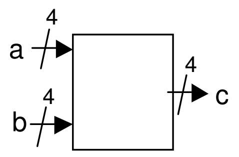
ENCS 533
Advanced Digital Design
Lecture 5
Iterative Designs

1 Introduction

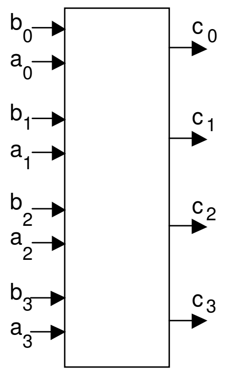
In lecture 3, we looked at how to do *structural* designs, in other words how to connect together basic gates to build up a design. In practice, these structural designs are not normally created by humans. Instead, humans write behavioural descriptions (which say what the design must do, not how it is built). In this lecture, we will introduce some examples of behavioural designs, but first we will need to look at how to deal with signals that are many bits wide, and hardware that consists of multiple copies of basic units.

2 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



2.1 STD_LOGIC_VECTORs

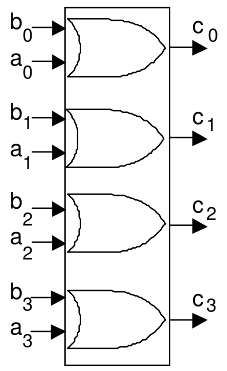
In VHDL, quantities such as a, b and c are called STD_LOGIC_VECTORs. If you are familiar with arrays in computer programming, you can think of a STD_LOGIC_VECTOR as being an array of STD_LOGIC signals. So the input a would be declared as being

```
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.

2.2 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.

2.3 GENERATE loops

We can also explicitly write a loop, using a GENERATE block.

```
ARCHITECTURE number4 OF orgate IS
BEGIN
      gen1: FOR i IN 0 TO 3 GENERATE
            BEGIN
                c(i) <= a(i) OR b(i);
            END GENERATE;
END ARCHITECTURE number4;
```

The value i loops from 0 to 3, giving us the required behaviour.

There are three slightly odd points compared to other languages that you may have come across:

- We don't need to declare i. VHDL can tell from the 0 TO 3 range that I has got to be an integer
- I is local to the loop. Outside the loop its value is undefined, and it can't be used.
- We have to give the loop a name[1]. For this example I've chosen the name *gen1*.

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";
```

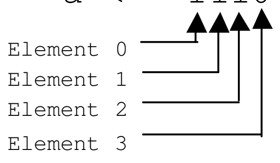
By default, VHDL expects the values to be binary, but sometimes it can be useful to use Hex numbers. This can be done by placing the letter X before the STD_LOGIC_VECTOR value:

```
a <= X"E";
```

3.1 Direction of numbering

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

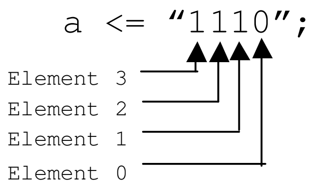
```
a: STD_LOGIC_VECTOR(0 TO 3)

a <= "1110";
```



[1] Strictly speaking, this is a statement label.

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)
```



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.

3.2 Aggregates

Aggregates are a group values, separated by commas, that will be used for an array. Here is an example:

```
ARCHITECTURE example OF aggregate IS
    SIGNAL nibble1, nibble2: STD_LOGIC_VECTOR ( 0 TO 3 );
BEGIN
    nibble1 <= ( '0','1','0','0');
    nibble2 <= ( '0','0','1','0');
END ARCHITECTURE example;
```

The assignment for nibble1 sets its 0th value to '0', its 1st value to '1', the 2nd to '0' and so on. This way of doing things is called *positional* assignment: the 0th value listed goes in the 0th position, the first goes in the first position and so on. We could instead use named association. So these statements have the same effect:

```
    nibble1 <= ( '0','1','0','0');
    nibble1 <= ( 1 => '1', 0 => '0', 3 => '0', 2 => '0');
```

With named association, we can just specify the values of some of the bit positions, and use an OTHERS value to provide a value for everything not explicitly mentioned. So these are all the same:

```
    nibble1 <= ( '0','1','0','0');
    nibble1 <= ( 1 => '1', 0 => '0', 3 => '0', 2 => '0');
    nibble1 <= ( 1 => '1', OTHERS => '0');
```

The OTHERS notation can be used as a convenient trick when we want to set all the values of an array to a particular value:

```
    nibble1 <= ( OTHERS => '1');
```

This would set all of the elements of nibble1 to '1'.

3.3 Concatenation

Concatenation merges two vectors to produce a longer vector. For example

```
ARCHITECTURE example OF aggregate IS
    SIGNAL byte: STD_LOGIC_VECTOR ( 0 TO 7 );
    SIGNAL nibble1, nibble2: STD_LOGIC_VECTOR ( 0 TO 3 );
BEGIN
    nibble1 <= ( '0','1','0','0');
    nibble2 <= ( '0','0','1','0');
    byte <= nibble1 & nibble2;
END ARCHITECTURE example;
```

would cause byte to assume the value ('0','1','0','0','0','0','1','0')

3.4 Literals

STD_LOGIC is a data type that has values '0', '1', 'X', 'U' etc. It is sub-type of CHARACTER. An array of characters is a *string*, and is denoted by double quotes. This is the same convention as used in the C programming language: '1' is a character; "1010" is an array of 4 characters [2]. We can use this notation for STD_LOGIC_VECTORS. So for example,

```
    nibble1 <= ( '0','1','0','0');
```

could be written as

```
    nibble1 <= "0100";
```

A value that is directly specified (as opposed to being calculated from other signals), like "0100" in the code above is called a *literal*. Standard logic vector literals may be specified in binary, octal or hexadecimal. By default, a string is interpreted as binary. To make it explicit that we wish the string to be interpreted as a binary number, we can place the letter B in front. For an octal string, we place the letter O in front, and for hexadecimal, we place X in front. So if a is 12-bit std_logic_vector, then these are all equivalent:

```
    a <= "010011001010";
    a <= B"010011001010";
    a <= O"2312";
    a <= X"4CA"
```

Long strings of '1's and '0's can be confusing, so in order to improve legibility, we can introduce underscores:

```
    a <= B"0100_1100_1010";
```

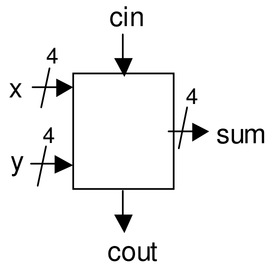
The underscores are ignored by VHDL; their only function is to space the digits out to make it easier for a human to read. Note that if you do use underscores in your values, you must put the B in front to make it clear that this should be interpreted as a binary value. This (without the B) would be an error:

```
    a <= "0100_1100_1010";    -- Wrong!
```

[2]Actually, it isn't quite identical to C. "1010" in C is an array that is terminated by a \0 character. The VHDL string doesn't have a terminator.

4 What a synthesis tool does

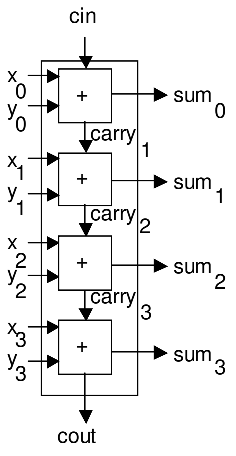
Imagine that we want to create a 4-bit adder, with carry in and carry out:



A human would normally describe this *behaviourally*:

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.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 behavioural OF adder IS
    SIGNAL carry: STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
    sum <= x + y;
END ARCHITECTURE behavioural;
```

This would then be given to a synthesis tool, a computer program whose purpose is to design a circuit that fulfils the required behaviour. One possible circuit would be this:



This is a carry-ripple adder, built from four identical full adders. We have already designed a full adder in lecture 3, so we can use this to build up a structural description of a circuit that achieves the required behaviour:

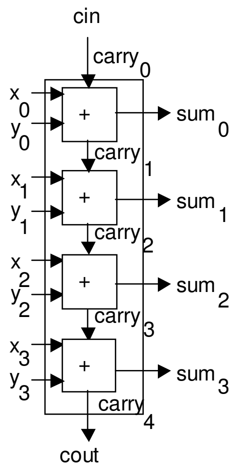
```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

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

So how did the synthesis tool know that an adder could be built like this? There are some things that a synthesis tool is smart enough to figure out for itself, and we'll see some examples later in the course. However, for devices like adders, comparators, multipliers, etc. it would be usual for the synthesis tool to be equipped with a library that had been written by an expert human being. The synthesis tool simply looks up in its library how it should build a circuit that implements the add behaviour. In lecture 5 we will return to some of the issues that are involved in writing good libraries.

4.1 Iterative Description of the 4-bit adder

Looking at our structural description of the 4-bit adder, it is clear that it has a very regular structure. Hardware that consists of the same unit repeated over and over again is called *iterative*. It is tempting to try to express this as some kind of loop, capturing the iterative nature of the design. Before we do this, we make one slight adjustment to make the design as regular as we possibly can.



Here we have extended the labelling of the carry chain to include carry$_0$ (which is connected to cin) and carry$_4$ (which is connected to cout). The adder description now looks like this:

```
ARCHITECTURE regularised OF adder IS
    SIGNAL carry: STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
    carry(0) <= cin;
    cout <= carry(4);
    g0:  entity work.fulladd(structural)
                PORT MAP (x(0),y(0),carry(0),sum(0),carry(1));
    g1:  entity work.fulladd(structural)
                PORT MAP (x(1),y(1),carry(1),sum(1),carry(2));
    g2:  entity work.fulladd(structural)
                PORT MAP (x(2),y(2),carry(2),sum(2),carry(3));
    g3:  entity work.fulladd(structural)
                PORT MAP (x(3),y(3),carry(3),sum(3),carry(4));
END ARCHITECTURE regularised;
```

Now each of the instantiations is identical, except that the numbers increase by 1 at each instance. This can be expressed as a GENERATE loop as follows:

```
ARCHITECTURE iterative OF adder IS
    SIGNAL carry: STD_LOGIC_VECTOR(4 DOWNTO 0);
BEGIN
    carry(0) <= cin;
    cout <= carry(4);
    gen1: FOR i IN 0 TO 3 GENERATE
            g:  ENTITY work.fulladd(structural)
                PORT MAP (x(i),y(i),carry(i),sum(i),carry(i+1));
          END GENERATE gen1;
END ARCHITECTURE iterative;
```

Remember that we have to give each GENERATE loop a name (in this case I've chosen *gen1*) and also each instantiation must have a name (in this case *g*).

5 Representation of logic levels

5.1 Type BIT: just 1's and 0's

The built-in logic used by the original VHDL has only two values: '1' and '0'. This isn't very useful, and normally this logic needs to be extended. The type "BIT" has been superseded by the IEEE 1164 standard logic.

5.2 IEEE 1164 standard logic

Because it is so common for users to require a more powerful and expressive type than BIT, there is a standard enhanced logic defined by the IEEE and implemented in all modern VHDL tools. The name of the standard is IEEE 1164, and it defines two principle types: std_logic and std_ulogic.

```
TYPE std_logic IS     (     'U',     --Unitialized
                            'X',     --Driven unknown
                            '0',     --Driven 0
                            '1',     --Driven 1
                            'Z',     --High impedance
                            'W',     --Weak unknown
                            'L',     --Weak 0
                            'H',     --Weak 1
                            '-'      --Don't care
                            );
```

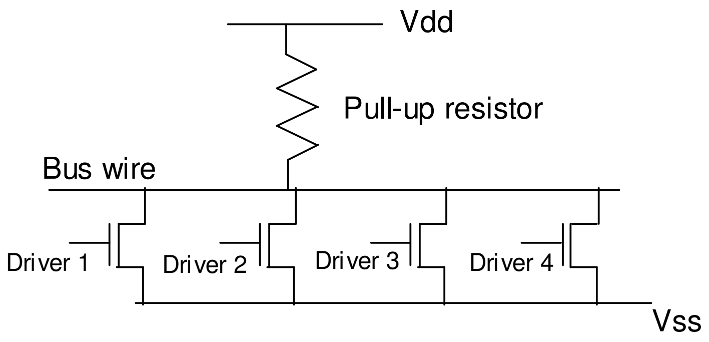
This isn't actually built in to the VHDL language, but appears in a standard add-on library that is shipped with most versions of VHDL. Type std_logic is *much* better than type BIT and in your designs you should *always* use std_logic in preference to BIT.

In order to use this logic, and all associated declarations and basic gate types, it is necessary to include the following at the head of your code:

```
    LIBRARY IEEE;
    USE IEEE.std_logic_1164.all;
```

5.2.1 Logic strength

Note that IEEE 1164 logic uses two different logic strengths, *driven* and *weak*. These exist in order to enable VHDL to model this sort of situation:



Here we have a wire on a bus, with four different circuits that might drive the wire down to a logic zero by asserting a 1 on one of the driver lines. When none of the driver lines is asserted, then the pull-up resistor will drive the bus wire to a logic one. The description might look like something like this:

```
buswire <= '1';
buswire <= '0' when driver1='1' or driver2='1'
                 or driver3='1' or driver4='1';
```

The difficulty with modelling this situation is that when one of the drive circuits tries to pull down the bus, the pull-up resistor is trying to write a 1 to a bus wire at the same time as the pull-down transistor is trying to write a 0. How does VHDL know which will win?

This is resolved by giving signals a strength. The transistors have a low resistance when switched on, and are therefore writing a strong 0. Strong signals are often referred to as "driven". The resistor has a high resistance and is therefore writing a weak 1. Weak signals are often referred to as "resistive". So the description now becomes:

```
buswire <= 'H';
buswire <= '0' when driver1='1' or driver2='1' or driver3='1' or
driver4='1';
```

Now VHDL can tell that when there are two sources trying to write simultaneously to the wire, that the 0 will win[3].

5.2.2 Resolved logic

There is actually an even more fundamental problem with this:

```
buswire <= '1';
buswire <= '0' when driver1='1' or driver2='1'
                 or driver3='1' or driver4='1';
```

In VHDL, it is normally *illegal* for two different statements to be *concurrently* writing to the same signal. If you declared *buswire* as being of type BIT, and tried to compile the above piece of code, the compiler would report an error (something like *ERROR: multiple drivers for unresolved signal*).

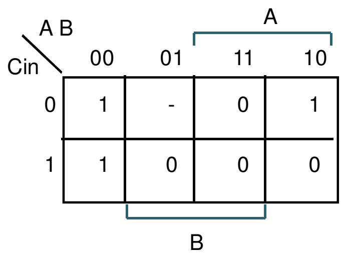
It is possible in VHDL to have *resolved* logic. This is a type of logic that knows how to handle a situation where two statements are concurrently writing to one signal. Std_logic *is* a resolved logic, so it *can* handle signals with multiple drivers. BIT is not.

There is also a version of IEEE 1164 that is *unresolved*, called std_ulogic (standard unresolved logic).

5.2.3 Don't care, don't know and uninitialised

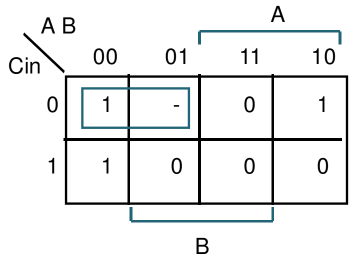
In most text books on digital logic, don't care and don't know are both represented by X. It is useful to be able to distinguish between the two.

Don't know normally occurs in simulation, and is often a sign that something has gone wrong. *Don't care* usually occurs in synthesis, and is usually quite useful. It means that we are liberty to teat the value as a 0 or a 1, whichever is most convenient. So for example, this



Cin \ AB	00	01	11	10
0	1	-	0	1
1	1	0	0	0

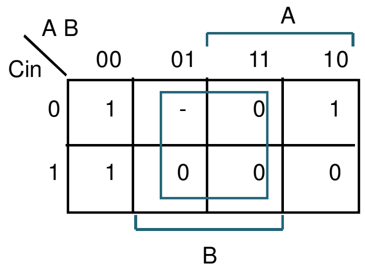
Could be factored as this (treating the *don't care* as a 1)



Cin \ AB	00	01	11	10
0	1	-	0	1
1	1	0	0	0

[3] This example is given by way of illustration. If you actually want to get this working correctly, there are a few other subtleties that you will need to address, but these are beyond the scope of this lecture.

or this (treating the *don’t care* as a 0)

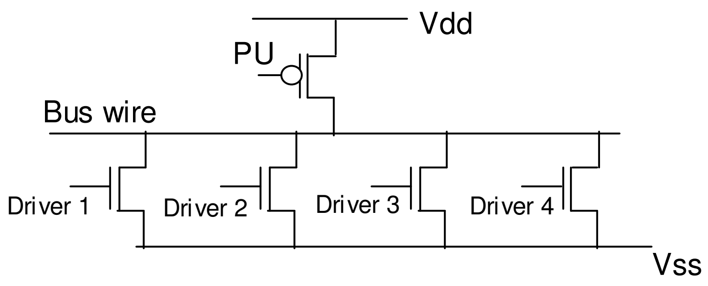


whichever is most convenient.

Std_logic allows us to distinguish between don’t know (X or W) and don’t care (-). There is also a designation for the uninitialised state (U). This is particularly important for state machines, where the state machines will contain garbage at power-on. Any logic gate with a U on one of its inputs will output a U; any Boolean expression which has a U on its RHS will have its LHS assigned to U. IN this way we can identify signals that contain garbage as a result of being fed by input registers that were not reset at power up.

5.2.4 High impedance

In some instances, notable tri-state buffers, we can have outputs that can be completely disconnected from any driving source. This happens in the following circuit when all the transistors are switched off.



The value on the bus wire is modelled as the high impedance state Z. This could be described as

```
buswire <= ‘1’ WHEN PU=’0’
         ELSE ‘0’ WHEN drv1=’1’ OR drv2=’1’ OR drv3=’1’ OR drv4=’1’
         ELSE ‘Z’;
```

6 Summary

We've looked at how develop descriptions of designs that are many bits wide. We have also seen how to exploit the underlying regularity of our designs to produce simple descriptions by using GENERATE loops.

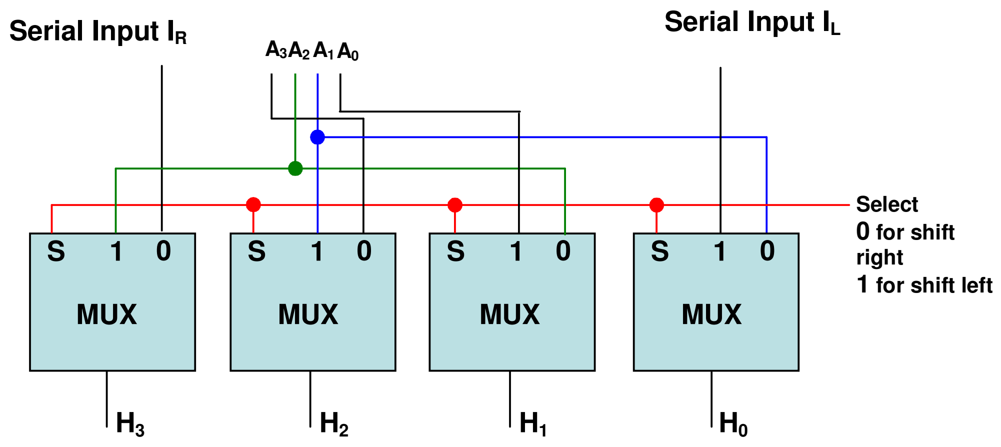
We have also seen how to write behavioural code, saying what we want the design to do, which is then turned by a synthesis tool into a circuit that implements the behaviour.

<u>You should now know...</u>

The meaning of the following:

- Standard Logic Vector (STD_LOGIC_VECTOR)
- Aggregates
- Iterative designs
- The GENERATE statement
- Use of multi-value logic
- The IEEE 1164 standard for multi-value logic

Exercise: Design a 4-bit shifter using multiplexers, you can build the multiplexer by either behavioural or structural description, but the shifter should be built using structural description.


Serial Input I_R
$A_3 A_2 A_1 A_0$
Serial Input I_L
S 1 0
MUX
S 1 0
MUX
S 1 0
MUX
S 1 0
MUX
Select
0 for shift right
1 for shift left
H_3
H_2
H_1
H_0