ENCS 533 - Advanced Digital Design
Lecture 10
Configuration Management and Packages

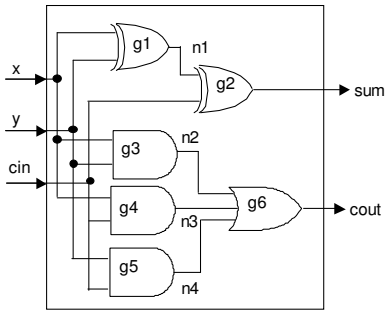
1 Introduction

In this lecture, we will look at a couple of the more advanced features of VHDL. *Configuration management* is a way to defer decisions about how our designs will be constructed until late in the design cycle. This enables us to change our mind about critical aspects of a completed design *without* having to rip everything and start from the beginning.

Packages are a way of putting commonly used definitions, declarations and functions into a place where they can easily be stored and accessed.

2 The Problem

Let's return to our example of the structural 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;
```

We assume that we have some logic gates available to build up our design, and we describe the design as an interconnection of these gates. The statements labelled g1 to g6 are instantiations of components in the library.

A simple metaphor for what is going is to imagine that the ARCHITECTURE is taking chips (xor2, and2 and or3) and soldering them onto a board with the wiring shown by the PORT MAPs.

This VHDL description is entirely legal and correct, but it is rather sloppy. There are two reasons why it is considered poor.

2.1 The problem with using the lazy form of instantiation

The method above assumes that there is something called xor2(simple) *already* compiled into the work library. But what if it isn't there? For example, it is not unusual for a design to split across several teams, and somebody else may be designing some of the parts that you are going to use for your design. It would be unfortunate if you are unable to begin until the other team has already finished.

2.2 The Problem with Static Configurations

Suppose we have several different families of logic gates to choose from. Let's say for the sake of argument that we have a Motorola family, a Siemens family and a Fujitsu family. Furthermore, let's assume that all of the gates have been modelled and compiled into the library. So in our gate library we have

- work.and2(motorola)
- work.or3(motorola)
- work.xor2(motorola)
- work.and2(siemens)
- work.or3(siemens)
- work.xor2(siemens)
- work.and2(fujitsu)
- work.or3(fujitsu)
- work.xor2(fujitsu)

Now we write our structural description of the full adder, building up our design from library elements. Now at this point we have to make a choice of a particular family, even though we don't really know which one we want at this stage.

The full-adders will be inserted into the n-bit adders, the n-bit adders will be inserted into the system, and *only then* we will know whether the total solution meets our requirements in terms of speed, cost, battery lifetime, etc. If we choose one particular family, and then go through the whole design cycle only to find out that the final solution violates a design constraint, it would be very tedious, time consuming, and possibly error-prone to track back through every single design unit, edit it and recompile.

We would like a way to reserve judgement about which library elements we are going to use until the last minute, when the whole design has been described, and the consequences of the decision can be properly simulated.

3 Configuration Management

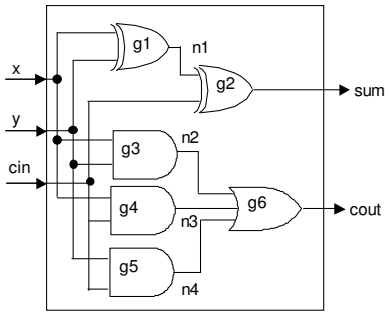
What we want is to be able to put down a component in our design that

- either hasn't been defined yet
- or we might decide to change later on

The process of tying up library elements to the instantiations in our design is called *binding*. We want to be able to write descriptions of designs before making the decision as to what exactly which components will be bound to the instantiations.

3.1 Writing instantiations that allow late binding

Here is a better description of the structural full adder (as usual, the lines that illustrate the new features are in **bold**):




```
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 late_binding OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
    COMPONENT comp1 IS
      PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
    END COMPONENT comp1;
    COMPONENT comp2 IS
      PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
    END COMPONENT comp2;
    COMPONENT comp3 IS
      PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
    END COMPONENT comp3;
BEGIN
    g1:  comp1 PORT MAP (x,y,n1);
    g2:  comp1 PORT MAP (n1,cin,sum);
    g3:  comp2 PORT MAP (x,y,n2);
    g4:  comp2 PORT MAP (x,cin,n3);
    g5:  comp2 PORT MAP (y,cin,n4);
    g6:  comp3  PORT MAP (n2,n3,n4,cout);
END ARCHITECTURE late_binding;
```

Here we have not (yet) tied our gates to anything stored in any library. Instead we have just used dummy names (comp1, comp2, comp3), and we will decide later what library elements we want to bind these names to.

In order for our code to be correctly analysed and compiled, VHDL needs to know which ports of the components are inputs and which are outputs. When we were directly binding the components to library elements, this information could be deduced by looking at the properties of the library elements. However, we haven't yet

chosen what elements of what library we are going to bind these components to, so we must explicitly declare the PORT MAP (and, if appropriate, the GENERIC MAP) of the components we are using. This is done through a COMPONENT declaration, which goes in the normal place for declarations, i.e. between the ARCHITECTURE statement and the first BEGIN.

A simple metaphor for what is going is to imagine that the ARCHITECTURE is taking sockets (comp1, comp2 and comp3) and soldering them onto a board with the wiring shown by the PORT MAPs. Later on we will decide which chips we want to plug into the sockets. When we solder a socket into the board, we may not yet know what chip we will place into it, but we do have to know how many inputs and outputs the chip will have so that we can choose a socket with the right number of pins. This is the function of the COMPONENT declarations.

3.2 Configurations

Eventually, we will have to decide what we want to bind the components to. This is done through a configuration. Here is a configuration that would bind our gates to the Fujitsu library of gates.

```
CONFIGURATION number1 OF fulladd IS        --Name of the entity
    FOR late_binding                       --Name of the architecture
      FOR ALL: comp1                       --Binding comp1 …
            USE ENTITY work.xor2(fujitsu); --to xor2(fujitsu)
      END FOR;
      FOR ALL: comp2                       --Binding comp2 …
            USE ENTITY work.and2(fujitsu); --to and2(fujitsu)
      END FOR;
      FOR ALL: comp3                       --Binding comp3 …
            USE ENTITY work.or3(fujitsu);  --to or3(fujitsu)

      END FOR;
    END FOR;
END CONFIGURATION number1;
```

Metaphorically, this is a list of what chip we want to plug into each of the sockets soldered into our design.

Usually there will be one configuration file for an entire design that may consist of many entities and architectures. The file will contain a configuration block for each of the design entities, and is a useful way of grouping together instructions as to how to bind the components used in entities and architectures.

The ALL keyword simply means that we should bind every instance of comp1 to work.xor2(fujitsu). If we wanted to, we could instead the instances explicitly:

```
CONFIGURATION number2 OF fulladd IS
    FOR late_binding
      FOR g1: comp1   USE ENTITY work.xor2(fujitsu);  END FOR;
      FOR g2: comp1   USE ENTITY work.xor2(fujitsu);  END FOR;
      FOR g3: comp2   USE ENTITY work.and2(fujitsu);  END FOR;
      FOR g4: comp2   USE ENTITY work.and2(fujitsu);  END FOR;
      FOR g5: comp2   USE ENTITY work.and2(fujitsu);  END FOR;
      FOR g6: comp3   USE ENTITY work.or3(fujitsu);   END FOR;
    END FOR;
```

```
END CONFIGURATION number2;
```

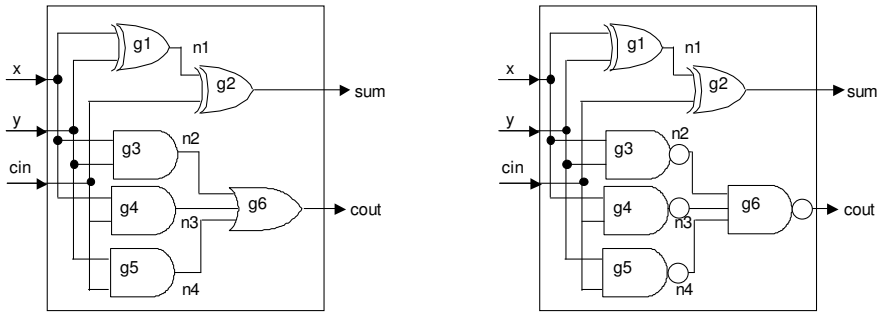
If we changed our mind about using the Fujitsu components, and decided instead to use Motorola, we would simply have to change the configuration:

```
CONFIGURATION number3 OF fulladd IS
    FOR late_binding
      FOR ALL: comp1
            USE ENTITY work.xor2(motorola);
      END FOR;
      FOR ALL: comp2
            USE ENTITY work.and2(motorola);
      END FOR;
      FOR ALL: comp3
            USE ENTITY work.or3(motorola);
      END FOR;
    END FOR;
END CONFIGURATION number3;
```

No matter how big the design, and how many files it occupied, we would need *only* to recompile the one file containing the configuration in order to reconfigure the whole design.

3.3 Reconfiguring gate function

So far we have implicitly assumed that the components from the different libraries do the same thing, but probably differ in some minor details, e.g. gate delay. So when we reconfigure the design, we are simply choosing between different versions of components that are functionally identical. *But this does not have to be the case*. Using the well-known identity that in Boolean logic OR-of-AND is identical in function to NAND-of-NAND, we can see that the following two designs have identical functionality



Because both use exactly the same placement of components, and wiring between them, *both* can be described by the following

```
ARCHITECTURE late_binding OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
    COMPONENT comp1 IS
      PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
    END COMPONENT comp1;
    COMPONENT comp2 IS
      PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
    END COMPONENT comp2;
```

```
    COMPONENT comp3 IS
      PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
    END COMPONENT comp3;
BEGIN
    g1:  comp1 PORT MAP (x,y,n1);
    g2:  comp1 PORT MAP (n1,cin,sum);
    g3:  comp2 PORT MAP (x,y,n2);
    g4:  comp2 PORT MAP (x,cin,n3);
    g5:  comp2 PORT MAP (y,cin,n4);
    g6:  comp3  PORT MAP (n2,n3,n4,cout);
END ARCHITECTURE late_binding;
```

The first design then represents the configuration

```
CONFIGURATION number4 OF fulladd IS
    FOR late_binding
      FOR ALL: comp1
            USE ENTITY work.xor2(simple);
      END FOR;
      FOR ALL: comp2
            USE ENTITY work.and2(simple);
      END FOR;
      FOR ALL: comp3
            USE ENTITY work.or3(simple);
      END FOR;
    END FOR;
END CONFIGURATION number4;
```

And the second is:

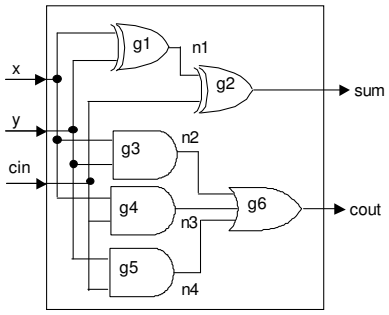
```
CONFIGURATION number5 OF fulladd IS
    FOR late_binding
      FOR ALL: comp1
            USE ENTITY work.xor2(simple);
      END FOR;
      FOR ALL: comp2
            USE ENTITY work.nand2(simple);
      END FOR;
      FOR ALL: comp3
            USE ENTITY work.nand3(simple);
      END FOR;
    END FOR;
END CONFIGURATION number5;
```

Returning to the metaphor, when we soldered the sockets into our board, we had in mind that the carry signal would be generated by inserting 2-input AND gate chips and a 3 input OR gate chip. But later on, we are free to change our mind about what chips to place into the sockets, as long as they have the same number of inputs and outputs, and here we have chosen to replace the AND and OR gates with NAND gates.

4 Other ways to define the binding

4.1 Component definition inside the ARCHITECTURE

Instead of using a separate configuration block, we can (if we wish) instead specify the binding inside the architecture. It comes in the normal place for declarations, i.e. between the ARCHITECTURE statement and the first BEGIN.




```
ARCHITECTURE late_binding OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
    COMPONENT comp1 IS
      PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
    END COMPONENT comp1;
    COMPONENT comp2 IS
      PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
    END COMPONENT comp2;
    COMPONENT comp3 IS
      PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
    END COMPONENT comp3;
    FOR ALL: comp1  USE ENTITY work.xor2(motorola);
    FOR ALL: comp2  USE ENTITY work.and2(motorola);
    FOR ALL: comp3  USE ENTITY work.or3(motorola);
BEGIN
    g1:  comp1 PORT MAP (x,y,n1);
    g2:  comp1 PORT MAP (n1,cin,sum);
    g3:  comp2 PORT MAP (x,y,n2);
    g4:  comp2 PORT MAP (x,cin,n3);
    g5:  comp2 PORT MAP (y,cin,n4);
    g6:  comp3  PORT MAP (n2,n3,n4,cout);
END ARCHITECTURE late_binding;
```

You will see this used very often in text books, though it lacks the flexibility of using a separate CONFIGURATION.

4.2 Default binding

Now let's return to our original late binding definition of the full adder:

```
ARCHITECTURE late_binding OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
    COMPONENT comp1 IS  PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
    END COMPONENT comp1;
    COMPONENT comp2 IS  PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
    END COMPONENT comp2;
    COMPONENT comp3 IS  PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
    END COMPONENT comp3;
BEGIN
    g1:  comp1 PORT MAP (x,y,n1);
    g2:  comp1 PORT MAP (n1,cin,sum);
    g3:  comp2 PORT MAP (x,y,n2);
    g4:  comp2 PORT MAP (x,cin,n3);
    g5:  comp2 PORT MAP (y,cin,n4);
    g6:  comp3  PORT MAP (n2,n3,n4,cout);
END ARCHITECTURE late_binding;
```

If we try to compile this, there will be no problem at all. However, if we then try to simulate it before we have provided a CONFIGURATION, then the VHDL simulator would report an error, something like

```
Elaboration error: comp1 not bound.
Elaboration error: comp2 not bound.
Elaboration error: comp3 not bound.
```

Metaphorically speaking, this means that no chip has been plugged into the sockets called comp1, comp2 and comp3. *Elaboration* is the process that a VHDL tool goes through in order to ensure that the system is fully specified in sufficient detail to be physically realisable. This involves ensuring that every component has a binding to a library element, i.e. ("making sure that every socket that you used has a chip plugged into it")[1].

Now consider the following description:

```
ARCHITECTURE default_binding OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
      COMPONENT xor2 IS
            PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
      END COMPONENT xor2;
      COMPONENT and2 IS
            PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
      END COMPONENT and2;
      COMPONENT or3 IS
            PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
      END COMPONENT or3;
BEGIN
    g1:  xor2 PORT MAP (x,y,n1);
    g2:  xor2 PORT MAP (n1,cin,sum);
    g3:  and2 PORT MAP (x,y,n2);
    g4:  and2 PORT MAP (x,cin,n3);
    g5:  and2 PORT MAP (y,cin,n4);
    g6:  or3  PORT MAP (n2,n3,n4,cout);
END ARCHITECTURE default_binding;
```

It is completely identical to the earlier description, with one important difference: *we have given the sockets the same name as gates that have already been compiled into the library.*

If we try to simulate this before providing a CONFIGURATION, the elaboration process will *assume* that we want to bind the components to the library elements of the same name. This is called *default binding*. If there are multiple architectures within the library (e.g. xor2(fujitsu), xor2(motorola) and xor2(siemens)), then default binding will use whichever architecture was compiled into the library last.

4 Packages

Throughout the code listings in this lecture, you will have noticed that the declarations of components have taken up a lot of room, without contributing much to the comprehensibility of our designs. It would be nice to find some way of tidying them away into some convenient location.

[1] It also involves ensuring that all generic parameters have a value assigned (i.e. if you have used an n-bit adder, making sure the value of n is assigned, and building the adder of the appropriate size)

Packages are ways of ways of grouping together definitions, declarations and functions in a convenient way. Here is an example that takes places our component declarations inside a PACKAGE called mygates.

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

PACKAGE mygates IS
      COMPONENT xor2 IS
            PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
      END COMPONENT xor2;
      COMPONENT and2 IS
            PORT (a, b: IN STD_LOGIC; c: OUT STD_LOGIC);
      END COMPONENT and2;
      COMPONENT or3 IS
            PORT (a, b, c: IN STD_LOGIC; d: OUT STD_LOGIC);
      END COMPONENT or3;
END PACKAGE mygates;
```

Once this is compiled into the library, it is available to use within my designs. So our structural full adder can now be written as

```
USE WORK.mygates.ALL;
ARCHITECTURE default_binding OF fulladd IS
    SIGNAL n1, n2, n3, n4: STD_LOGIC;
BEGIN
    g1:  xor2 PORT MAP (x,y,n1);
    g2:  xor2 PORT MAP (n1,cin,sum);
    g3:  and2 PORT MAP (x,y,n2);
    g4:  and2 PORT MAP (x,cin,n3);
    g5:  and2 PORT MAP (y,cin,n4);
    g6:  or3  PORT MAP (n2,n3,n4,cout);
END ARCHITECTURE default_binding;
```

The format of the USE statement is

```
USE Name_of_library.Name_of_package.Name_of_item.
```

If we want to use every item in the package, we use the keyword ALL. If we wanted to explicitly list the items in the package that we want to use, we could do so like this

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

By now you should be able to guess that STD_LOGIC_1164 is a package stored in the IEEE library.

5 Package bodies

If the package contains nothing but declarations, then the method shown in the last section is sufficient. If, on the other hand, we want to put executable code into a package, we use something called a *package body*.
Suppose, for example, that we wanted to create a utility function, max(a,b) which return the larger of its two arguments, a and b. This is how it would be done.

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_signed.ALL;
      PACKAGE my_utils IS
      FUNCTION max (a,b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
      END PACKAGE my_utils;

      PACKAGE BODY my_utils IS
      FUNCTION max (a,b: STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR IS
      BEGIN
            IF a>b THEN
                  RETURN a;
            ELSE
                  RETURN b;
            END IF;
      END FUNCTION max;
      END PACKAGE BODY my_utils;
```

The interface declaration of the function is declared in the package. This declares how many arguments there are, what their type is, and what the return type of the function is. The actual code of the function is contained in a package body.
This process of having to declare a function interface, and then separately having to say what goes on inside the function may look irritating, but it satisfies the VHDL philosophy[2] of separating the description of the interface of a unit from the description of its function.
This approach has some important uses. One of these is that a designer may develop code with the intention of selling it to users. If its source code could be freely read, then the designer may suffer financial loss, so the *code* of the function may be confidential (and just distributed in compiled form). However, if anyone else is to use the code, then users must be able to know what its interface is.

Now if we want to use the max function, then we have a USE clause, opening up the my_utils package and exposing all of its features, and then we can use the max function in our code.

```
      USE my_utils.ALL;
      ARCHITECTURE simple OF test IS
      SIGNAL a: STD_LOGIC_VECTOR(7 DOWNTO 0):=X"0A";
      SIGNAL b: STD_LOGIC_VECTOR(7 DOWNTO 0):=X"06";
      SIGNAL c: STD_LOGIC_VECTOR(7 DOWNTO 0);
      BEGIN
            c <= max(a,b); --c will get the value X”0A”
      END ARCHITECTURE simple;
```

[2] As is the case with the distinction between ENTITY and ARCHITECTURE.

You should now know...

The meaning of the following

- Elaboration
- Configuration
- Binding
- Late binding
- Default binding
- Packages and Package body

How to use configuration management features of VHDL.
How to write simple packages.