n2, but it still computes to 0. This is not a change, so there is no event on n2.
- Similarly statement 4 computes a new value of n3. It is 0, which is not a change.
- The consequences of the event on x have been full worked through (all statements where it appears on the RHS have been re-evaluated). So now we look for other events that need to be processed. There has been an event on n1.
- Statement 2 has n1 on its RHS, so it is triggered into life and executes
- A new value is computed for *sum.* It becomes 1, and an event has occurred on *sum.*
- Any statement with *sum* on its RHS would triggered, but no statement has sum on its RHS.
- There are no pending events, so there is nothing left to do. Finish processing.

Concurrent processing
Now we come to a very important point. Consider these two descriptions of the full adder:

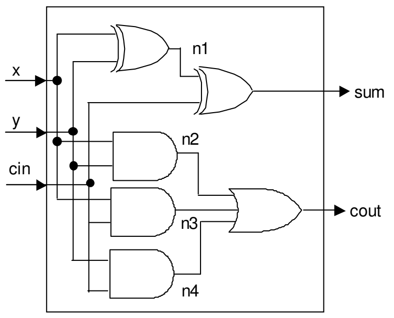
```
ARCHITECTURE number3 OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
BEGIN
      n1 <= x XOR y;
      sum <= cin XOR n1;
      n2 <= x AND y;
      n3 <= cin AND x;
      n4 <= y AND cin;
      cout <= n2 OR n3 OR n4;
END ARCHITECTURE number3;
```

```
ARCHITECTURE number4 OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
BEGIN
      sum <= cin XOR n1;
      cout <= n2 OR n3 OR n4;
      n1 <= x XOR y;
      n2 <= x AND y;
      n3 <= cin AND x;
      n4 <= y AND cin;
END ARCHITECTURE number4;
```

Although they are written in a different order, *they do exactly the same thing*. Unlike programming languages such as C, which process lines in the order that they are written, VHDL normally monitors all statements at the same time, and executes a statement when one of its RHS values changes. This is called *concurrent* execution.

Structural Description

Earlier on, we looked at this behavioural description



```
ARCHITECTURE simple OF fulladd IS
BEGIN
      sum <= cin XOR x XOR y;
      cout <= ( x AND y ) OR ( cin AND x ) OR ( y AND cin );
END ARCHITECTURE simple;
```

We said that this is *not* ordering the synthesis tool to use two XOR gates, 3 AND gates and an OR gate. It's simply a shorthand for saying how the output relates to the inputs.

It is also possible to write VHDL that *does* explicitly construct the full adder from two XOR gates, 3 AND gates and an OR gate. This would be a *structural* description (since it is saying how we want to build up our design out of simpler components). In order to write a structural description, we first of all need to create a library of gates that we can use as building blocks. Here are descriptions of the *and* gate, the *xor* gate and the 3-input *or* gate.

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

ENTITY and2 is
    PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY and2;
```

```
ARCHITECTURE simple OF and2 IS
BEGIN
    C <= a AND b;
END ARCHITECTURE simple;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY or3 is
    PORT ( a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
END ENTITY or3;

ARCHITECTURE simple OF or3 IS
BEGIN
    d <= a OR b OR c;
END ARCHITECTURE simple;

LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY xor2 is
    PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY xor2;

ARCHITECTURE simple OF xor2 IS
BEGIN
    C <= a XOR b;
END ARCHITECTURE simple;
```

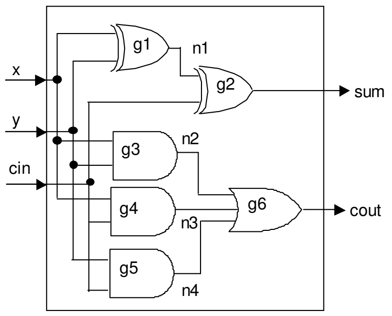
Note that I have chosen the names *and2*, *or3* and *xor2* for the 2-input and 3-input or and 2-input xor gates. I could have chosen *almost* any name I liked, but I could not have called the AND gate "AND" because this is a protected keyword of the VHDL language. "AND" (and other words like BEGIN, END, ENTITY, etc.) are part of the VHDL language, and have a special meaning. VHDL does not allow you to use any of its keywords to use as names for your designs.

The work library
When you compile your designs they are placed into a library ready to be used by other designs. By default, the current working library is called *work.*

The gate designs will be store in the library with the names
- work.and2(simple)
- work.or3(simple)
- work.xor2(simple)

The name is constructed from the library name followed by a point, then the entity name, then the architecture name.

Structural descriptions
Now that we have a library of gates available to us to use in our design, let's look at a structural description of the full adder.



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

ENTITY fulladd IS
    PORT ( x, y, cin: IN STD_LOGIC;
             sum, cout: OUT STD_LOGIC);
END ENTITY fulladd;

ARCHITECTURE structural OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
BEGIN
    g1:  ENTITY work.xor2(simple) PORT MAP (x,y,n1);
    g2:  ENTITY work.xor2(simple) PORT MAP (n1,cin,sum);
    g3:  ENTITY work.and2(simple) PORT MAP (x,y,n2);
    g4:  ENTITY work.and2(simple) PORT MAP (x,cin,n3);
    g5:  ENTITY work.and2(simple) PORT MAP (y,cin,n4);
    g6:  ENTITY work.or3(simple) PORT MAP (n2,n3,n4,cout);
END ARCHITECTURE structural;
```

Each of the gates is defined by a statement providing

- a name for the gate[1] (I've chosen g1 … g6, but I could have chosen any name I like.)
- the keyword ENTITY
- the full name of the gate that I want to use
- the keyword PORT MAP
- a list of the wires that I am connecting to the inputs and outputs of the gate

In the jargon of VHDL, each of these statements is called an *instantiation.* I have created two *instances* of the XOR gate, three *instances* of the AND gate and one *instance* of the OR gate.

Positional association

How does VHDL know which of the wires I am connecting to g1 are inputs and which are outputs? If we compare the instantiation

```
    g1:  ENTITY work.xor2(simple) PORT MAP (x,y,n1);
```

with the definition of the and gate

```
ENTITY and2 is
    PORT ( a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
END ENTITY and2;
```

[1] Strictly speaking, *g1* is a statement label, but you can think of it as just providing a name for the gate.

We see that the first two signals in the port map are inputs and the third is the output. So the first two signals in the instantiation x and y will be attached to the inputs a and b, and the third n1 will be attached to the output c. This is called *positional association.*

Named association
If you prefer, you can explicitly tell VHDL how you want to connect up the wires in your design to the inputs and outputs of the gate, like this

```
    g1:  ENTITY work.xor2(simple) PORT MAP ( a=>x, b=>y, c=>n1 );
```

This is called *named association*. With named association, the order doesn't matter, so you could write the instantiation like this

```
    g1:  ENTITY work.xor2(simple) PORT MAP (c=>n1, b=>y, a=>x );
```

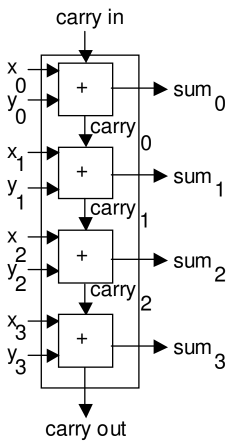
How the statements are processed
For the sake of clarity, let's look again at the structural description, and highlight which of the signals are inputs to which gate.

```
ARCHITECTURE structural OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
BEGIN
    g1:  ENTITY work.xor2(simple) PORT MAP (x,y,n1);
    g2:  ENTITY work.xor2(simple) PORT MAP (n1,cin,sum);
    g3:  ENTITY work.and2(simple) PORT MAP (x,y,n2);
    g4:  ENTITY work.and2(simple) PORT MAP (x,cin,n3);
    g5:  ENTITY work.and2(simple) PORT MAP (y,cin,n4);
    g6:  ENTITY work.or3(simple) PORT MAP (n2,n3,n4,cout);
END ARCHITECTURE structural;
```

Again, suppose that the signals x, y and cin are initially at zero, so n1, n2, n3, n4, sum and cout are also at zero. Now suppose that x changes from 0 to 1.

- All statements g1-g6 are scanned simultaneously, waiting for an event on an input signal.
- There is an event on x, so g1, g3 and g5 re-evaluate their outputs.
- Statement g1 computes a new value of n1. It changes from 0 to 1. There is an event on n1.
- Statement g3 re-computes n2, but it still computes to 0. This is not a change, so there is no event on n2.
- Similarly statement g4 computes a new value of n3. It is 0, which is not a change.
- The event on n1 causes g2 to re-evaluate its output
- A new value is computed for *sum*. It becomes 1, and an event has occurred on *sum*.
- *sum* is not an input to any instantiation, and there are no pending events, so there is nothing left to do. Finish processing.

The description of the 4-bit adder
Finally, we can write a structural description of the four bit adder by building it from four full adders.



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

ENTITY adder IS
    PORT ( x, y: IN STD_LOGIC_VECTOR(3 DOWNTO 0);
             cin:  IN STD_LOGIC;
             sum: OUT STD_LOGIC_VECTOR(3 DOWNTO 0);
               cout: OUT STD_LOGIC);
END ENTITY adder;

ARCHITECTURE structural OF adder IS
    SIGNAL carry: STD_LOGIC_VECTOR(2 DOWNTO 0);
BEGIN
    g0:  entity work.fulladd(structural)
               PORT MAP (x(0),y(0),cin,sum(0),carry(0));
    g1:  entity work.fulladd(structural)
               PORT MAP (x(1),y(1),carry(0),sum(1),carry(1));
    g2:  entity work.fulladd(structural)
               PORT MAP (x(2),y(2),carry(1),sum(2),carry(2));
    g3:  entity work.fulladd(structural)
               PORT MAP (x(3),y(3),carry(2),sum(3),cout);
END ARCHITECTURE structural;
```

Looking at this code, you should be able to guess that there is a more efficient way of writing it, using a kind of loop. Constructing loops that define arrays of hardware is beyond the scope of this module, but if you want to find out more, look up the FOR and GENERATE statements in the Active HDL online help.

Summary
In this lecture we have looked at how behavioural and structural descriptions are processed using concurrent execution. We have also seen how to build up structural descriptions from instantiation statements.

You should now know...

The meaning of the following:
- *Local signals*
- *Event*
- *Concurrent execution*
- *Protected keyword*
- The *work* library
- *Instantiation* and *Instance*
- *Positional association* and *Named association*