

VHDL 6

VHDL Objects

Objects in VHDL

6.1

- VHDL provides a number of elements for storing and representing data.
- These elements are collectively known as **objects**.
- In VHDL there are 3 primary classes of object:
 - **signals** — *to represent wires, buses and registers in a circuit*
 - **variables** — *similar to variables in a computer program*
 - **constants** — *values that do not change but are useful to label*
- All objects must be designated a data **type**, such as **integer**, **real** etc... this is done when the object is declared.
- VHDL is a **strongly typed language** and there are several different types available. This chapter will summarise the main types of interest.
- Subsequent chapters will then provide more detail on VHDL data types.

Notes:

VHDL Objects

In VHDL there are a number of elements which can be used to represent and store data in a description of a system. These objects are collectively known as objects. The three basic objects are **signals**, **variables** and **constants**.

In VHDL-93 a fourth object called **file** was introduced. In your hardware descriptions you will mainly make use of signals, variables and constants to achieve given functionality. Files are mostly used in test benches for reading and writing test vectors to and from data files.

When you declare an object you must give it a type, such as *bit*, *integer*, *boolean*, *bit* or *real*.

Constants

6.2

- An example of declaring a constant is:

```
constant NUMBER_OF_BITS: integer := 8;
```

literal

- Constants** are **objects** which:
 - ... are assigned a value during declaration ...
 - ... and do not change value during simulation or execution.
- Constants can help to create designs which are:
 - More readable
 - More easily modified
- A **literal** is simply an explicit value, and literals do not necessarily have a type.



Variables

6.3

- An example of declaring a variable is:

```
variable current_bit: integer := 0;
```

optional initialisation

- VHDL variables are similar to variables in conventional programming languages. They are generally used to store **intermediate values** in **sequential statements**.
- Some restrictions and properties of **VHDL variables** are:
 - Can only be used in **processes**, **procedures** and **functions**
 - Unlike signals, variables are assigned values **immediately**
- Variable assignments differ in syntax from signal assignments, e.g:

```
current_bit := current_bit + 1;
```



Notes: Constants

VHDL Objects

A constant is an object which is declared and initialised with a value and then does not change during a simulation. Constants can be declared in any VHDL construct which has a declaration region. For example you can declare a constant in an architecture body, in a process or in a package.

Once a constant has been declared it can be used in a "read only" fashion within a design. For example if you were designing an 8-bit counter, you could define a constant called NUMBER_OF_BITS. You could then design the counter and write the code based on the NUMBER_OF_BITS constant. Setting NUMBER_OF_BITS := 8 would give you an 8-bit counter. However if you later decide that you require a 10-bit counter, all you need to do is change NUMBER_OF_BITS to 10. Overall the design is easy to modify to implement any number of bits.

Using the NUMBER_OF_BITS constant in the design also makes it more readable. For instance consider what the design would be like if the literal value 8 was scattered throughout it. NUMBER_OF_BITS is far more meaningful than just the number 8.

Literals

A literal is an explicit value which is assigned or used within an expression. For example 2, '0' or 'hello' are all literals.

Ver 21.130



Notes:

VHDL Objects

A variable is an alternative to a signal for storing a value within a process. So far, signals have been considered as "pieces of wire" that connect together the elements of a design. Variables can represent wires as well as being used in other applications. Only shared variables can be used by more than one process. Variables have a name and type, just like a signal.

Variables are primarily used in behavioural descriptions. They are used in the same way as variables in conventional programming languages. That is to say, they are used to store intermediate values in the implementation of an algorithm. Recall that an algorithm is essentially a sequence of steps that describe one way to perform a particular task. In VHDL variables can *only* be used in sequential statements (i.e. processes, procedures and functions).

Unlike a signal, a variable does not necessarily correspond to a particular element in a circuit. If the design is synthesisable, then the synthesis tool will decide what the variable represents. For example a variable could represent a register.

An important property of a variable is that it is assigned a value immediately. Recall that a signal can be assigned an explicit delay after which the assignment occurs. A variable assignment cannot specify a delay since events cannot be scheduled on a variable.

Another important property of a variable is that, unlike signals, a variable does not have a history of changes in value (events) which have occurred during the simulation.

Ver 21.130



Signals

6.4

- An example of a signal declaration is:

```
signal carry_out: std_logic := '0';
```

optional initialisation - ignored during synthesis

- Signals are used to **connect concurrent elements** (they are similar to wires in an electronic circuit).
- Some important properties of signals are:
 - each signal in a simulation has an associated **history**
 - signal initialisation** is typically ignored during synthesis
 - a **signal assignment** (shown below) can include a **delay**

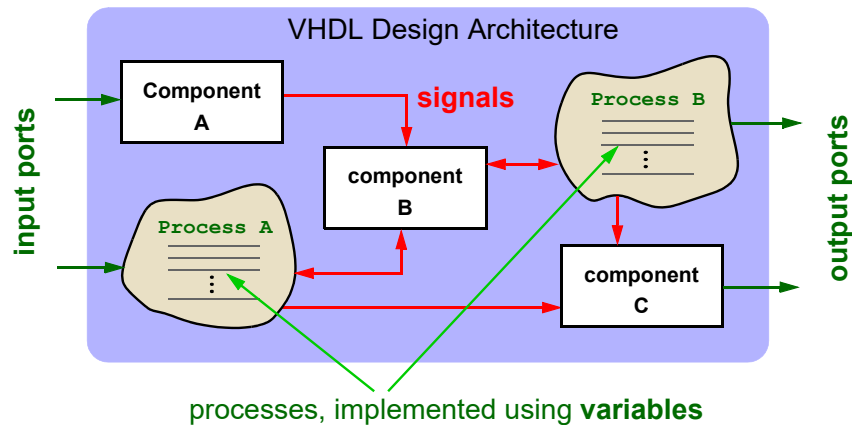
```
carry_out <= '1' after 10ns;
```



Signals and Variables

6.5

- Signals** are used as interconnects between **concurrent elements**.



- Variables** are used in sections of sequential code such as **processes**.



Notes:

VHDL Objects

Signals are used to connect concurrent elements (such as components, processes and concurrent assignments) together. This is similar to the way that wires are used to connect components on a circuit board or in a schematic. In fact it is very often the case that a signal will be translated into a wire or a set of wires during synthesis.

Signals can be declared globally in an external package, or locally within an architecture, block or other declarative region. Note that not all synthesis tools support global signals.

Ver 21.130



VHDL Objects

Notes:

Signals and variables can be used interchangeably in some instances. However, there are some important differences between them which should be remembered.

Variables

- Can only be used inside the process where they are declared;
- Cannot appear in a sensitivity list;
- Cannot have delays - assigned instantaneously.

Signals

- Can be used for communication between processes;
- Can appear in a sensitivity list and therefore cause a process to wake up when an event occurs;
- Can have delays;
- Have an associated history which can be viewed at the end of the simulation as a waveform.

Ver 21.130



VHDL Types

6.6

- VHDL is a **strongly-typed** language:
 - All **data elements** must have an **associated type**.
 - Conversions between types** are only possible with the aid of **explicit conversion functions**.
 - Constrained uses of a type can be defined by a **subtype**.
- VHDL types may be categorised into five classes:
 - Scalar Types**: *integers, reals, physical and enumerated types*
 - Access Types**: *like pointers in software programming*
 - File Types**: *typically disk files - contain a sequence of values*
 - Composite Types**: *arrays and records*
 - Protected Types**: *used for implementing shared variables*



VHDL Objects

Notes:

All types in VHDL can be placed in one of the four categories on the previous slide. Most synthesis tools only support synthesis of some scalar types and composite types.

Scalar types are the most commonly used, and are used to represent various types of numbers, signals, and physical quantities.

Access types are like pointers in software languages. They are used for describing high level algorithms. Most synthesis tools do not support them.

File types are not synthesisable however they are very useful for storing test vectors during verification.

Composite types are particularly useful for grouping together signals to represent buses.

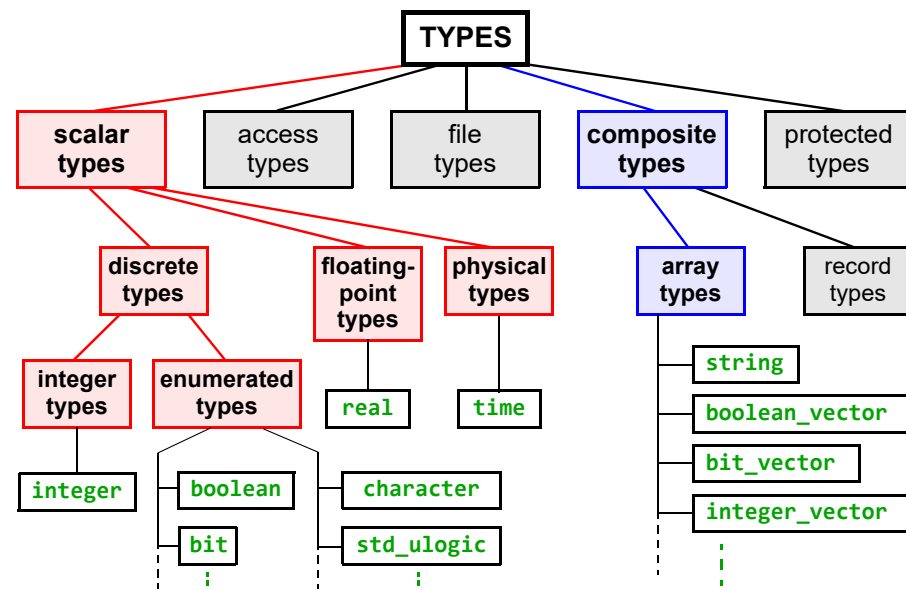
Protected types are used when implementing shared variables, i.e. variables which can be accessed by more than one process or subprogram. This is an advanced feature of the language, and will not be covered in this course.

Ver 21.130



Type Classification

6.7



VHDL Objects

Notes:

The diagram in the main slide shows the different classes of types in the VHDL language. We will be concentrating on the two classes which are highlighted: **Scalar Types** and **Composite Types**. Within the **Composite Types** class, we will concentrate on **array types**.

Ver 21.130



Summary

6.8

- In VHDL, **constants**, **variables** and **signals** are **objects** for storing information.
- **Signals** describe **connections** between concurrent elements in a circuit, whereas **variables** store data within **sequential descriptions**.
- All constants, variables and signals must have a **type**.
- VHDL is a **strongly typed** language.
- A number of **predefined (standard) types** exist in the language. The categories of types that are of principal interest are:
 - Scalar types
 - Composite types
- ... and we will go on to cover these in more detail over the next few chapters.



Notes:

VHDL Objects

Ver 21.130



VHDL 7

Scalar Data Types

Scalar Types in VHDL

7.1

- VHDL **objects** are elements for storing and representing data. Every object must have an associated **type**.
- The most fundamental data types are **scalar types**. A scalar type is used to represent a single element of data.
- Several different scalar types exist. The choice of scalar type depends on the nature of the data to be represented, which can include:
 - Numbers, bits, characters, and time.
- We will review existing types that are defined within the VHDL **standard** package, and within the **IEEE std_logic_1164** package.
- It will be shown how to work with these types.
- We will also demonstrate how to **define your own** types and subtypes.

Notes:

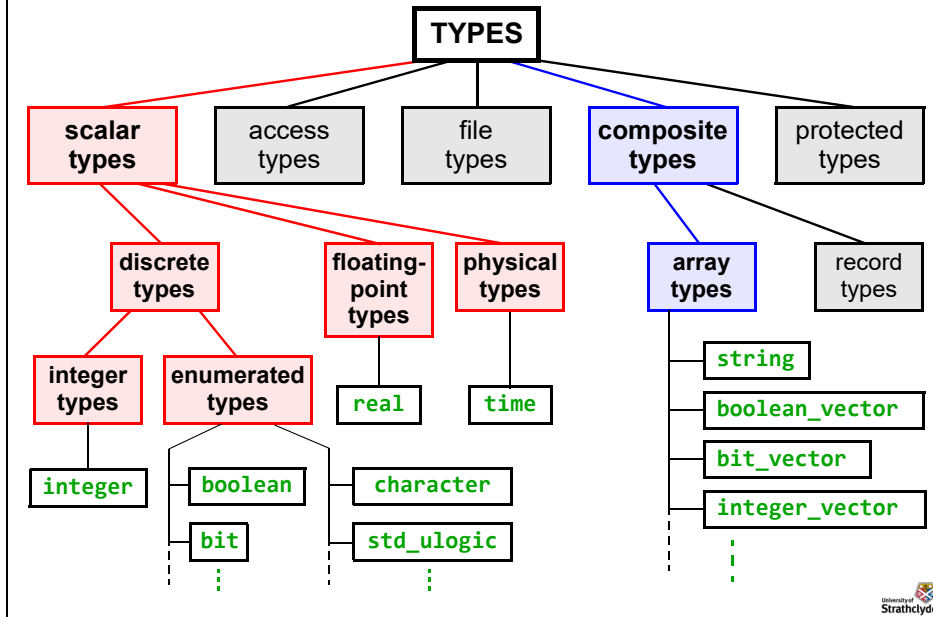
Scalar Data Types

In the last chapter, we saw that any time we define a new object (i.e. a signal, variable or constant), then it must be given a type. As we will see, there are various different categories of types, and therefore quite a selection of possibilities. It is important to have a good understanding of the different data types that are available, and how to work with them.

This chapter covers scalar types, which are the most fundamental type of types!

VHDL Type Classification (recap)

7.2



Notes:

Scalar Data Types

The diagram in the main slide shows the different classes of types in the VHDL language. This slide is repeated from the previous chapter on VHDL objects.

In this presentation, we focus on **Scalar Types**, which correspond to individual data elements.

Within the **Composite Types** category, we will concentrate on **array types**, and you can see that some array types are related to a scalar type, e.g. a `bit_vector` is an array of bits.

Ver 21.131



The Standard Package

7.3

- VHDL has a standard package which contains some **predefined types** available for you to use.
- The standard package is **implicitly declared** (i.e. you don't actually see any code!) at the top of any VHDL file you create.
However, if you did see the declaration typed out, it would be....

```
library std;
use std.standard.all;
```

- The standard package contains many of the **basic types** we use very often... such as:
 - boolean, bit, bit_vector, character, integer, real, time, string
- Further types are defined for working with **files**.
- There is also a type called **severity_level** which is used for **reporting in simulations**.

Notes:

Scalar Data Types

As we will see later in the course, a package is simply a VHDL construct which is used to group frequently used pieces of code together. In the general case, packages contain things like type and constant declarations, subprograms, and components.

The standard package mostly just declares types. After some of the types, subtypes of that type are also declared. A subtype is a constrained version of a type; i.e. it contains a subset of the values belonging to the type. This is perhaps easiest to illustrate with an example.... on the next slide, we will consider the **natural** and **positive** subtypes of **integer**.

```
package standard is

    type boolean is (false, true);

    type bit is ('0', '1');

    -- ... other type and subtype declarations...
    -- ... further examples coming up!

end standard;
```

One of the few items in the standard package which is *not* a type declaration is the "now" function. When this function is executed, it returns the current (simulation) time.

Ver 21.131



Scalar Types (Numeric)

7.4

- Scalar types have a single **numeric** or **enumerated** value.
- Below we consider **numeric** types.
- Some examples from the **standard package** are:
 - **integer** — integer values from -2147483647 to +2147483647
 - **natural** — subtype of **integer** ... from 0 to largest integer
 - **positive** — subtype of **integer** ... from 1 to largest integer
 - **real** — for *floating point* numbers
 - **time** — a physical type for representing time
- Note that the **time** type has units associated with it, whereas all of the other types listed above are simply numbers (with no units).



Scalar Data Types

Notes:

As defined in the **standard** package:

```
-- .....  
type integer is range -2147483647 to 2147483647;  
type real is range -1.0E308 to 1.0E308;  
type time is range -2147483647 to 2147483647  
  units  
    fs;  
    ps = 1000 fs;  
    ns = 1000 ps;  
    us = 1000 ns;  
    ms = 1000 us;  
    sec = 1000 ms;  
    min = 60 sec;  
    hr = 60 min;  
  end units;  
subtype natural is integer range 0 to integer'high;  
subtype positive is integer range 1 to integer'high;  
-- .....
```

Ver 21.131



Scalar Types (Enumerated)

7.5

- An **enumerated** type is composed of a set of **literals**.
- The type can **only** take on one of these **literal values** (as defined in the type declaration).
- Examples from the **standard package** are:
 - **boolean** — can take on a (literal) value of true or false
 - **bit** — can take on a value of '0' or '1'
 - **character** — type for representing alphanumeric characters
 - **severity_level** — for invoking assertions - can take on values:
note, warning, error, failure



Scalar Data Types

Notes:

As defined in the **standard** package, the built in enumerated types are:

```
type boolean is (false,true);  
type bit is ('0', '1');  
type character is (  
  ...  
  ...  
  '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',  
  'H', 'I', 'J', ...  
  ...  
  ...  
);  
type severity_level is (note, warning, error, failure);
```

Ver 21.131



Defining Your Own Enumerated Types 7.6

- **Enumerated types** greatly increase the **readability** of code. Sometimes it is useful to define your own enumerated type.
- You can define your own **enumerated type** using the syntax...

```
type primary_colour is (red, green, blue);
```

↑
name of type

↓
set of literals

- Objects of an enumerated type **can only take on values** belonging to the set of literals which define that type.
- Your defined types can also be constrained to form **subtypes**, which contain fewer literals.



Defining Your Own Subtypes 7.7

- A subtype is a **constrained** version of a type.
 - We can create new subtypes of **predefined types**.
 - We can also create subtypes of **our own types**.
- For example, suppose we wanted to create a **subtype** of the **integer type**, corresponding to the days of the week...

```
subtype days_of_week is integer range 1 to 7;
```

- We could also create a **subtype** of the primary_colour **type** we defined previously...

```
subtype red_green_only is primary_colour range red .. green;
```



Notes:

The main reason for the existence of enumerated types is that they greatly increase the readability of the code. For example a traffic light controller might use an enumerated type such as:

```
type light_state is (red, red_amber, green, amber);
```

The VHDL code can then contain a variable or signal of type light_state which stores the current state of the traffic lights. The code works directly with the literals of the enumerated type ... i.e. red, red_amber, green, amber. This makes life easy for the programmer when compared to using a set of integers, for example 0 to 3 to represent the states.

Ver 21.131



Notes:

A subtype defines a new type, which is based on an existing type, but can only take on values which lie within a defined subset of the values of the base type. When we define the subtype, we must specify the range of values that it can take on.

For example, the implication of defining the subtype days_of_week is that we will never try to assign a constant / variable / signal of this type with a value lying outside the defined range. For example, if we tried to assign it with the value 8, an error would be flagged.

Similarly, having defined the subtype red_green_only, we cannot attempt to use the literal blue when working with the subtype. (This is because blue belongs to the type, but not the subtype!)

There is an advantage to forming subtypes; if we constrain the number of literals to only those required, then the synthesis tool will be able to realise the circuit more efficiently.

Ver 21.131



Synthesis of Integers and Enums

7.8

- Some of the predefined VHDL types can be **synthesised**.
- These include **integers** and **enumerated** types.
- For **integers**, the synthesiser will create the number of binary logic elements required to represent the number, for example:

| (constrained) range | # bits required |
|---------------------|-----------------|
| 0 to 15 | 4 |
| 14 to 15 | 4 |
| -128 to 127 | 8 |

- Hence the importance of subtypes! They help to **reduce complexity**.
- For **enumerated** types, the synthesiser works out how many bits will provide enough combinations to represent the entire set of literals.
- For example, if there are **7 literals**, then **3 bits** will be enough.



Modelling Signal Values in VHDL

7.9

- Recall that **bit** is an **enumerated type** with the set of literals ('0', '1'). The **bit** type is useful for describing digital circuits at a high level of abstraction.
- However, at a lower level of abstraction, signals need to be described and modelled in **more detail**.
- The **std_ulogic** type (also an enumerated type) provides the facility to model the electrical characteristics of the signal...
 - i.e. not just '0' and '1', but **other conceptual levels** too.
- The **std_logic** type is a resolved subtype of **std_ulogic** (more on this in the next slide).
- std_ulogic** and **std_logic** are not a built-in part of the language, but they are standardised in IEEE package **std_logic_1164**. To use these types and associated functions, we must **explicitly include** the IEEE package.



Notes:

In the traffic light example, there are 4 literals in the enumerated type, each representing one of the states of a state machine. The synthesis tool might choose to encode the state as a 2-bit number as shown in the table below.

| | |
|-----------|-----|
| red | 0 0 |
| red_amber | 0 1 |
| green | 1 0 |
| amber | 1 1 |

The example above used sequential encoding of the states (i.e. consecutive binary numbers). However there are also other methods of encoding using binary numbers, for example Gray coding, One-hot and One-cold. Larger state machines (i.e. enumerated types with larger sets of literals) tend to use one of these methods, whereas sequential encoding is preferred for smaller sets. Ultimately you can leave this aspect to the synthesiser and it will work out the best scheme to use.

The key point of enumerated types is that the programmer can refer to the literals, which make sense to a human. The synthesis tool deals with how to represent the states as a binary code and creates the required logic elements or wires corresponding to the object.

Ver 21.131



Scalar Data Types

Notes:

When we want to use the **std_ulogic** and/or **std_logic** types, (which is often!), then we must first include the IEEE **std_logic_1164** package. Doing so makes available the type definitions, and the resolution functions and other functions defined for working with these types.

For this reason, you will frequently see the following two lines of code at the top of VHDL files. You must add these lines before **any** use of **std_ulogic** or **std_logic**.

```
-- library and package declaration
-- (at the top of any file requiring std_ulogic / std_logic)

library IEEE;
use IEEE.std_logic_1164.all;
```

Ver 21.131



STD_ULOGIC and STD_LOGIC

7.10

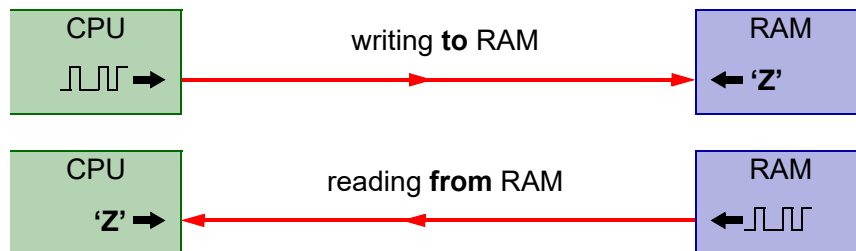
- **std_ulogic** is an enumerated character type with values:
('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-')
- For synthesis the most important levels are '0', '1' and 'Z'
- All levels **other than '0'** and '1' are known as **metallogical** values
 - Metallogical values do not exist in the real world
 - they are just a concept used for simulation
- **std_logic** is a **resolved subtype** of **std_ulogic**
 - a resolved type is one which has a resolution function
 - a resolution function decides which logic level the signal takes on when more than one source is driving the signal



Signals with Multiple Drivers

7.11

- While many of our designs involve signals with **a single driver**, sometimes we need to model signals with **more than one driver**.
- An example is a **bus**, which may have **several connected modules**.
- When a source is not writing to the bus, it should be in the **high impedance state** (driving a 'Z' on to the bus).



- The **resolution function** determines the signal value resulting from the two drivers. In this case the signal is resolved to the **non-Z driver**.



Notes:

As defined in package std_logic_1164 :

Scalar Data Types

```
-- the std_ulogic enumeration
type std_ulogic is
(
    'U',      -- Uninitialised
    'X',      -- Forcing Unknown
    '0',      -- Forcing 0
    '1',      -- Forcing 1
    'Z',      -- High Impedence
    'W',      -- Weak Unknown
    'L',      -- Weak 0
    'H',      -- Weak 1
    '-'      -- Don't Care
);

-- resolution function declaration
function resolved ( s : std_ulogic_vector ) return std_ulogic;

-- define std_logic as a resolved version of std_ulogic
subtype std_logic is resolved std_ulogic;
```

IMPORTANT NOTE: since **std_ulogic** is an enumerated type with character literals, the literals are **case sensitive!**

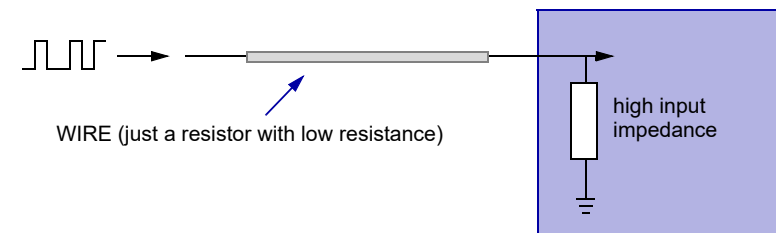
Ver 21.131



Scalar Data Types

Notes:

The high impedance state is required for modelling tristate buses. A component which is reading data from a bus must have high input impedance in order to maximise the voltage which is dropped across it. This high input impedance state is modelled by the **std_logic** type as an additional 'Z' logic state.



Obviously 'Z' is not an actual logic level. You could not measure it on a wire with a voltmeter. However it is essential for modelling situations where more than one circuit element can drive a logic level onto a wire.

Tristate signals must be **resolved**. For example if one source drives a signal to logic level '0' and another drives a 'Z' onto the signal, the simulator must decide who wins! In this case logic level '0' would win. Similarly, if the drivers were '1' and 'Z', then the '1' driver would win.

Another possibility is that — perhaps by accident — your VHDL code has two sources writing to the same signal. If one writes that value '0' and the other '0', then the resolved value will also be '0'. However if one writes '0' and the other '1', what happens? In fact, the value will be resolved to 'X' - which means 'forcing unknown'.

This resolution is made possible when using the **std_logic** type, because it has a resolution function capable of determining what the eventual output value ought to be, when there are two or more drivers.

Ver 21.131



Type Conversion

7.12

- Sometimes it is necessary to **convert between types**, and this is done **explicitly** with the aid of **type conversion functions**.
- For example, suppose we wanted to convert numbers between **integer** and **real** types....

```
-- convert and assign to signals
```

```
r <= real(1234);  
i <= integer(76.4);
```

- Note that when a **real** number is converted to an **integer**, it is rounded to the **nearest whole number** (thus a loss of precision may result).
- Another useful function is **to_string(x)**, which will take a **scalar value of any type** (denoted by x) and convert it to a **character string** representation.
- Other conversions can be made between **closely related types**.



Scalar Attributes

7.13

- Attributes provide a way to extract:
 - **information about a type**
 - **about the value(s) of an object of a given type**
- Let us consider again the following user defined enumerated type

```
type light_state is (red, red_amber, green, amber);
```

- Some examples of attributes of enumerated types

| | | |
|----------------------------|---|-----------|
| light_state'left | → | red |
| light_state'right | → | amber |
| light_state'val(0) | → | red |
| light_state'pos(red_amber) | → | 1 |
| light_state'right_of(red) | → | red_amber |



Notes:

The **to_string()** function is capable of taking any scalar type (integer, real, enumerated etc.) and converting it to a character string. The default formatting rules which apply in each case are different... for example a real number expressed as a string will use exponential notation. There is also the facility to explicitly specify how the string should be formatted, although that is beyond the scope of these notes.

Conversion from numeric and enumerated types to strings may be useful, for example, when writing log files during the debugging process.

Scalar Data Types

Ver 21.131

Notes:

The following attributes are predefined for integer and enumerated types.

type'left — returns the left most literal

type'right — returns the right most literal

type'high — returns the highest literal

type'low — returns the lowest literal

type'pred(value) — returns the literal which is one lower than **value**

type'succ(value) — returns the literal which is one higher than **value**

type'leftof(value) — returns the literal which is left of **value**

type'rightof(value) — returns the literal which is right of **value**

type'val(value) — returns the value at a given position numbering is 0,1,2 ... etc ... from left to right

type'pos(value) — returns the position of a value within the set of literals

Ver 21.131



Scalar Data Types



Summary

7.14

- All constants, variables and signals must have a **type**. VHDL is said to be a **strongly typed** language.
- In this chapter we have focused on one major category of types, that is, **scalar types**. A scalar type can be used for individual data elements.
- A number of **predefined (standard) types and subtypes** exist in the language — we saw examples from:
 - The **VHDL standard package**, such as numeric types (integer, natural), and enumerated types (bit, boolean).
 - The **IEEE std_logic_1164 package** defines additional types which have some useful features for describing hardware.
- Programmers can also define their own **types** and **subtypes**.
- **Conversion functions** are needed to convert between types. We can also use **attributes** to find out particular properties of defined types.



Notes:

Scalar Data Types

Ver 21.131



VHDL 8

Composite Data Types

Introduction

8.1

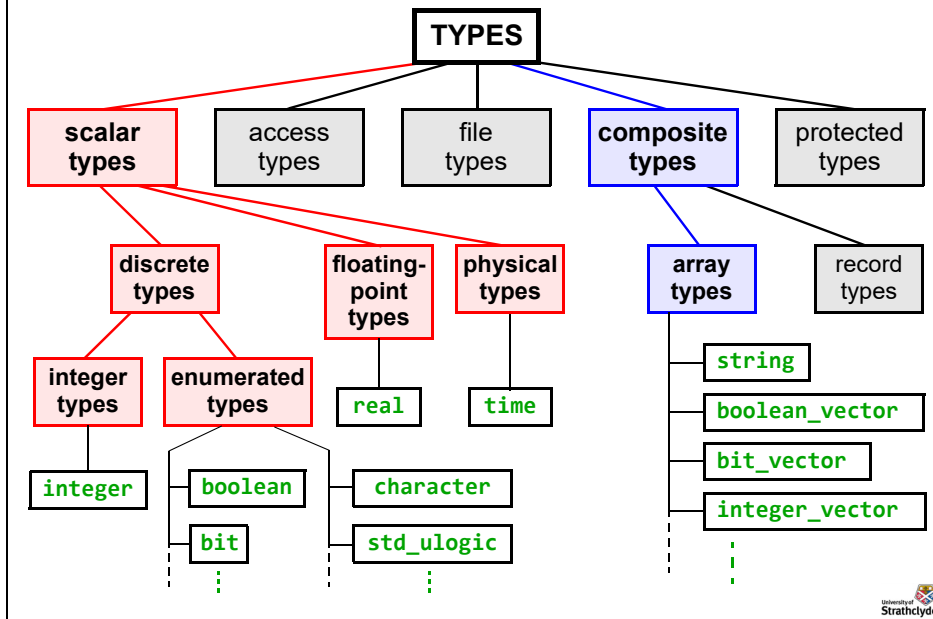
- Previously, the different *VHDL objects* were introduced...
 - *Constants*
 - *Variables*
 - *Signals*
- We saw that there are various different sorts of *data types* in VHDL...
- The most often used are *scalar* and *composite* types.
- In the last presentation, we looked at *scalar* types. Now we will focus on *composite* types, and in particular we will consider how to work with *arrays*.
- The last part of the presentation is about *VHDL operators*, and this applies to both *scalar* and *array* types.

Notes:

Composite Data Types

Type Classification (recap)

8.2

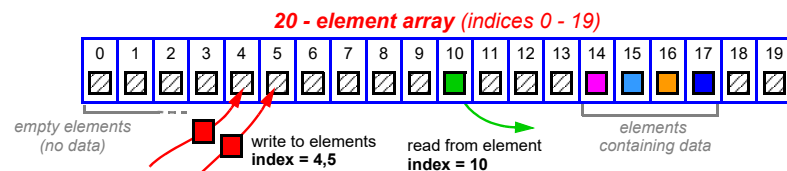


Arrays

8.3

- An **array** contains a **collection of elements** of **the same data type**.
- Arrays are **fundamental data structures** in software programming. You can think of an array as a series of **"boxes"** which can store data.

- We can refer to each of the boxes (elements of the array) individually or in groups, using an **array index**. For instance, **write to elements 4 and 5** or **read from element 10**.



- We can also work with the collection (array) **as a whole**.
- In VHDL, we are describing **hardware**, so an array usually represents something physical, like a collection of **wires** or **memories**.

Notes:

Composite Data Types

The diagram in the main slide shows the different classes of types in the VHDL language.

In this presentation, we deal with **Composite Types**. Within the *Composite Types* class, we will concentrate on **array types**, and review records only briefly. This is because arrays are in far more common usage.

Ver 21.26

Notes:

Composite Data Types

Arrays are 1-dimensional in the simplest case, although you can also build 2-dimensional or potentially even higher order arrays.

In the case of a 2-dimensional array, then the array is referenced by two indices. This is analogous to the indexing of elements of a matrix. As before, we can work with the matrix as a whole, or with individual elements or groups of elements, and we can also reference rows and columns as required.

20 x 4 element array: (indices 0 - 19)(0 - 3)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | | | | | | | | | | | | | | | | | | | | |
| 1 | | | | | | | | | | | | | | | | | | | | |
| 2 | | | | | | | | | | | | | | | | | | | | |
| 3 | | | | | | | | | | | | | | | | | | | | |



access elements in
column index = 3



access element in
row index = 2,
column index = 12



access elements in
row indices = 0:3,
column indices = 16:18

Ver 21.26

Array Types in VHDL

8.4

- In VHDL, arrays are essential for describing **regular hardware elements**, such as **memory blocks** and **buses of wires**.
- There are two predefined array types in VHDL (from the **standard** package) ...

- **string** — *an array of characters*
- **bit_vector** — *an array of bits*

- Examples of declaring signals of these types are:

```
signal str : string(1 to 5);  
signal bts : bit_vector(0 to 4);
```

- **IEEE std_logic_1164** also defines **std_logic_vector** — an **array** form of **std_logic**.
- We can also declare **our own array types**.



Declaring an Array Type

8.5

- A full declaration of an array must specify:
 - the **type of data** that will be stored in the array
 - the **type of indexing** (an integer or enumerated type)
 - the **range of the indexing**
- For example, the code below creates a signal array of real numbers...

```
type real_vect is array (natural range <>) of real;  
signal temp : real_vect(0 to 4);
```

index range

index type

array type

indicates an
unconstrained array

- Notice that we have effectively created a **new type** (real_vect)!



Notes:

Composite Data Types

As defined in the **standard** package ... the predefined array types are:

```
type string is array (positive range <>) of character;  
type bit_vector is array (natural range <>) of bit;
```

Note that due to this definition, the index range of a string can only include integers greater than or equal to 1.

There are only a few predefined array types, and it is common to declare your own types to satisfy the requirements of your design.

This means that if we want to create a signal or other object of a given form, not covered by one of the predefined array types, we must take two steps:

- First, write a declaration for the new array type.
- Second, declare a signal (constant, or variable) of that type.

Ver 21.26



Notes:

Composite Data Types

As noted in the previous slide, we effectively created a new type in order to define an array. The new type was defined using the following notation ...

```
type real_vect is array (natural range <>) of real;
```

The symbol <> is used here to indicate that the real_vect is an **unconstrained** array ... that means that the range of its indexing is not defined yet ... and therefore its size is undefined. Currently the only restriction on the range is that it must include only *natural* integers ... which are integers greater than or equal to zero.

Alternatively we could have declared a **constrained** array using the following notation ...

```
type real_vect2 is array (natural range 0 to 5) of real;
```

Therefore to declare a signal of type real_vect2 we do not need to specify its range as this is already specified by its type.

```
signal temp : real_vect2;
```

To summarise then, we have two options:

- Declare an **unconstrained** array type (range unspecified), then **constrain** (specify the range) when declaring an object of that type.
- Declare a **constrained** array type (range specified), and then simply declare an object of that type. The object will be constrained to the same array dimensions as the type.

Ver 21.26



Index Range of an Array

8.6

- The **index range** is specified as either **ascending** or **descending** from left to right - note that the correct keyword must be used:

```
signal A : real_vect(0 to 15);      -- ascending
signal B : real_vect(15 downto 0); -- descending
```

- The range does not have to start or end at zero ... for example:

```
signal some_reals : real_vect(5 to 20);
```

- Individual elements** of an array can be accessed using parenthesis..

```
-- assume R is a signal of type real
R <= some_reals(5);  -- get leftmost element
some_reals(20) <= 1e3; -- set rightmost element
```



Array Assignments

8.7

- Arrays **can be assigned to each other**, provided that the **source** and **target** arrays:

- Store **the same type of data**, and ...
- ... contain **the same number of elements**.

- This means that the following two arrays can be assigned ...

```
signal A : bit_vector(0 to 7);
signal B : bit_vector(3 to 10);
-----
B <= A;  -- signal assignment is possible
```

- Elements are assigned **left to right** without reference to the ranges.
- In addition we can **access and assign** part of an array called a **slice**.

```
B(3 to 5) <= A(5 to 7);
```



Notes:

Composite Data Types

The index range of an array specifies which numbers correspond to which entries in the array.

For example:

```
signal A : bit_vector(0 to 7);
```

A :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

```
signal B : bit_vector(3 to 10);
```

B :

| | | | | | | | |
|---|---|---|---|---|---|---|----|
| 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|----|

```
signal C : bit_vector(7 downto 0);
```

C :

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|

Ver 21.26

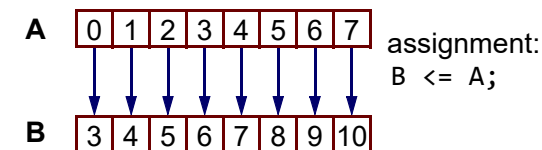


Notes:

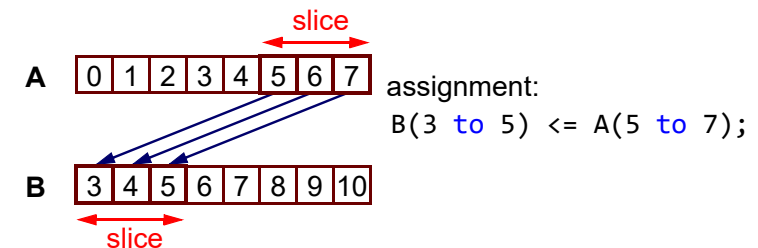
Composite Data Types

VHDL is flexible about the indexing of arrays.

The example on the previous slide defined two signal arrays, **A** and **B**, of the same size and type, but with different ranges. The diagram below shows how the elements are assigned from one array to the other. Note that the assignment makes no reference to the indexing of the arrays. It simply assigns the elements from left to right.



A diagram of the **slice** assignment is shown below.



Ver 21.26



Array Literals

8.8

```
signal a : bit_vector(3 downto 0);
```

- Assuming the above declaration, we can assign literals *one by one*...

```
a(3) <= '1';  
a(2) <= '0';  
a(1) <= '0';  
a(0) <= '0';
```

- ... or more concisely using the following *aggregates* ...

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

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

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

```
a <= (3=>'1', others=>'0');
```



Array Literals Continued

8.9

- Alternatively we could use *positional notation* as follows ...

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

index: 3 2 1 0

- For *character arrays* we can supply a *string literal* ...

```
a <= "1000";
```

- Perhaps surprisingly, character array types actually include **bit_vector** and **std_logic_vector** !
- One reason why **bit** and **std_logic** are defined as enumerated character types is because of the *convenience* of string literal assignments...
 - Dealing with binary words as "0011101...00110" is easy!



Notes:

Composite Data Types

Aggregate assignments such as those on the previous slide can potentially lead to confusion.

When you perform an aggregate assignment, the simulator creates a temporary object from the aggregate expression. Remember that in an array assignment, the indices of the source and target arrays are not referred to during the assignment; they are assigned based on their position within the array.

So if we take the following signal ...

```
signal a : bit_vector(3 downto 0);
```

... then the following aggregate assignment is valid ...

```
a <= (10=>'1', 9=>'0', 8=>'0', 7=>'0');
```

creates a temporary array object

... even although the indices do not correspond. The indices are effectively ignored during the assignment!

Since aggregate expressions are rather confusing, it is not recommended that you use them unless necessary.

However, one useful and non-ambiguous aggregate expression is ...

```
a <= (others => '0');
```

Ver 21.26



Notes:

Composite Data Types

In the next chapter, the **std_logic_vector** type will be covered in more detail.

For now, it is sufficient to note that a **std_logic_vector** is an array of **std_logic** elements.

As you may recall, **std_logic** is an enumerated data type, which can take on one of defined set of literals (including '1', '0', 'Z', 'X', 'U', and so on). Each **std_logic** element can therefore be assigned with a character literal.

In the same way as an array of characters becomes a string...

... a **std_logic_vector** can be assigned with a literal in the form of a string. This makes it very easy to work with binary words. For instance, for the 6-bit **std_logic_vector** signal s, defined below, we can easily assign a 6-bit value as a string literal.

```
signal s : std_logic_vector(5 downto 0);
```

...

```
s <= "010101";
```

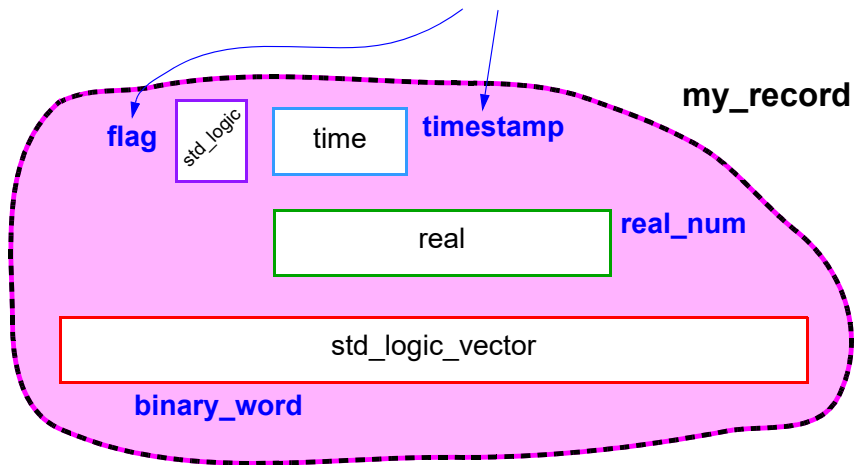
Ver 21.26



Records

8.10

- A **record** is a **collection of elements** which may be of **different types**.
- The **elements** of a record are given **names** rather than **indices**.



Working with VHDL Records

8.11

- A record in VHDL is equivalent to a struct in C or C++. It is a **collection of elements** which may be of **different types**.
- For example, this record contains three elements: two of type **std_logic**, and the third of type **std_logic_vector**.

```
type device_input is record
    CLK : std_logic;
    CE  : std_logic;
    DIN : std_logic_vector(7 downto 0);
end record;
```

- The **elements** can be accessed using the **dot notation** as in C / C++

```
signal a : device_input;
...
a.CLK <= '0';
a.DIN <= "01001001";
```

Notes:

Note that more complex composite types can be constructed, for example a record may contain an array, or an array of records is also possible!

We will not be doing anything so extravagant in this course, though!

Composite Data Types

Ver 21.26

Notes:

Records can also be assigned using aggregates. For example

```
a <= (CLK=>'0', CE=>'1', DIN => "01001001");
```

or by positional assignment ...

```
a <= ('0','1',"01001001");
```

Records are not always supported by synthesis tools so be careful when using them.

Ver 21.26

Operators

8.12

- VHDL has a **standard set of operators**
- Operators are used to:
 - perform arithmetic comparisons
 - form boolean equations
 - form arithmetic equations
- When using operators, you should be aware of operator **precedence**.
- For example $b + c * d$ is interpreted as $b + (c * d)$
- If you want to force an expression to be evaluated a certain way, **use parenthesis!**
- If in doubt ... **use parenthesis!**
- A list of **built-in** VHDL operators is given following slides.



Notes:

Composite Data Types

Logical operators: the logical operators *not*, *and*, *or*, *nand*, *nor*, *xor* and *xnor* are used to describe Boolean logic operations, or perform bit-wise operations, on bits or arrays of bits.

| Operator | Description | Operand Type(s) | Result Types |
|----------|---------------|-------------------------|--------------|
| not | Not | Any Bit or Boolean type | Same Type |
| and | And | Any Bit or Boolean type | Same Type |
| or | Or | Any Bit or Boolean type | Same Type |
| nand | Not And | Any Bit or Boolean type | Same Type |
| nor | Not Or | Any Bit or Boolean type | Same Type |
| xor | Exclusive OR | Any Bit or Boolean type | Same Type |
| xnor | Exclusive NOR | Any Bit or Boolean type | Same Type |

Relational operators: used to test the relative values of two scalar types. The result is always boolean.

| Operator | Description | Operand Types | Result Type |
|----------|-----------------------|-----------------------------------|-------------|
| = | Equality | Any type | Boolean |
| /= | Inequality | Any type | Boolean |
| < | Less than | Any scalar type or discrete array | Boolean |
| <= | Less than or equal | Any scalar type or discrete array | Boolean |
| > | Greater than | Any scalar type or discrete array | Boolean |
| >= | Greater than or equal | Any scalar type or discrete array | Boolean |

Ver 21.26



Matching relational operators: used to test the relative values of two std_ulogic or std_logic types.

These operators allow logic values to be compared, regardless of drive strength. The result is always boolean.

| Operator | Description | Operand Types | Result Type |
|----------|-----------------------|-------------------------|-------------|
| ?= | Equality | std_logic or std_ulogic | Boolean |
| ?/= | Inequality | std_logic or std_ulogic | Boolean |
| ?< | Less than | std_logic or std_ulogic | Boolean |
| ?<= | Less than or equal | std_logic or std_ulogic | Boolean |
| ?> | Greater than | std_logic or std_ulogic | Boolean |
| ?>= | Greater than or equal | std_logic or std_ulogic | Boolean |

Arithmetic operators:

| Operator | Description | Operand Types | Result Type |
|----------|----------------|---------------------------------------------------------------|---------------|
| + | Addition | Any numeric type | Same type |
| - | Subtraction | Any numeric type | Same type |
| * | Multiplication | Left: any integer or floating point type. Right: same type | Same as left |
| * | Multiplication | Left: any physical type. Right: integer or real type. | Same as left |
| * | Multiplication | Left: integer or real type. Right: any physical type. | Same as right |



Notes:

Composite Data Types

| | | | |
|-----|----------------|---------------------------------------------------------------|-------------------|
| / | Division | Left: any integer or floating point type. Right: same type | Same as left |
| / | Division | Left: integer or real type. Right: any physical type. | Same as right |
| mod | Modulus | Any integer type, time (in VHDL-2008) | Same type |
| rem | Remainder | Any integer type, time (in VHDL-2008) | Same type |
| ** | Exponentiation | Left: any integer or floating point type Right: integer | Same as left |
| abs | Absolute value | Any numeric type | Same as left type |

Sign operators: used to specify the sign (either positive or negative) of a numeric object or literal.

| Operator | Description | Operand Types | Result Type |
|----------|-------------|------------------|-------------|
| + | Identity | Any numeric type | Same type |
| - | Negation | Any numeric type | Same type |

Ver 21.26



Miscellaneous operators

| Operator | Description | Operand Types | Result Type |
|----------|------------------------|--------------------------------------------------------------------------|-------------------|
| sll | Shift left logical | Left: Any one-dimensional array of Bit or Boolean Right: integer type | Same as left type |
| srl | Shift right logical | Left: Any one-dimensional array of Bit or Boolean Right: integer type | Same as left type |
| slla | Shift left arithmetic | Left: Any one-dimensional array of Bit or Boolean Right: integer type | Same as left type |
| sra | Shift right arithmetic | Left: Any one-dimensional array of Bit or Boolean Right: integer type | Same as left type |
| rol | Rotate left | Left: Any one-dimensional array of Bit or Boolean Right: integer type | Same as left type |
| ror | Rotate right | Left: Any one-dimensional array of Bit or Boolean Right: integer type | Same as left type |
| & | Concatenation | Any numeric type | Same type |
| & | Concatenation | Any array or element type | Same array type |
| ?? | Condition | Converts a bit or std_logic expression to a boolean one | boolean |



Ver 21.26



Array Attributes

8.13

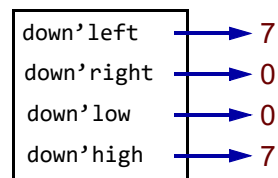
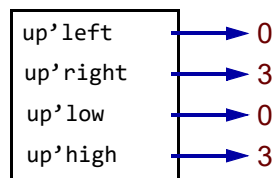
- Array attributes apply to **constrained** arrays
- Consider two arrays ... one **ascending** and one **descending** in range:

```
signal up    : std_logic_vector(0 to 3);    -- ascending
signal down  : std_logic_vector(7 downto 0); -- descending
```

- Based on these declarations, some examples of array attributes are:

up (ascending range)

down (descending range)



- Note that these return **indices of the array**, not the values stored in it!



Notes:

The following attributes are predefined for array types.

signal'left - returns the index of the left most element

signal'right - returns the index of the right most element

signal'high - returns the highest index

signal'low - returns the lowest index

signal'range - returns the range constraint ... used in loop structure or the declaration of new objects

signal'reverse_range - returns the range constraint except literally reversed

signal'length - returns the number of elements in the array

The example shown here is for std_logic_vector signals, but attributes can also be used for any other array types, including array types that you have defined yourself.

Ver 21.26



Summary

8.14

- VHDL includes the option to work with **composite data types** (arrays and records).
- **Arrays** are collections of elements of **the same data type**, whereas **records** are collections which may have **different data types**.
- Arrays are fundamentally important in describing **regular hardware structures** (like memories, repeated elements, buses of wires).
- There are certain **predefined array types**, but it is also common to **define your own types**.
- **std_logic_vector** is an array of **std_logic** - very useful for **buses**!
- A set of **operators** is defined in VHDL, covering logical, relational, and arithmetic operations.
- **Array attributes** are a convenient method of finding out information about an array.



Notes:

Composite Data Types

Ver 21.26



VHDL 9

Std_logic_vector, Signed, and Unsigned Types

Introduction

9.1

- In the last chapter, composite types (arrays and records) were introduced.
- We now look at a particularly useful array type — **std_logic_vector**.
- Two subtypes of std_logic_vector are also introduced:
 - **Unsigned** — for working with unsigned binary numbers.
 - **Signed** — for working with signed 2's complement binary numbers.
- We confirm how to assign and manipulate these array types.

Notes:

Std_logic_vector, Signed, and Unsigned Types

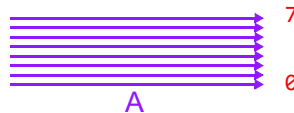
STD_LOGIC_VECTOR

9.2

- Package **std_logic_1164** defines two unconstrained arrays for working with **std_logic** signals in bus form:
 - std_ulogic_vector** — an array of **std_ulogic** elements
 - std_logic_vector** — an array of **std_logic** elements
- Thus we can conveniently declare an array of **std_logic** elements

```
signal A : std_logic_vector(7 downto 0);
```

creates an array of 8 **std_logic** signals :



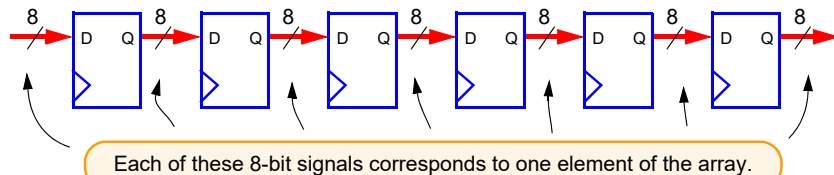
- We can also declare **ports** and **variables** of these types.



Example Array Signals...

9.3

- Suppose that you needed to create a **delay line** (signals passing through a series of registers)...



- Here you would declare an **array type** as follows, then create a signal of that type:

```
-- first create the array type (assume unconstrained)
type delayline is array (natural range <>) of std_logic_vector(7 downto 0);
-- now declare a signal of that type (set to 7 elements)
signal my_delayline : delayline (0 to 6);
```

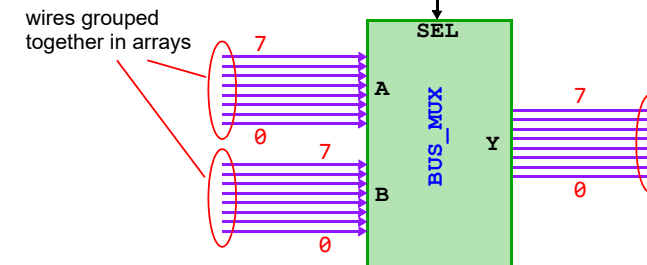


Notes:

Std_logic_vector, Signed, and Unsigned Types

For example, we can conveniently declare an entity which multiplexes two 8-bit buses onto a one 8-bit bus.

```
entity BUS_MUX is
    port(A, B : in std_logic_vector(7 downto 0);
         SEL : in std_logic;
         Y : out std_logic_vector(7 downto 0));
end entity BUS_MUX;
```



Ver 21.24



Notes:

Std_logic_vector, Signed, and Unsigned Types

As the type declaration in this example is an unconstrained array, signals can be created which have any number of delay line elements. However, the width of the bus is fixed to 8 bits, because this forms part of the type declaration.

For example, we could create three different signals from this type declaration, each for different lengths of delayline...

```
-- first create the array type (assume unconstrained)
type delayline is array (natural range <>) of std_logic_vector(7 downto 0);
-- now declare three signals of this type, of different lengths
signal my_delayline : delayline (0 to 6);
signal my_wee_delayline : delayline (0 to 3);
signal my_big_delayline : delayline (0 to 11);
```

If we wanted to be flexible about the width of the signals, this might be achieved using a generic passed down from a higher level in the hierarchy, or a constant set within the current level of hierarchy. The type declaration would then look as follows, where the constant / generic is given by "n":

```
-- create the array type (assume unconstrained, width specified by "n")
type delayline is array (natural range <>) of std_logic_vector(n-1 downto 0);
```

Ver 21.24



Binary Arithmetic in VHDL

9.4

- Often when we are designing digital circuits, particularly for DSP, it is necessary to perform **arithmetic operations**.
- We therefore require the facility to work with **binary numbers**.

110000111.....110101

- Fortunately there is support for this in VHDL!
 - The IEEE **numeric_std** library defines two useful formats: **signed** and **unsigned**.
 - Both **signed** and **unsigned** are subtypes of **std_logic_vector**.
 - This means that we can treat them as **character strings**, which is convenient, but they also have **numeric meaning**.



UNSIGNED and SIGNED Types

9.5

- signed** and **unsigned** types are arrays of **std_logic** elements, **however they also have properties of integers and other scalars!**
- For example let us declare A,B and C as 8-bit **unsigned** arrays:

```
signal A, B, C : unsigned(7 downto 0);
```
- We can access the individual elements of these arrays ...

```
A(0) <= '0';   A(1) <= '1';   .....
```
- Assign string literals to them...

```
A <= "10010100";
```
- But most importantly ... **we can perform arithmetic operations** using simple operators!

```
A <= B + C;
```

```
A <= B - C;
```

```
A <= B * C;
```



Notes:

Std_logic_vector, Signed, and Unsigned Types

There are also other libraries available for arithmetic, such as **std_logic_arith**, which was developed by the synthesis tools developer, Synopsys. This provides similar functionality. However, we will be working only with the **numeric_std** library in this course.

We can include this package by making an appropriate addition to the library declarations at the top of the VHDL file:

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
```



Ver 21.24



Notes:

Std_logic_vector, Signed, and Unsigned Types

Unsigned and signed types have properties of both scalar and array types. They are in fact arrays of **std_logic** elements, however many of the arithmetic operators have been **overloaded** for the signed and unsigned types. What's more, these types and their associated operators are synthesisable. This allows a design to be described at a reasonably high level, while having the flexibility of being able to access the individual logic elements.

As defined in package **numeric_std** and also in **std_logic_arith**, unsigned and signed are:

```
type unsigned is array (natural range <>) of std_logic;
type signed is array (natural range <>) of std_logic;
```

Note that when using signed and unsigned types, you should use **either** the **std_logic_arith** package **or** the **numeric_std** package, **but not both!** This is because they contain almost exactly the same set of declarations and are essentially intended for the same purpose. If you try to use both of these then you will almost certainly have problems compiling your code!

std_logic_arith is a proprietary package the copyright of which belongs to Synopsys Incorporated.

numeric_std is a direct replacement for **std_logic_arith**, but is standardised by the IEEE.

We will be using **numeric_std** in this course.

Ver 21.24



Range Limits and Bit Indices

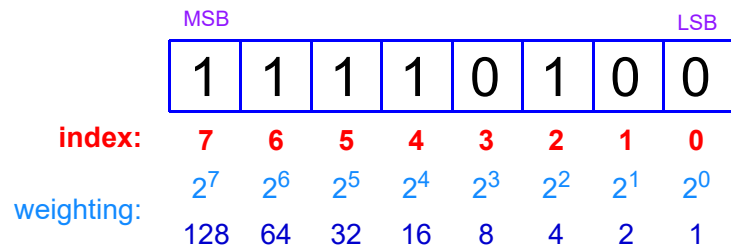
9.6

- When working with array types that represent binary numbers, the specified range is almost always a descending, ending in zero. Why?

```
signal A, B, C : unsigned(7 downto 0);
```

descending range

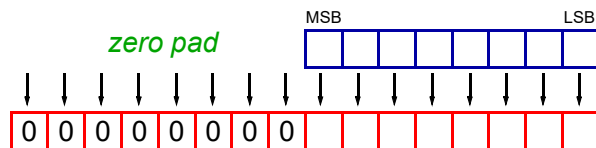
- This is because of how unsigned and signed numbers are composed. The indices used in the signal definition match the bit indices of the number representation. For instance, take this unsigned example:



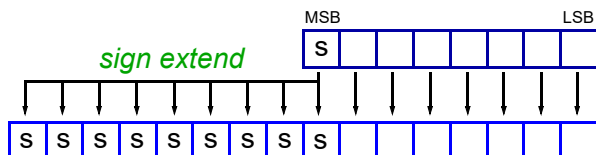
Resizing UNSIGNED and SIGNED

9.7

- The **resize** operator illustrates the difference between these types
- When using resizing to create a larger array:
 - for **unsigned**, **resize()** simply pads with '0's...



- for a **signed** number, **resize()** performs sign extension...



Notes:

Std_logic_vector, Signed, and Unsigned Types

Using a descending range, ending in zero, matches the bit indices, and makes it easier to manipulate signals or variables that represent numbers, using the signed or unsigned types.

This also applies when a **std_logic_vector** is being used to represent a numeric value, as well as when using the **signed** and **unsigned** types.

Ver 21.24



Std_logic_vector, Signed, and Unsigned Types

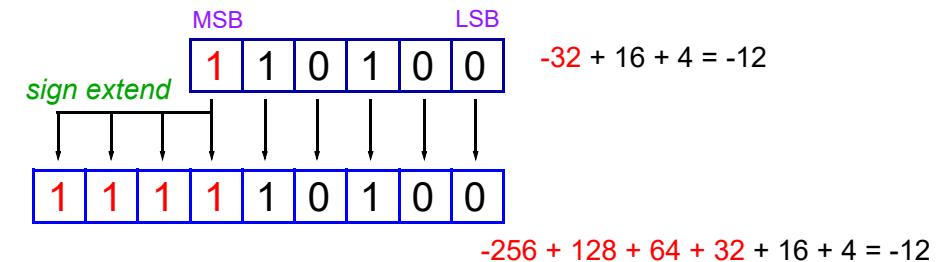
Notes:

The **resize** operator is *overloaded* for the signed and unsigned types. That means that there are actually two functions called **resize()**: one which operates on signed types, and the other which operates on unsigned types. The VHDL compiler will choose the appropriate function depending on the type of argument you provide.

Other examples of functionality contained within the numeric packages are boolean, comparison, arithmetic and concatenation operators, shift functions and type conversions. For further information on these packages see a VHDL textbook or try an internet search. The book "VHDL for Logic Synthesis" by Andrew Rushton has two very good chapters on the differences between the two packages.

For **unsigned**, it is quite intuitive that adding zeros at the left hand side of the number (i.e. adding 0's for Most Significant Bits, or MSBs) will not change the value represented.

For **signed**, it is perhaps a little less obvious that the number remains the same when copying the old MSB into the new MSB positions, when the MSB is a 1 (when it's a 0, the operation is identical to the unsigned case). Let's look at an example of sign-extending a negative-valued signed number.



Ver 21.24



Array Conversions

9.8

- There is support for making conversions between certain common array formats. Of particular interest:
 - **std_logic_vector** to **signed / unsigned**
 - **signed / unsigned** to **std_logic_vector**
 - **std_logic_vector** to **bit_vector**
 - **bit_vector** to **std_logic_vector**

```
my_std_logic_vector <= std_logic_vector(my_bit_vector);
my_bit_vector <= bit_vector(my_std_logic_vector);
my_unsigned <= unsigned(my_std_logic_vector);
my_std_logic_vector <= std_logic_vector(my_unsigned);
```

- If dealing with numbers, we can also convert between **integer** and **signed / unsigned** array formats.



Summary

9.9

- **std_logic_vector** is an array of **std_logic** — very useful for **buses**!
- It is also possible to define array types that compose elements of **std_logic_vector**. You could think of this as an array of arrays!
- The **numeric_std** library defines subtypes of **std_logic_vector** (**signed** and **unsigned**) which can be used for **arithmetic**.
 - This enables arithmetic operators for addition, subtraction, and multiplication to be used directly in the code.
- **Signed** and **unsigned** types are normally defined with a **descending range**, to match the format of **signed** and **unsigned** numbers.
- The **resize()** function is useful, and permits the dimensions of a signed or unsigned number to be altered.
- It is possible to convert between integer and signed / unsigned, and between signed / unsigned and **std_logic_vector**.



Notes:

Std_logic_vector, Signed, and Unsigned Types

Note that we cannot convert directly between integer and **std_logic_vector**, because there are no conversion functions defined for this. However, we can make the conversion in two steps.

For example, suppose we wanted to convert the **integer** 52 into a **std_logic_vector**.

- The first point is to decide upon the size, or wordlength, of the binary representation (how many bits?).
- The second point to consider is whether the number should be formatted as **unsigned** or **signed** (2's complement), as these would correspond to different sequences.

Say we decided that an 8-bit unsigned representation was desired. The VHDL code would be as follows... note that we must first convert from integer to unsigned (specifying the wordlength), and then from unsigned to **std_logic_vector**. This can be written in one line.

```
my_slvector <= std_logic_vector(to_unsigned(52, 8));
```

The syntax is similar for converting to signed numbers; we used instead the function:

to_signed(argument, length);

If we want to convert in the opposite direction, we must again take two steps. For example, below we take the **std_logic_vector** "11111111" and convert it to a signed representation, and then to an integer.

```
my_integer <= to_integer(signed("11111111"));
```

Ver 21.24

Notes:

Std_logic_vector, Signed, and Unsigned Types

Ver 21.24



VHDL 10

Concurrent Statements

VHDL Statements

10.1

- In VHDL there are two types of statement:
- Concurrent Statements:**
 - Concurrent statements appear in the **concurrent area** of an architecture
 - Concurrent statements are **executed simultaneously**
 - The **order** of a list of concurrent statements **does not matter**
- Sequential Statements:**
 - Sequential statements appear in the bodies of **processes**, **functions** and **procedures**
 - Sequential statements are **executed in order**
 - The **order** of a list of sequential statements **does matter**

Notes:

Concurrent Statements

The testbench architecture below uses sequential statements to generate the clock and reset signals within two separate processes. However, processes themselves are concurrent statements. Component instantiations are concurrent statements too.

```

-----
clock_gen: process
begin
    while now <= 3000ns loop
        CLK_TB <= '1'; wait for 5 ns;
        CLK_TB <= '0'; wait for 5 ns;
    end loop;
    wait;
end process;
-----
reset_gen: process
begin
    RST_TB <= '0'; wait for 5ns;
    RST_TB <= '1'; wait for 15ns;
    RST_TB <= '0';
    wait;
end process;
-----
inst: counter
port map ( CLK => CLK_TB, RST => RST_TB, Q => DATA_OUT );
-----

```

Diagram illustrating the structure of concurrent and sequential statements in the testbench architecture:

- The **clock_gen** and **reset_gen** processes are grouped by a large blue bracket on the right, labeled **concurrent statements**.
- Inside each process, the loop and wait statements are grouped by a red bracket on the right, labeled **sequential statements**.

Concurrent Statements

10.2

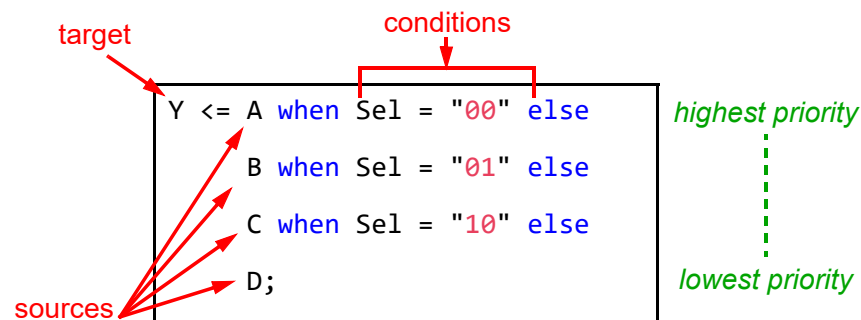
- Concurrent statements include the following:
 - Concurrent Signal Assignments
 - Conditional Signal Assignments**
 - Selective Signal Assignments**
 - Component Instantiations
 - Procedure calls
 - Generate Statements**
 - Processes
- This set of notes covers the statement types highlighted above.



Conditional Signal Assignments

10.3

- A **conditional signal assignment** allows the source of a signal assignment to depend on certain conditions.
- Conditional signal assignments are the **concurrent equivalent** of **if-then-else** statements in other languages.
- The expressions are **prioritised** based on their order in the statement:



Notes:

Concurrent Statements

Ver 21.211



Notes:

Concurrent Statements

The sources of a conditional signal assignment can be anything that you can assign with a standard concurrent signal assignment. Therefore as well as assigning other signals you can assign **literals**, **constants** and **aggregates**.

```
entity mux1 is
    port (Sel      : in  std_logic_vector (1 downto 0);
          A, B, C, D : in  std_logic_vector (3 downto 0);
          Y        : out std_logic_vector (3 downto 0));
end mux1;

architecture mux1_arch of mux1 is
begin
    Y <= A when Sel = "00" else
        B when Sel = "01" else
        C when Sel = "10" else
        D when others;
end mux1_arch;
```

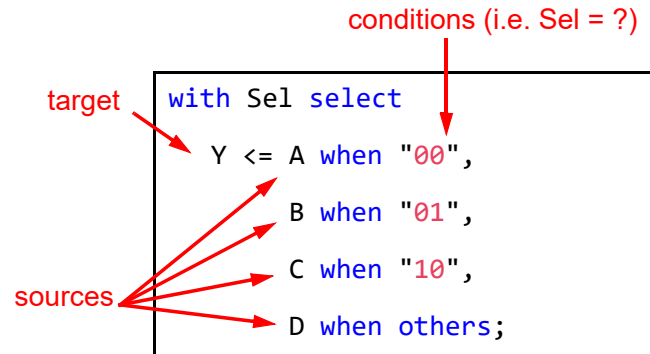
Ver 21.211



Selective Signal Assignments

10.4

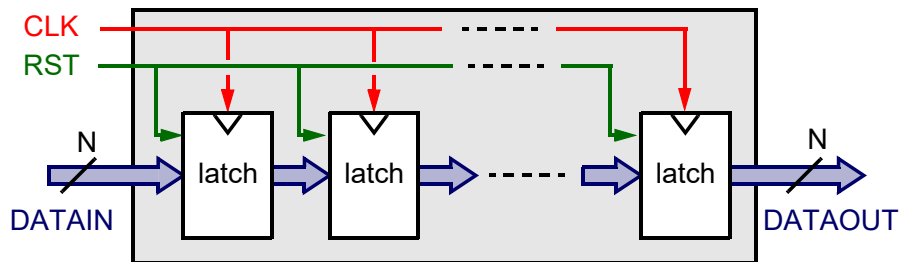
- **Selective signal assignments** are **not prioritised**.
- The condition can only be on **one object** - below it is **Sel**
- The **others** keyword should be used to cover all other unspecified cases.



Generate Statements

10.5

- Generate statements **instantiate multiple instances** of a component.
- They are useful for creating **regular hardware structures**.
- Consider a **delay line** with a total delay of **L** samples
- This will be constructed from a series of **L** latches as shown:



- There are two forms of generate ... **for-generate** and **if-generate**.

Notes:

Concurrent Statements

Selective assignments often generate simpler hardware because no priority is implied in the signal assignments. In addition, the condition is on a single object.

Selective assignments may also include ranges, as in the example below.

```
entity selector is
    port (Sel : in std_logic_vector (1 downto 0);
          A, B : in std_logic_vector (3 downto 0);
          Y : out std_logic_vector (3 downto 0));
end selector;

architecture selector_arch of selector is
begin
    with to_integer(unsigned(Sel)) select
        Y <= A when 0 to 2,
            B when others;
end selector_arch;
```

Ver 21.211

Notes:

Concurrent Statements

Generate statements are a convenient way to create multiple instances of concurrent statements. They are almost always used to perform multiple component instantiations.

As a result, generate statements are useful for generating regular hardware structures. One example is creating an *N*-bit adder circuit from a number of full adders. It would be inefficient if you had to write a component instantiation for every single full adder in the circuit! The component instantiations would all look almost exactly the same, just with a change of index in an array of signals.

This is where generate statements come in!

We shall consider the code for a delay line circuit. Delay lines are required in digital filters to store a finite number of past input sample values. A delay line is essentially just a shift register that stores a number of bits at each stage in order to represent the samples of a digital signal.

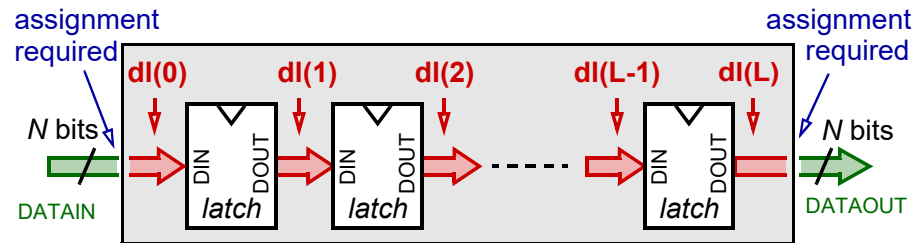
NOTE: In the following example the CLK and RST signals have been omitted for brevity. In the full design they must be included for correct operation of the circuit.

Ver 21.211

For-Generate Statement

10.6

- In a **for-generate**, every instantiation follows the same pattern.
- This is similar to a **for loop** ... every iteration instantiates a component
- We require an **array of L+1 signals** for the interconnects
- We can create **L** component instances using a **for-generate**...



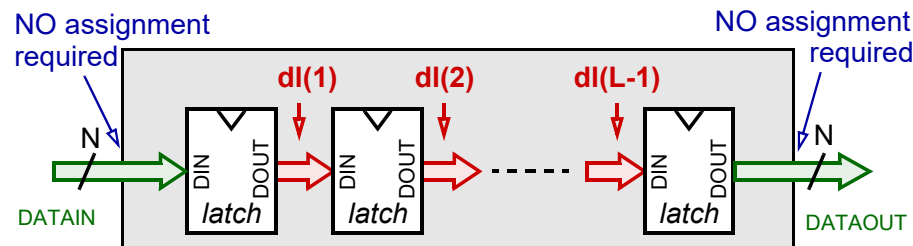
- Signal assignments are required to connect **dl(0)** to **DATAIN** and **dl(L)** to **DATAOUT**



If-Generate

10.7

- **if-generate** statements allow you to **condition the concurrent assignments** in a generate statement, based on the **loop variable**.
- In our example we can use this to connect the **first** and **last** latches directly to the **DATAIN** and **DATAOUT** ports, respectively.
- The result is that we only require an **array of L-1 signals** for the interconnects...



Notes:

Concurrent Statements

Before instantiating the latch components, we must create signals with which to connect them together. The code below creates an array of signals called **dl** which we will use as interconnects between the latch elements. Note that we require **L+1** elements in the array, since every latch has an output which forms the input to the next latch, and we require a signal to input to the first latch instance.

```
type DelayLine_T is array (natural range <>)
of signed(N-1 downto 0);

signal dl : DelayLine_T(0 to L);
```

The **generate statement** creates the actual latch instances...

```
-- connect the ends of the delay line to the IO ports
DL_GEN : for i in 0 to L-1 generate

    DL_ELEM_GEN: latch
        port map ( DIN=>dl(i), DOUT=>dl(i+1) );

end generate;

-- connect the first and last array elements to the IO ports
dl(0) <= DATAIN;
DATAOUT <= dl(L);
```

Ver 21.211



Notes:

Concurrent Statements

An example of using **if-generate** statements is shown below. The if-generate statement allows us to change the instantiation depending on the current array index. As shown by the example, the condition can be used to pick out a specific index or a range of indices over which the instantiation follows the same pattern.

Below, this feature has been used to connect the first and last elements directly to the ports of the entity (i.e. DATAIN and DATAOUT are the input/output port names), rather than creating two extra elements in the signal array.

```
DL_GEN: for i in 0 to L-1 generate

    DL_GEN0: if i = 0 generate -- generate first latch
        DL_ELEM_GEN: latch
            port map ( DIN=>DATAIN, DOUT=>dl(i+1) );
        end generate;

    DL_GEN1: if (i > 0) and (i < (L-1)) generate
        DL_ELEM_GEN: latch
            port map ( DIN=>dl(i), DOUT=>dl(i+1) );
        end generate;

    DL_GEN2: if i = L-1 generate -- generate last latch
        DL_ELEM_GEN: latch
            port map ( DIN=>dl(i), DOUT=>DATAOUT );
        end generate;

end generate;
```

Ver 21.211



Summary

10.8

- There are two different types of VHDL statements:
 - **Concurrent** statements — the order does not matter; all statements in a concurrent region are executed at the same time.
 - **Sequential** statements — these are executed line by line, from top to bottom within a sequential section of the code.
- The concurrent statements considered were: **signal and variable assignments**, **conditional and selective assignments**, and the **generate** statement.
- **Generate statements** are particularly useful for creating regular hardware structures. They can be used to create **multiple instances of concurrent statements**, primarily component instantiations.



Notes:

Concurrent Statements

Ver 21.211



VHDL 11

Sequential Statements

VHDL Statements (recap)

11.1

- In VHDL there are two types of statement:
- Concurrent Statements:**
 - Concurrent statements appear in the **concurrent area** of an architecture
 - Concurrent statements are **executed simultaneously**
 - The **order** of a list of concurrent statements **does not matter**
- Sequential Statements:**
 - Sequential statements appear in the bodies of **processes**, **functions** and **procedures**
 - Sequential statements are **executed in order**
 - The **order** of a list of sequential statements **does matter**

Notes:

Sequential Statements

The testbench architecture below uses sequential statements to generate the clock and reset signals within two separate processes. However, processes themselves are concurrent statements. Component instantiations are concurrent statements too.

```

-----
clock_gen: process
begin
    while now <= 3000ns loop
        CLK_TB <= '1'; wait for 5 ns;
        CLK_TB <= '0'; wait for 5 ns;
    end loop;
    wait;
end process;
-----
reset_gen: process
begin
    RST_TB <= '0'; wait for 5ns;
    RST_TB <= '1'; wait for 15ns;
    RST_TB <= '0';
    wait;
end process;
-----
inst: counter
port map ( CLK => CLK_TB, RST => RST_TB, Q => DATA_OUT );
-----

```

Diagram illustrating the structure of the testbench architecture:

- The **clock_gen** and **reset_gen** processes are grouped by a red bracket labeled **sequential statements**.
- The **inst** component instantiation is grouped by a blue bracket labeled **concurrent statements**.

Sequential Statements

11.2

- Sequential VHDL **does not necessarily describe sequential logic ...**
- ... it can describe either **combinational** or **sequential** logic!
- Sequential statements include the following:
 - **Signal and Variable Assignments**
 - **if-then-else statements**
 - **Case statements**
 - **For loops**
 - **While loops**
 - **Infinite loops**



Signal and Variable Assignments

11.3

- **Signal** and **variable** assignments can be performed **inside a process**... however there are some important point to remember here:
- There is a fundamental difference between **signal** and **variable** assignments:

"Variable assignments occur immediately, whereas signal assignments are scheduled to occur at some point in the future."
- Signals are effectively **constant** inside a process. You can perform a signal assignment, however... the signal only takes on the new value when the process is **suspended**.
- A process **suspends** when it **completes execution** or reaches a **wait statement**.
- The implications of this are demonstrated in the following three segments of VHDL code, which implement an 8-bit counter.



Notes:

Sequential VHDL is very similar to code in conventional programming languages. The code is executed as a series of steps, one after the other. This is obviously very different from concurrent VHDL where we have to imagine every statement essentially executing simultaneously.

Concurrent VHDL is easy to synthesise compared to sequential VHDL, since hardware is inherently parallel and consists of concurrent processes which interact with each other. Sequential VHDL is much easier for humans to write and understand, however it is difficult to interpret as hardware, due to the inherent parallelism of hardware.

When synthesising sequential VHDL, the synthesis tools produce a concurrent equivalent of the sequential code. This is only possible if the sequential VHDL follows certain rules and coding styles.

Sequential Statements

Ver 21.214



Notes:

Sequential Statements

```
architecture counter_arch1 of counter is
    signal REG : unsigned(7 downto 0) := "00000000";
begin

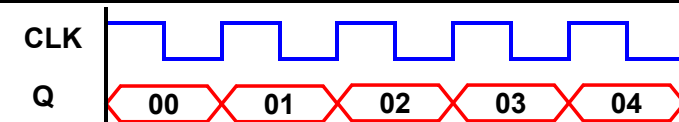
    process (CLK)
    begin

        if rising_edge(CLK) then
            REG <= REG + 1;
        end if;

    end process;

    Q <= std_logic_vector(REG); -- concurrent signal assignment

end counter_arch1;
```



In this case a signal is used to store the current value of the counter. A concurrent assignment is used to transfer the value in the register to the output port. (Note that Q is an output port on the entity declaration). A change of REG triggers the concurrent signal assignment to execute. This code works as expected / intended.

Ver 21.214



```

architecture counter_arch2 of counter is
    signal REG : unsigned(7 downto 0) := "00000000";
begin

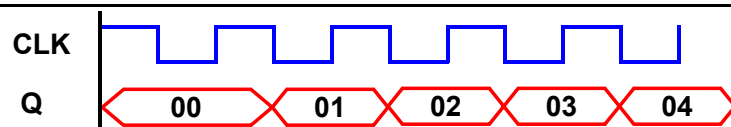
    process (CLK)
    begin

        if rising_edge(CLK) then
            REG <= REG + 1; -- this assignment doesn't happen ...
        end if;           -- ... until the process is suspended

        Q <= std_logic_vector(REG); -- REG is not updated yet!

    end process;
end counter_arch2;

```



In this case the signal assignment has been moved inside the process. As can be seen from the timing diagram, the output now changes on the **falling clock edge**. This is because REG is not updated until after the process is suspended. This problem could be fixed by adding REG to the process sensitivity list.



If-then-else statements

11.4

- **If-then-else** statements are the most common way to control the program flow in sequential VHDL.
- **Priority** is inferred by the **order** of the conditions / statements.
- The conditions must result in a **boolean** value.

```

if first_condition then
    statements
elsif second_condition then
    statements
else
    statements
end if;

```

conditions
(must result in **boolean**)



Notes:

```

architecture counter_arch3 of counter is
begin

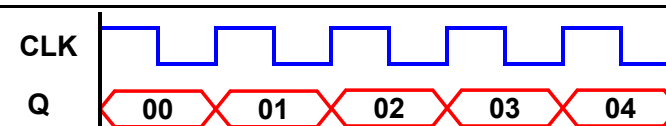
    process (CLK)
        variable REG : unsigned(7 downto 0) := "00000000";
    begin

        if rising_edge(clk) then
            REG := REG + 1; -- this happens immediately
        end if;

        Q <= std_logic_vector(REG);

    end process;
end counter_arch3;

```



In this example, the counter's current value is held in a variable that is local to the process. Since it is a variable the assignment happens immediately. Therefore the value assigned to Q is the updated value of REG.

Ver 21.214



Notes:

The conditions in an **if-then-else** statement must evaluate to a BOOLEAN value. The condition can either be specified as a boolean object (for example **if** (RST = '1') **then** ...), or a comparison between two objects (e.g. **if** (count > 100) **then** ...).

The objects used to form the conditions can be signals, variables or constants.

The sequential statements contained within the **if** statement are evaluated **if and only if**, the condition is TRUE. If the condition is not true, execution skips to the end of the **if** statement and resumes evaluation with the first statement following the end.

The **elsif** and **else** parts of the statement are optional. There can also be more than one **elsif** part. An example of using the **if-then-else** statement is given below:

```

if rising_edge(clk) then
    if (RST = '1') then
        REG <= (others => '0');
    else
        REG <= DATAIN;
    end if;
end if;

```

Ver 21.214

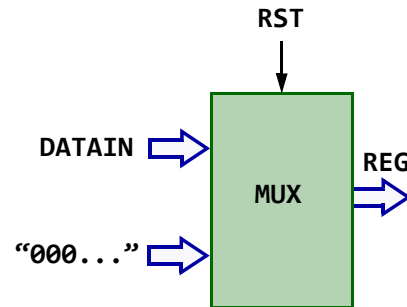


If-then-else Synthesis

11.5

- **If statements** are synthesised using **multiplexers**
- The conditions are used to generate the MUX **selection signal**
- A **multiplexer** is required for each signal or variable which is assigned within the **if-then-else** statement.

```
if (RST = '1') then
    REG <= (others => '0');
else
    REG <= DATAIN;
end if;
```

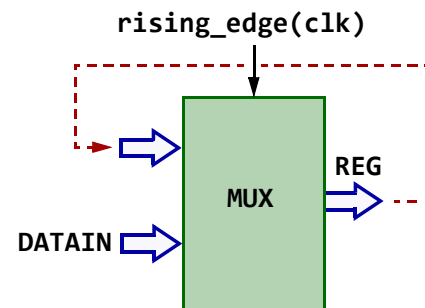


Latch Inference

11.6

- Consider the **incomplete if statement** shown below (note it has no **else** part!)
- What happens if the condition is tested and found to be **"false"**?
 - In this case the value of **REG** should **stay the same**
 - Therefore a **latch** is required to store the value of **REG**...

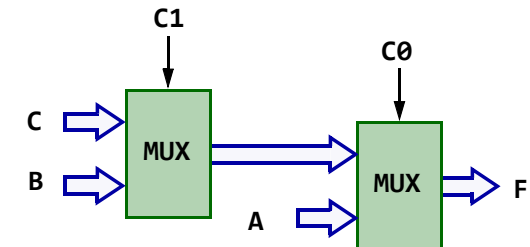
```
if rising_edge(clk) then
    REG <= DATAIN;
end if;
```



Notes:

The priority in an if-then-else statement is implemented by concatenating multiplexers. This is shown in the example below. The rightmost multiplexer represents the highest priority condition. Only if C0 is found to be '0' are the other conditions taken into account.

```
if (C0 = '1') then
    F <= A;
elsif (C1 = '1') then
    F <= B;
else
    F <= C;
end if;
```



Ver 21.214



Sequential Statements

Notes:

For an incomplete if statement, the synthesis tool needs to decide what to do when the condition is not true. In simulation the behaviour would be to leave the signal or variable unchanged. So in the example on the previous slide, REG would still hold the same value from a previous assignment.

In order to implement this functionality in hardware, the synthesis tool must create some storage to hold the signal value in case it is required again when the condition is not true. When there are undefined conditions for a particular output signal in an if statement, then storage, in the form of a latch, is *inferred* during synthesis.

A latch is synthesised from an HDL model when a signal needs to hold its value over time.

Ver 21.214



Case Statements

11.7

- The **case statement** can be used as an alternative to the **if statement**
- The **control expression** is compared to each of the **test expressions**, and the appropriate statements are then executed...

```
case control_expression is
    when test_expression1 =>
        statements
    when test_expression2 =>
        statements
    when others =>
        statements
end case;
```



For Loops

11.8

- The execution of a **for loop** is controlled by the **loop parameter**.
- The loop parameter is **implicitly declared** and **cannot be referenced outside of the loop**.
- Also, it **cannot be modified inside the loop**... actually it is effectively a constant during each iteration.
- The range can be either **ascending** or **descending**.

```
for i in 0 to 3 loop
    F(i) <= A(i) and B(3-i);
    V := V xor A(i);
end loop;
```

Ascending Range

```
for i in 3 downto 0 loop
    F(i) <= A(i) and B(3-i);
    V := V xor A(i);
end loop;
```

Descending Range

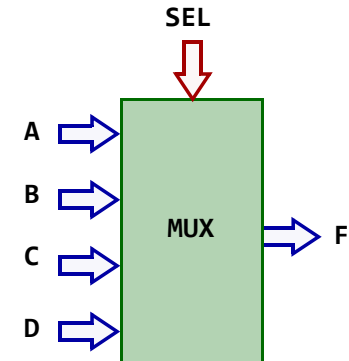


Notes:

An example of a multiplexer implemented using the case statement is shown below. The case statement has the advantage for synthesis that no priority is implied by the structure. So for example although the "00" case is the first condition in the example below, it does not have a higher priority than any of the other cases.

The "others" keyword is used to cover all other cases that have not been explicitly specified in the statement. In the example below, the others case would not be meaningful for synthesis, because all of the cases which have a real meaning in terms of hardware have been covered by the explicit clauses.

```
case SEL is
    when "00" =>
        F <= A;
    when "01" =>
        F <= B;
    when "10" =>
        F <= C;
    when "11" =>
        F <= D;
    when others =>
        F <= 'X';
end case;
```



Ver 21.214



Notes:

What happens if the loop parameter is given the same name as an object that exists outside of the loop? The answer is that, within the loop, the loop parameter effectively hides the other object with the same name. Outside of the loop, the loop parameter does not exist, so it is not possible to reference it!

```
process (A, B)
    variable i : std_logic;
begin
    for i in 3 downto 0 loop
        F(i) <= A(i) and B(3-i);
        V := V xor A(i);
    end loop;
    i := not i;
end process;
```

variable i

loop parameter i

loop parameter i

variable i

Ver 21.214

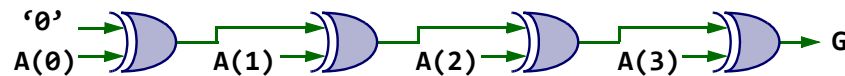
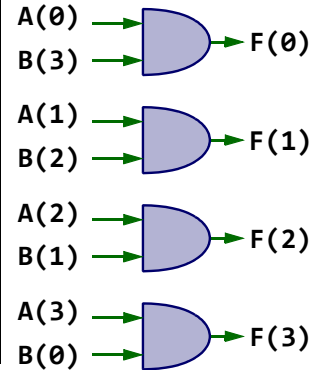


For Loop Synthesis

11.9

- for loops** are synthesised by **unrolling the loop** and generating **multiple copies** of the hardware described within the loop

```
process (A, B)
  variable V : std_logic;
begin
  V := '0';
  for i in 3 downto 0 loop
    F(i) <= A(i) and B(3-i);
    V := V xor A(i);
  end loop;
  G <= V;
end process;
```



While Loops

11.10

- A **while loop** executes **while a certain condition is true**.
- In a while loop, an **index** is **not automatically declared**, so if you want to use an index you must declare it yourself.
- The condition is evaluated at the **start** of each iteration.
- When the condition is **false** the loop execution **terminates**.

```
from_in_to_out:           -- optional label
while index < 8 loop

  ray_out(index) <= ray_in(index);
  index := index + 1;

end loop from_in_to_out;
```

- While loops are **not synthesisable** since it is not generally known how many iterations the loop will execute for!

Sequential Statements

Notes:

When a **for loop** is synthesised, one copy of the hardware is generated for each iteration. Thus there is a copy of the hardware for each possible value of the loop parameter.

If a **for loop** is to be synthesisable, then its range must be a constant. This is because the synthesis tool needs to know how many copies of the hardware to generate. If the loop limits are a function of a variable or a signal, then the number of iterations is not known until run time, and consequently it is not possible to synthesise the hardware.

Ver 21.214

Sequential Statements

Notes:

Although while loops are not synthesisable, they are very useful in test benches for generating test vectors. For example, the following code could be used to generate a clock signal:

```
process
begin
  while now <= 200ns loop
    CLOCK <= '1';
    wait for 5 ns;
    CLOCK <= '0';
    wait for 5 ns;
  end loop;
  wait;
end process;
```

If test vectors for a testbench were being read from a file, a while loop would be used to read data values and check for the end of the file.

Ver 21.214

Infinite Loop

11.11

- An **infinite loop** does not necessarily include a termination case...

```
L: loop
    statements_to_repeat
end loop L;
```

- However there are optional **keywords** for **controlling the loop**
 - exit** — exits the loop altogether
 - next** — continues to the next iteration of the loop
- Both **exit** and **next** can be **conditioned** — i.e. only execute under certain conditions.
- The **exit** and **next** statements can be used in any type of loop, not just the infinite loop!



Summary

11.12

- In this section we have introduced **sequential** VHDL statements, which can be written only within sequential sections of code (like processes).
- The differences between **signal** and **variable** assignments were explored in the context of a counter example.
 - We saw that **signal assignments** do not take place until the process **suspends**, whereas
 - Variable assignments are immediate.
- The sequential statements considered were: **if-then-else statements**, **case statements**, **for loops**, **while loops** and **infinite loops**.
- Although sequential VHDL does not naturally describe hardware, it is **synthesizable** if certain rules are followed.



Notes:

This loop is an **infinite loop** and is functionally equivalent to the **while loop** for generating a clock signal. The **exit** keyword is used to terminate the loop when the simulation time exceeds 200ns.

```
loop
    exit when now > 200ns;
    CLOCK <= '1';
    wait for 5 ns;
    CLOCK <= '0';
    wait for 5 ns;
end loop;
```

This code shows two nested **for loops**. Both loops have been given labels. Since these are **for** loops they will terminate when the loop parameters exceed the specified range limits. However, exit statements have also been included. Although both exit statements are within the inner loop, the first one applies to the inner loop and the second to the outer loop. This is because they use labels to specify which loop they apply to.

```
L1 : for i in 0 to 7 loop
    L2 : for j in 0 to 7 loop
        C := C + 1;
        exit L2 when A(j) = B(i);
        exit L1 when B(C) = 'U';
    end loop L2;
end loop L1;
```



Ver 21.214

Notes:

Sequential Statements

Sequential Statements

Ver 21.214



VHDL 12

VHDL Processes

Introduction

12.1

- **Processes** are one of the key building blocks in VHDL.
- These can be used both for *designing circuits (synthesisable VHDL)* and *testbenches (non-synthesisable VHDL)*.
- In this presentation we look at some important aspects of processes:
 - The *sensitivity list*
 - *Local declarations*
 - *Process execution*
 - *Synthesisable* processes...
 - Describing *combinatorial* circuits
 - Describing *sequential* circuits
 - *Non-synthesisable* processes

Notes:

VHDL Processes

Outline of a Process

12.2

- The general structure of a process is as follows...

```
process_label : process (sensitivity list) is
-- variable declarations to go here
-- other local declarations to go here

begin

-- ...
-- sequential statements
-- more sequential statements...
-- ...

end process;
```

- Note that the **process label** and **sensitivity list** are optional.



Local Declarations

12.3

- Several different types of local declaration can be made inside a process:
 - Constants**
 - Signals**
 - Subprograms** (procedures and functions)
 - Variables**
- All of these quantities can only be used by the sequential statements within the process; they **cannot be accessed outside the process**.
- Variables** are especially useful for storing **intermediate values** during the execution of the process.
- Subprograms** include procedures and functions (not part of this class; a more advanced topic).



Notes:

VHDL Processes

The process label is a name that you can choose to give a process to distinguish it from other processes. It is useful to take advantage of this facility, as the result is more readable code, and the process can also be more easily identified during a simulation (for example for monitoring variables inside the process). If you do not explicitly label a process, then the compiler will assign it with one, but this would usually be a short and meaningless name.

The choice of whether to include a sensitivity list has a more fundamental effect on the functionality of the process, and this aspect will be discussed in detail shortly.

Ver 21.221



Notes:

VHDL Processes

To confirm where these local declarations are made, the declaration region of a process is similar to architectures (and also functions and procedures).

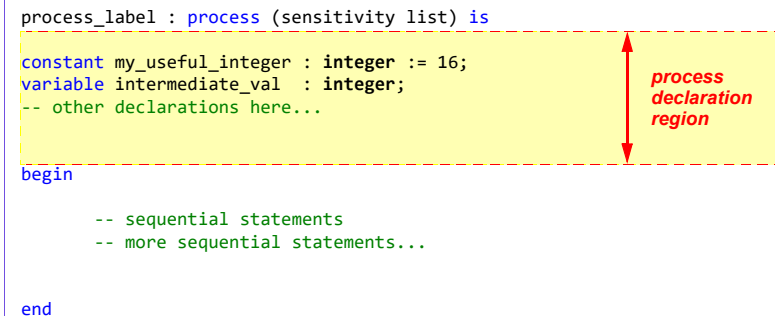
The declaration region is the section of code between the first line of the process, and the "begin" keyword.

```
process_label : process (sensitivity list) is
constant my_useful_integer : integer := 16;
variable intermediate_val : integer;
-- other declarations here...

begin

-- sequential statements
-- more sequential statements...

end
```



Ver 21.221



Process Execution

12.4

- Processes execute **sequentially** (line-by-line from top to bottom).
- Sections of code** may however be **repeated** (by including them within a **loop**).
- Suspension** of a process occurs due to one of two things:
 - Execution reaches a **wait** statement.
 - The execution of the process **completes**, i.e. the last line of the process is executed.
- As noted in a previous chapter, signal assignments can be included within a process, but these assignments do not actually take place (i.e. the signals do not take on their new values) until the process is **suspended**.
- Many processes can be included in a single design, and these will operate **concurrently to each other**.



The Sensitivity List

12.5

- The process sensitivity list is **optional**.
- Processes **without sensitivity lists** behave very differently to those **with sensitivity lists**. We can think of them as two distinct classes.
 - Processes **with sensitivity lists** are **synthesisable** (used to design **circuits**)
 - Processes **without sensitivity lists** are generally **non-synthesisable** (used to design **testbenches**)
- The **sensitivity list** is like a list of “**watched signals**” - a change in value on any of these signals causes the process to execute.
- The **choice of signals** to include in the sensitivity list depends on the intended functionality.
- Processes without sensitivity lists (used in testbenches) simply **begin execution at the start of a simulation**.



Notes:

Given that there are two means of entering suspension, let us consider two examples where signal assignments are made within a process. The first is in a piece of code intended for synthesis, and the second is in a testbench (not intended for synthesis). Remember that wait statements are non-synthesisable statements used in testbench design.

```
synth : process (A,B,..., F) is
begin
  G <= A and B;
  -- other statements...
end process;
```

signal assignment scheduled

signal assignment made
G only takes on the new value when the process suspends (completes).

```
unsyn : process is
begin
  G <= A and B;
  wait for 50ns;
  -- other statements...
end process;
```

signal assignment scheduled

signal assignment made
The wait statement causes the process to suspend; G is updated.

Ver 21.221



Notes:

Just to explicitly confirm what is meant by the sensitivity list...

The sensitivity list is a list of signals given after the VHDL keyword process, in brackets. This is a list of all signals the process is “sensitive” to, i.e. on which an event will cause the process to execute.

For example, if the sensitivity list included the signals reset, shift, B, D, and M, it would look like the following...

```
process_label : process (reset, shift, B, D, M) is
```

sensitivity list

NOTE ABOUT WAIT STATEMENTS AND SENSITIVITY LISTS

In this course we present a VHDL coding style wherein we include a sensitivity list in any processes we intend for synthesis. We also avoid the use of wait statements, apart from in non-synthesisable code.

Both of these conventions can be avoided using an alternative coding style, which has some complicated rules as to its use and acceptability for synthesis! Hence in the interests of avoiding confusion, this course will present one style only.

Ver 21.221



Sensitivity List Example (AOI)

12.6

- The **sensitivity list** is a list of **watched signals**.
- A **change in value** on any of these signals will cause the process to **execute**.
- For example, we could implement the AOI functionality considered in earlier chapters, using a **process**.
- The process should execute if **any of the input signals changes value**, because this could change the value of the output.
- Hence the **sensitivity list** should include signals **A, B, C, and D**.

sensitivity list
↓

```
AOI_p : process (A,B,C,D) is
  variable intA : std_logic;
  variable intB : std_logic;

  begin

    intA := A and B;
    intB := C and D;
    F <= not (intA or intB);

  end process;
```



Sensitivity List for Combinatorial Circuits 12.7

- Combinatorial circuits are those with **no memory**, whose output depends **only on the current inputs**.
- As in the AOI example, the sensitivity list for a **combinatorial circuit** must contain the **full set of inputs**.
- A change on **any of the inputs** may cause a change on the output. The **sensitivity list** ensures that the output is re-evaluated.

```
combinatorial_process: process (all input signals)
  begin

    -- sequential statements
    -- describing combinatorial functionality

  end process;
```

- Note that we can use **sequential statements** to implement **combinatorial functionality**!



Notes:

Actually we can categorise circuits into two classes (combinatorial and sequential), and the sensitivity list is formed differently in these two cases. We will go on to consider some examples in the next few slides.

In short:

- Combinatorial circuits require that the process sensitivity list contains all input signals.
- Sequential circuits require that the process sensitivity list contains the clock signal, together with any asynchronous inputs (for example an asynchronous reset signal).

Ver 21.221

VHDL Processes



Notes:

As stated on the main slide, sequential statements can be used to describe combinatorial circuits as well as sequential ones.

When talking about VHDL statements, *sequential* simply means that the statements are executed in the order they are written in the code. This does not preclude the description of combinatorial circuits, and in fact in some cases it may be easier to describe a combinatorial circuit using sequential rather than concurrent code. For example, if the description of the circuit is more simply expressed using intermediate results, then it would be appropriate to use a VHDL process, and make use of variables within the description.

Ver 21.221

VHDL Processes



Sensitivity List for Synchronous Circuits 12.8

- The sensitivity list for processes describing synchronous circuits **always includes the clock signal**.
 - This ensures that the process is executed **every time** there is **an event on the clock**.
- Inside the process, there is usually an **if** statement which detects the **rising** or **falling** edge of the clock.
- This means that the statements inside the **if** statement **only execute on the intended clock edge** (rising or falling edge).
- All actions are synchronised to the clock.

```
synch_process : process (clk)
begin
    if rising_edge(clk) then
        -- synchronous actions
        -- ...
        -- ...
    end if;
end process;
```



Notes:

Taking for instance, the example of a D-type flip flop, the behaviour of the circuit is that, when the positive clock edge occurs, the output Q is assigned with the current input, D.

This can be coded simply as shown below.

```
d_type : process (clk)
begin
    if rising_edge(clk) then
        -- synchronous actions
        Q <= D;
    end if;
end process;
```

An event, i.e. a change in value, on the clk signal causes the process to execute.

The rising edge of the clock is detected... the statements inside the "if" statement are **only** executed on the positive clock edge (otherwise the if statement is passed over). Hence nothing happens on the negative clock edge!

Ver 21.221



Synchronous Circuits 12.9

- There may be further **control signals** which condition the operation of the circuit.
- A common example is the **clock enable** (clk_enable) signal.
- The circuit only operates if **clk_enable = '1'**, otherwise it does nothing.
- To include the clk_enable, another **if** statement must be nested inside the **clock detection if** statement...
- A **synchronous reset** can be included within the clock detection **if** statement too.

```
synch_process_en : process (clk)
begin
    if rising_edge(clk) then
        if clk_enable = '1' then
            -- synch actions
            -- ...
            -- ...
        end if;
    end if;
end process;
```



Notes:

A synchronous reset means that the circuit can only be reset on the active clock edge. Thus, if reset = '1' when the clock edge is detected, then the circuit is reset; otherwise, the circuit operates as normal.

If the circuit has both a clock enable and a synchronous reset, then the two conditions can be tested using an if-then-else statement (although notice there is no *else* part here!). Again, this resides *inside* the clock detection **if** statement, because the circuit is synchronous (all actions are synchronised to the clock).

```
synch_process_en_rst : process (clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            -- reset actions
        elsif clk_enable = '1' then
            -- synch. operations
        end if;
    end if;
end process;
```

If reset is activated, the reset actions are performed; **OTHERWISE...** If clk_enable is activated, then normal synchronous operations are performed; **OTHERWISE...** the circuit does nothing.

Ver 21.221



Sensitivity List for Asynchronous Inputs 12.10

- Including the clock in the sensitivity list caters for all **synchronous** behaviour (i.e. all actions which take place on the clock edge).
- What about **asynchronous** behaviours?
- Any signals which operate **asynchronously** to the clock should also be included in the **sensitivity list**.
- The usual example is an **asynchronous reset**, i.e. the circuit is reset as soon as the reset signal is activated, irrespective of whether this coincides with a clock edge.

```
asynch_process: process (clk, reset)
begin
    -- asynchronous reset and
    -- synchronous actions
end process;
```



Sequential Circuit Descriptions 12.11

- Including the necessary signals in the **sensitivity list** is **not sufficient on its own** to ensure that the intended behaviour is realised.
- There could be a number of control signals which control the execution of the process — **operation must be conditioned** on these signals.
- For example, a clocked process often has a reset and a clock enable signal. How is this functionality implemented?
 - Is the reset **synchronous** or **asynchronous**?
 - Reset is usually the highest priority, i.e. if **reset** = '1', the reset actions will be performed regardless of the value of the **clk_enable** signal.
- There may also be other control signals... for example a counter might have a "load" port (to preset to a particular value), and an input to determine whether the count should increment (count up) or decrement (count down). Their **order of priority** is important.



Notes:

As the reset is asynchronous here, the reset behaviour is implemented *outside* the clock detection if statement.

If the reset signal is not active, then the clock edge is detected and synchronous actions take place.

```
asynch_process : process (clk, reset)
begin
    if reset = '1' then
        -- asynch reset actions
    elsif rising_edge(clk)
        -- synchronous actions
    end if;
end process;
```

An event, i.e. a change in value, on the clk signal OR the reset signal causes the process to execute.

The reset signal is tested first, outside the clock detection if statement - hence this circuit has an asynchronous reset. All other functionality is described within the clock detection if statement, so it is synchronous.

Ver 21.221



VHDL Processes

Notes:

Remember that when if-then-else statements are used, priority is implied. This is therefore a useful construct when specifying the behaviour of a process in response to various control input signals, which naturally have some order of priority.

Also, it is useful to note that the "else" part is often not needed. For instance, refer back to the previous code extract, dealing with the asynchronous reset example. The lack of an "else" clause means that the circuit effectively does nothing when none of the other conditions is met - which is the intended behaviour. When there is no rising clock edge detected, the circuit should not perform any operations.

Ver 21.221



Processes Not Intended for Synthesis

12.12

- We can make use of processes with sensitivity lists inside **testbenches**, as well as to design circuits for synthesis
- Additionally, testbenches permit the use of processes **without sensitivity lists**. Such processes are **not synthesisable** (which is OK because testbenches are not synthesised!).
 - Processes without sensitivity lists simply begin execution at the start of the simulation, and proceed as defined by the statements inside the process.
- We can utilise **non-synthesisable statements** within such testbench processes, for example...
 - **wait** statements
 - **while** loops
 - infinite loops



Synthesisable Process Templates

12.13

- Circuits are **inherently concurrent**, and are not naturally described by sequential statements like processes.
- However, processes can be synthesised into hardware successfully, **provided that certain coding styles and guidelines are observed**.
- A set of **“synthesisable process templates”** can be used as the basis of process descriptions for synthesis.
 - Provided these rules are followed, a circuit can be **synthesised** from a process-based description.
 - Note that successful **simulation** of a design **does not necessarily mean** that the circuit can be **synthesised** (mapped to hardware)!



VHDL Processes

Notes:

We make extensive use of testbenches in VHDL, and there is always at least one process in a testbench (in order to create the test stimulus).

Processes without sensitivity lists are commonly used, and this type of process must be controlled explicitly using for example wait statements and while loops. Remember that the process will repeat indefinitely unless the process includes a final wait statement!

It may also be appropriate to include a process *with* a sensitivity list in a testbench, as we will see later in the course.



Ver 21.221

VHDL Processes

Notes:

We will not discuss the process templates themselves in this class — for now, it is sufficient just to be aware that they exist!

Bear in mind that when targeting an FPGA, the described circuit must map to circuit elements that are available on the chip. For instance, flip-flops are built into the fabric of the FPGA, so the description of a flip-flop in the VHDL code needs to map to the available flip-flop resources.



Ver 21.221

Conclusions

12.14

- In this presentation, we have looked specifically at VHDL **processes**, which are the most used construct for writing **sequential** code in VHDL.
- Processes can be written both to describe hardware (i.e. intended for **synthesis** to a circuit) and for testbenches (**non-synthesisable**).
- We discussed the behaviour of processes in detail, in particular the scope of local declarations, and the role of the sensitivity list.
- The choice of signals to include in the process **sensitivity list** is crucial to its operation: a process runs only when one of the signals changes.
- Processes **without sensitivity lists** are not synthesisable, but they can be very useful in **testbenches**.
- For processes intended for synthesis, a set of **templates / rules** must be followed to ensure that the description can be successfully converted into a hardware circuit.

Notes:

VHDL Processes

Ver 21.221

VHDL 13

Introduction to Digital Devices and Design Flows

Semiconductor Chips

13.1

- You may be familiar with the idea of semiconductor 'chips' that can be integrated into electronic circuits. There are several different kinds of these chips.
- In this presentation we will consider the following:
 - **FPGA** devices
 - Application Specific Integrated Circuits (**ASICs**)
 - Application Specific Standard Products (**ASSPs**)
 - **Processors** (there are several different types!)
- There are also other types that we won't cover here, including memory, analogue, optoelectronic and sensor chips.

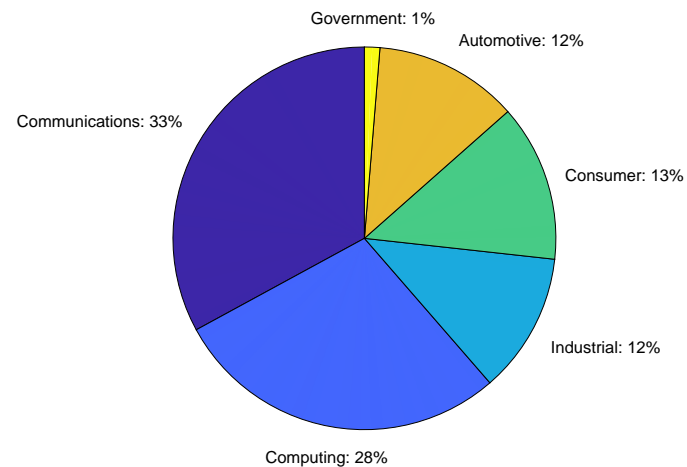
Notes:

Introduction to Digital Devices and Design Flows

Semiconductor Market by Application

13.2

- The global market for semiconductors was **\$412.3 billion** (US dollars) in 2019, split across the following application segments (by revenue).



Notes:

'Government' includes, for instance, defence applications.

The source for the data on the main slide is:

Semiconductor Industry Association, *2020 Factbook*. Available:

https://www.semiconductors.org/wp-content/uploads/2020/04/2020-SIA-Factbook-FINAL_reduced-size.pdf

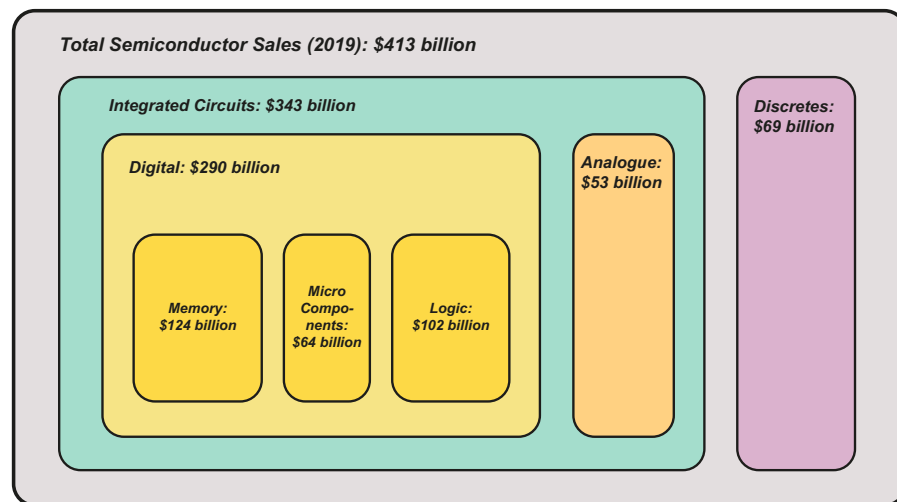
Ver 21.314



Semiconductor Product Types

13.3

- Under the "semiconductors" umbrella term, there are a number of different product types, shown below.



Notes:

Notice from the diagram on the main slide that:

- The vast majority of semiconductors (by value) are **integrated circuits**.
- Of those integrated circuits, the vast majority are **digital** ones.

There are three subcategories within the 'digital' category, and these are:

- Memory** (modules for data storage)
- Micro-components** (this category includes microprocessors and microcontrollers)
- Logic** (which includes FPGAs, along with other kinds of fixed and programmable logic circuits)

Also, notice that approximately a quarter of all semiconductors fall into the (digital) 'logic' category.

Information source (and diagram based upon):

Semiconductor Industry Association (SIA), *Trade Poster (2019 update)*. Available:

<https://www.semiconductors.org/wp-content/uploads/2018/10/SIA-Trade-Poster-2019-Update-WEB.pdf>

Ver 21.314



Integrated Circuits vs 'Discretes'

13.4

- As their name suggests, 'integrated circuits' combine a number of different components on a single device, to create a complete circuit.
- Integrated circuits can be complex, implementing very **sophisticated** functionality using **billions of transistors**. More on these later!
- The 'Discretes' category shown on the previous slide is often known as O-S-D, and it includes:
 - **O**ptoelectronic components — e.g. light sensors, laser diodes, waveguides.
 - **S**ensors and actuators — pressure, temperature sensors; valves, switches.
 - **D**iscrete components — individual components with simple functionality, e.g. transistors, diodes, rectifiers.



Digital Integrated Circuits

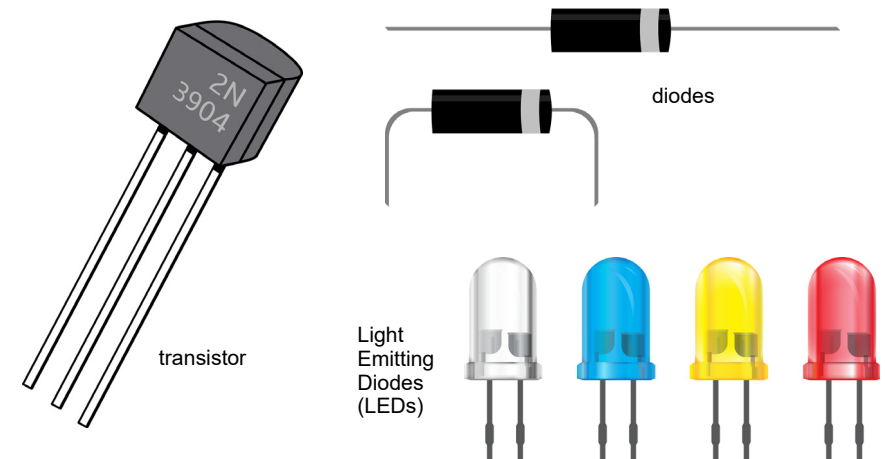
13.5

- As shown a couple of slides ago, the category of digital integrated circuits (or ICs) includes three sub-categories:
 - **Memories** (data storage)
 - **Microcomponents** (microprocessors etc.)
 - **Logic** (digital hardware circuits, fixed or programmable)
- We will not focus on dedicated memory chips in this chapter, although memory may form part of a larger circuit system implemented on an IC.
- Instead, we will evaluate the device options for creating a new digital system design. To do so, we will consider:
 - A **processor** running software;
 - A **logic circuit** implementing functionality in hardware.



Notes:

Discrete components can perform important functions, such as acting as a power regulator on a Printed Circuit Board (PCB). A power regulator ensures that voltage and current stay within specified ranges. Some example images of **discrete** semiconductor components are shown below.



Vector images: free, public domain (Pixabay).

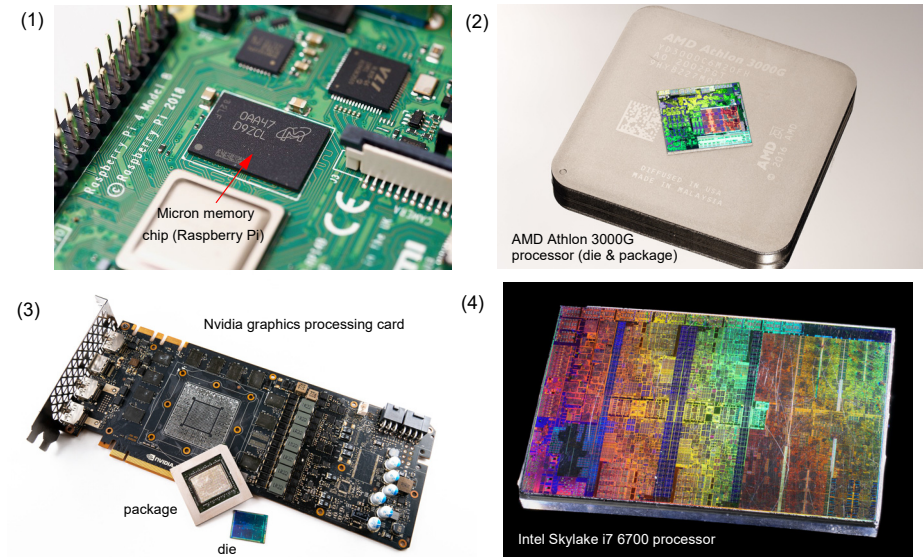
Ver 21.314



Introduction to Digital Devices and Design Flows

Notes:

Here are some examples of semiconductor chips.



Photos, Flickr: (1) Jeef Geerling ([Creative Commons CC BY 2.0](https://creativecommons.org/licenses/by/2.0/)); (2),(3), (4) Fritzchens Fritz, public domain.

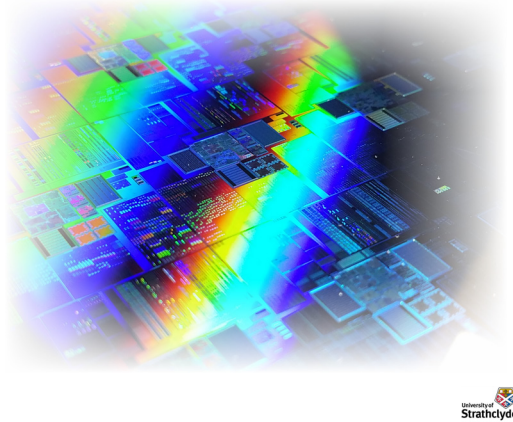
Ver 21.314



Application Specific Integrated Circuits (ASICs)

13.6

- An ASIC is an integrated circuit that has been designed to implement a specific set of functionality. It can therefore be optimised for the target purpose.
- ASICs are manufactured from silicon wafers.
- The functionality of an ASIC is fixed at manufacture, so...
- ... any mistakes are costly!
- Rectifying an error requires a new iteration of the design, then re-manufacture.



The Creation of an ASIC

13.7

- ASIC design and manufacture is a long and complex process, entailing:
 - Functional design
 - Layout of the circuit
 - Creation of masks (for the lithography process)
 - Manufacture of silicon wafers
 - Creation of multiple chips on each wafer (copper / lithography)
 - Cleaning
 - Separation of a wafer into individual chips, and packaging.
- To learn more about ASIC manufacture, please watch the recommended videos (links are provided on the class MyPlace page).

Notes:

Introduction to Digital Devices and Design Flows

As ASICs are designed for a particular purpose and set of functionality, they can be as large or small as required. In general, however, ASICs are physically smaller than FPGAs. They are placed inside packages to protect them and enable interfacing to external signals.

Another advantage of ASICs is that they consume very little power. This can be especially advantageous in applications that are battery-powered.

However, the functionality of an ASIC cannot be changed after manufacture, either to correct a mistake or to integrate an upgrade.

There are different approaches to designing an ASIC:

- **Gate Array** — The ASIC design is developed on top of a pre-defined layer of silicon that implements an array of gates. This design approach provides the lowest level of customisation, but may provide a speedier design process with a reduced likelihood of errors.
- **Standard Cell** — The ASIC design is developed using libraries of pre-designed and verified blocks. These might not be 100% optimised for the specific design requirements of a given ASIC, but this approach has the advantages of being quicker and carrying less risk.
- **Fully Custom** — The ASIC is designed at a more fundamental level, without making use of library components. This can lead to a more optimised solution (e.g. reduced area and power), but requires more extensive design effort and expertise. There is also a higher risk of errors and additional design cycles*.

* Design cycle: the process of updating a design, and then going through all subsequent steps of manufacturing and verification.

Main slide image: Wikimedia Commons ([CC BY-SA 4.0 licence](https://commons.wikimedia.org/wiki/File:ASIC_chip.jpg), user: 2x910). The image has been modified by applying a fade effect at the edges.

Ver 21.314

Notes:

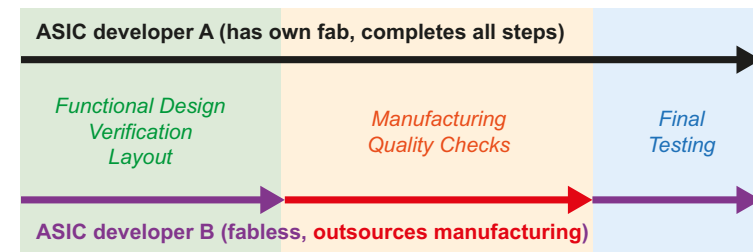
Introduction to Digital Devices and Design Flows

A small number of semiconductor companies perform all stages of ASIC design and manufacture themselves. Examples include Intel and Samsung (i.e. really big companies!).

Most ASIC designs, however, are developed by two companies operating in partnership.

- The company developing the ASIC produces the design for the circuit.
- They then contract a fabrication company to manufacture the ASIC.

This is often referred to as “the fabless model” (from the perspective of the company developing the ASIC). In other words, that company does not have a fabrication facility (“fab”) of its own.



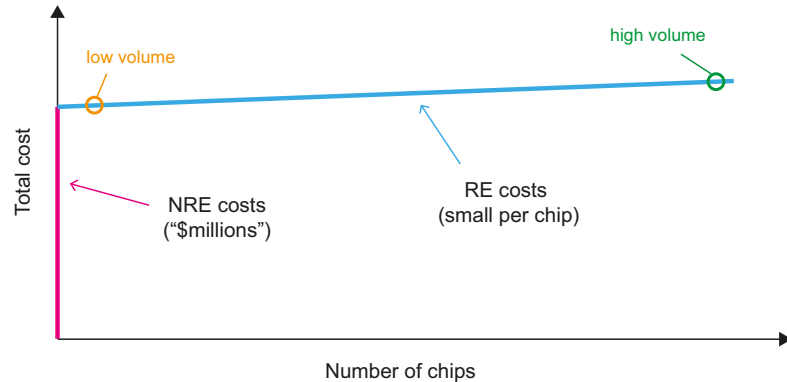
Why does this occur? ASIC manufacture is extremely expensive, requiring factories with highly controlled environments, along with specialist equipment and staff. Unless the company developing the ASIC will produce a huge volume of chips, then it does not make financial sense to have its own fab. Even in this fabless model, the costs of ASIC manufacture are considerable. We will discuss the economics of ASIC manufacture next.

Ver 21.314

The Economics of ASICs

13.8

- When developing an ASIC, there is a large upfront cost involved in generating the layout, creating the masks, and preparing for manufacturing (Non-Recurring Engineering costs, or NRE costs).
- Thereafter, the additional unit cost of producing each chip, i.e. Recurring Engineering (RE) cost, is relatively low.



Notes:

Introduction to Digital Devices and Design Flows

Let's look at a couple of examples to understand the impact of NRE costs when considering an ASIC design. We'll review these costs in terms of US dollars (common in the industry).

Assuming that the NRE costs of developing our ASIC are \$2 million, let's define $C_{NRE} = 2,000,000$.

Thereafter, the cost of developing each chip (RE cost) is 15 cents, i.e. $C_{RE} = 0.15$.

For a low volume of production (depicted at the left hand side of the main slide), let's assume that 1,000 chips are produced ($N_{chips} = 1,000$). The total cost is:

$$C_{TOTAL} = C_{NRE} + (N_{chips} \times C_{RE}) = 2,000,000 + (1,000 \times 0.15) = \$2,000,150$$

Therefore, the overall unit cost is 1000th of the total cost, i.e.

$$C_{unit} = C_{TOTAL} / N_{chips} = 2,000,150 / 1,000 = \text{\textbf{\$2,000.15 per chip}}$$

Clearly, this is very expensive! Let's try again, assuming that we want to make 1 million chips (high volume, depicted on the right hand side of the main slide plot).

The total cost is now:

$$C_{TOTAL} = C_{NRE} + (N_{chips} \times C_{RE}) = 2,000,000 + (1,000,000 \times 0.15) = \$2,150,000$$

And when we calculate the cost per ASIC chip, we get:

$$C_{unit} = C_{TOTAL} / N_{chips} = 2,150,000 / 1,000,000 = \text{\textbf{\$2.15 per chip}}$$

Which is much more reasonable!

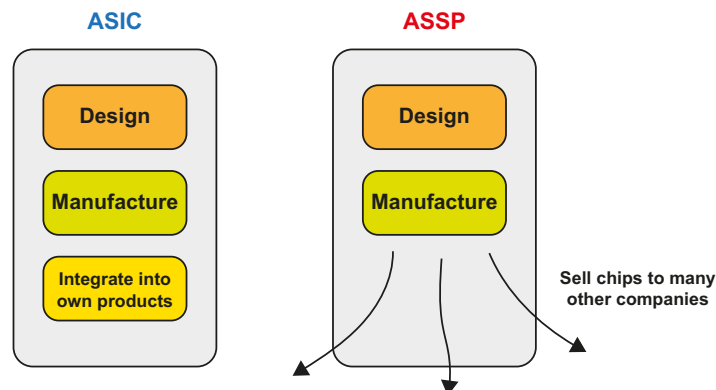
The takeaway point is that ASIC designs are only viable when the volume of production is HIGH.

Ver 21.314

Application Specific Standard Products (ASSPs)

13.9

- This sounds a bit like an ASIC ("application specific") and indeed it is.
- An ASSP is a chip that is tailored to a particular application, but which is ***sold as a product to many other companies***.



Notes:

Introduction to Digital Devices and Design Flows

To give an example, if your company was developing smart lighting solutions, then you might buy a standard WiFi chip "off the shelf" to integrate into your products. The WiFi chip is an ASSP.

This would be a much cheaper, quicker and less risky approach than designing your own chip (an ASIC). However, you would need to integrate the WiFi functionality from this ASSP with potentially several other ASSPs, depending on the overall system design. This would involve a more complex Printed Circuit Board (PCB), and a physically larger end product.

The company that developed the WiFi chip would have gone through the whole ASIC design and manufacture process... but they might have manufactured 10's or 100's of millions of these chips... so from their perspective, the economics would be good!

ASSPs can include a number of different application areas, e.g.

- Communications functions, e.g. WiFi, Bluetooth, Ethernet, USB, GPS...
- Sensor / actuator processing modules
- Image sensors / camera modules
- Audio codecs
- Video codecs
- ... and many others.

However, if you are developing a new product, you might not be able to find an ASSP that caters for your specific application requirements. Also, the purchase price for ASSPs will incorporate the profit made by the manufacturer of that ASSP (but it may still be an attractive option, given the design effort saved, and risk avoided).

Ver 21.314

Field Programmable Gate Arrays (FPGAs) 13.10

- To break down this name:
 - Field programmable** — function is defined by the customer; the device can be reprogrammed in situ (“in the **field**”).
 - Gate array** — the device is comprised of logic resources laid out in a regular pattern.
- Advantages** of FPGAs include:
 - Custom functionality can be designed without the expense and risk of developing an ASIC.
 - FPGAs can be reprogrammed as often as you like.
 - They come in multiple different sizes with different feature sets.
 - The design process is less complex than for ASICs (no layout), and designs can be easily ported between FPGA devices.



Notes:

Introduction to Digital Devices and Design Flows

There are also other devices that fall into the same category of “Programmable Logic” as FPGAs.

In particular, **Complex Programmable Logic Devices (CPLDs)** are similar to FPGAs in the sense that they are parallel and reconfigurable, but they are smaller and far less sophisticated. CPLDs have very low power consumption, and are suited to “glue logic” type applications.

One disadvantage of FPGAs as compared to ASICs is power consumption. FPGAs provide (re)programmable logic, and power is required to program the desired circuit onto the FPGA, and then hold its state. FPGAs are often quoted as drawing about 10x the power of an equivalent ASIC, although this figure is hotly debated!

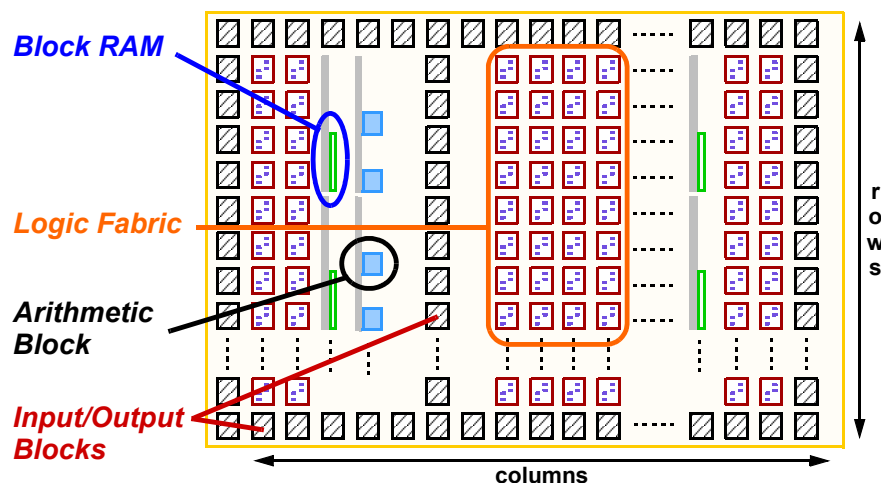
The major advantage of FPGAs is their flexibility. These devices can be configured into any circuit, and reconfigured thereafter. This takes away the risk of spending millions developing an ASIC and finding an error — if an error is found in an FPGA design, it can be very quickly corrected, with no re-manufacturing costs. It also makes it very easy to upgrade functionality in the future, e.g. to add new features.



Ver 21.314

FPGA Architecture 13.11

- FPGAs provide a highly parallel and flexible implementation platform for designing digital circuits. This diagram is representative of Xilinx devices.



13.11



Notes:

Introduction to Digital Devices and Design Flows

Notice that the structure of an FPGA is very parallel. There are many processing resources available to be reconfigured into a circuit, and these can all operate at the same time as each other.

Depending on the device, FPGAs can operate at clock frequencies of several 100's of MHz. This might not be as fast as some processors (you might hear of processors operating at up to around 4GHz) but FPGAs can do much more simultaneous computation.

For instance, if you had four people making sandwiches quickly (let's say 50 sandwiches per hour), then after an hour you'd have 200 sandwiches. But on the other hand if you had 100 people making sandwiches a bit more slowly (say 10 sandwiches an hour), then after an hour you'd have 1,000 sandwiches! So in total, more work would have been achieved in the same amount of time. It's similar when comparing FPGAs with processors.

That said, processors do certain types of computation better than FPGAs, and vice versa, so it's not quite as simple as a straightforward calculation. The key point is that clock frequency is not a direct measure of how much work an FPGA can do, so it's unfair to compare processors and FPGAs based on clock frequency alone.

When inspecting your own FPGA designs in Vivado, you may notice that certain types of resources are used on the FPGA. In particular, simple designs will probably be implemented using the Logic Fabric (shown on the main slide). The low level components that make up the logic fabric are:

- Lookup Tables (LUTs)** — LUTs can implement simple logic functions, and they can also act as small memories (Read-Only or Random Access Memories, i.e. ROMs and RAMs, and shift registers).
- Flip-Flops (FFs)** — The basic functionality of a FF is as a D-type register / flip-flop. Each FF can store 1 bit.

There are also **Input / Output Blocks (IOBs)** which carry signals into and out of the FPGA. IOBs are situated at the edges of the FPGA, and/or in columns in the centre of the device.

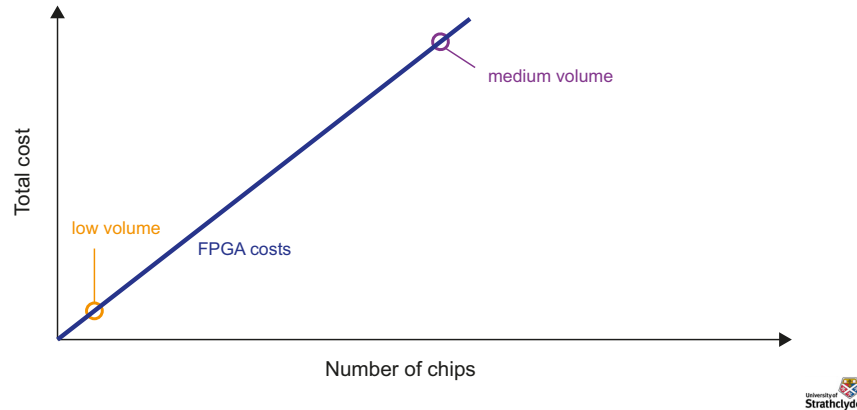


Ver 21.314

The Economics of FPGAs

13.12

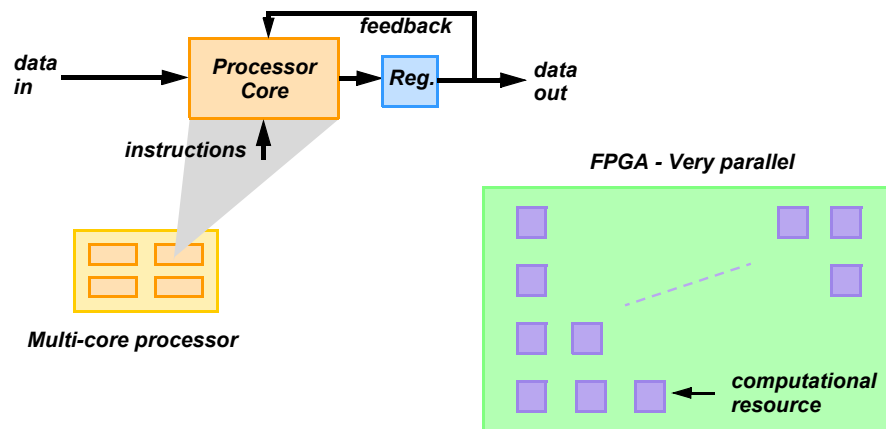
- The NRE costs of producing an FPGA-based system are negligible. We simply need to deploy a bitstream onto an FPGA to program it.
- FPGAs can be purchased in any quantity, so the cost scales linearly at worst (in reality, discounts may apply for larger volumes). However, the cost per unit will be higher than the RE cost of producing an ASIC.



Processor Architecture

13.13

- Processors** have one (or a few) cores which execute instructions in a serial fashion, at very high speed. Processors have fewer, more sophisticated, processing units than FPGAs.



Notes:

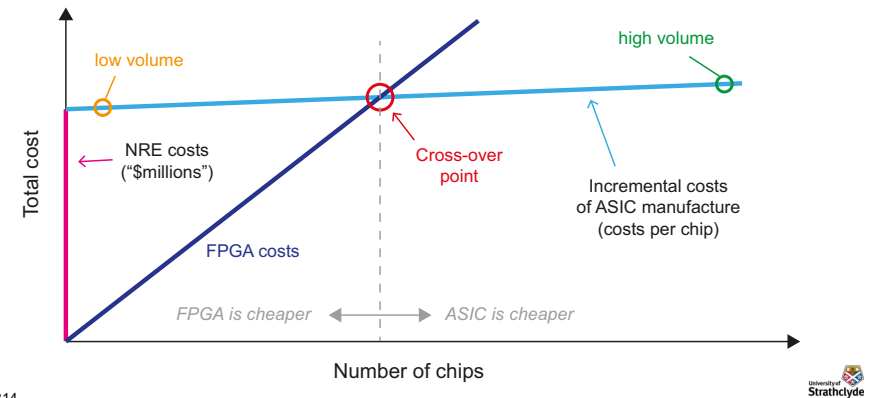
Introduction to Digital Devices and Design Flows

Thinking again about costs, let's compare the options to: (i) generate a design based on an FPGA, and (ii) develop a full ASIC design. The FPGA option would involve buying FPGAs from an FPGA vendor. Let's say that the required FPGAs cost \$5 each.

There is a cross-over point when the two options cost the same:

$$C_{unit} = [C_{NRE} + (N_{chips} \times C_{chip})] / N_{chips} = \$5$$

Given the values defined previously, i.e. $C_{NRE} = \$2,000,000$ and $C_{RE} = \$0.15$, then the cross-over point can be calculated as 412,371 units. If a lower quantity is required, then an FPGA would be more cost effective; if a higher quantity is needed, then an ASIC becomes the cheaper solution.



Ver 21.314

Notes:

Introduction to Digital Devices and Design Flows

The diagram on the main slide depicts the operation of a processor at a very high level of abstraction. There are many more details that you could look into. Processors are designed and manufactured in the same way that ASICs are.

Processors are available with many different specifications, and price points — everything from a server-grade Xeon processor to a low cost processor for embedded applications. That said, processors generally fall into two categories:

- General Purpose Processor (GPP)** — GPPs are designed to serve a variety of purposes well. This may include processors used in desktop or laptop computers, servers, etc.; or lower power GPPs suitable for used in embedded systems.
- Application-Specific Processor (ASP) / Application-Specific Instruction Set Processor (ASIP)** — this category of processor is optimised for a specific application area. Examples include: Graphics Processors / Graphics Processing Units (GPUs); Digital Signal Processors (DSPs); Floating Point Units (FPUs), and so on. Dedicated processors for Artificial Intelligence (AI) are now becoming popular too.

The operations performed by a processor are defined by **software**, using a software programming language. Processors are especially useful when an operating system is needed, and for processing that involves sequential tasks and decision making.

Processors fundamentally differ from ASICs and FPGAs in this respect:

- FPGAs and ASICs both implement hardware circuits.
- Processors are used to run software.

Ver 21.314

Device Comparison

13.14

- We can compare the attributes of the device categories considered.

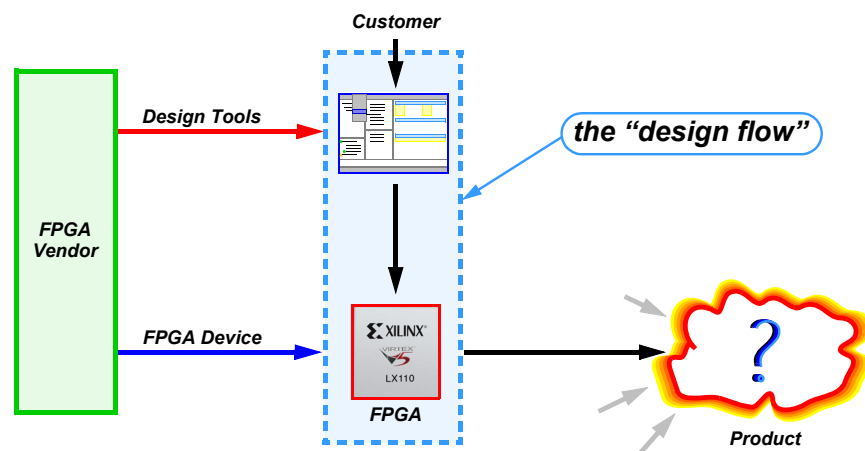
| Attribute | ASSP | ASIC | FPGA | Processor |
|--------------------------|---------|---------------------------|------------------|-------------|
| Customise functionality? | No | Yes | Yes | Yes |
| Re-programmable? | No | No | Yes | Yes |
| Architecture | Various | Various | Parallel | A few cores |
| Physical size | Small * | Small | Larger | Small |
| Clocking rate | Various | Very high | Medium / high | Very high |
| Power consumption | Low | Low | Medium | Various |
| Design method | None | Functional (HDL) + Layout | Functional (HDL) | Software |
| Development time | None | Long | Short | Short |
| Best for (quantities) | Low | Very high | Low / Medium | Varies |
| Time to market | Fast | Slow | Fast | Fast |

- * The physical size of an individual ASSP may be small, but your design may require several ASSPs, resulting in the largest footprint overall.

Design Flow for FPGAs

13.15

- FPGAs are programmable, and so FPGA companies have to provide not just the devices, but software tools for programming them with!



Notes:

Introduction to Digital Devices and Design Flows

As you can see from the table on the main slide, there is no single option that wins in all cases. Deciding on the best approach involves weighing up many different factors.

One important factor not discussed so far is time-to-market, and this can be crucial. Often, the company that brings the first product to market for a particular application wins (and then retains) much of the market share. The ability to bring a product to market quickly can therefore be a very important factor.

The design process for ASICs and FPGAs begins in a very similar way. The desired functionality is described using a Hardware Description Language (VHDL or Verilog), and simulations are performed to verify correct operation of the designed circuit. Thereafter:

- For FPGA designs, specialist FPGA tools are used to automatically synthesise this description into a bitstream for programming the FPGA, with little human intervention required.
- For ASIC designs, a completely different set of tools is required to generate a layout based on standard cells, or a fully custom design. Further simulations are then required to check the post-layout design against the functional design developed in HDL.

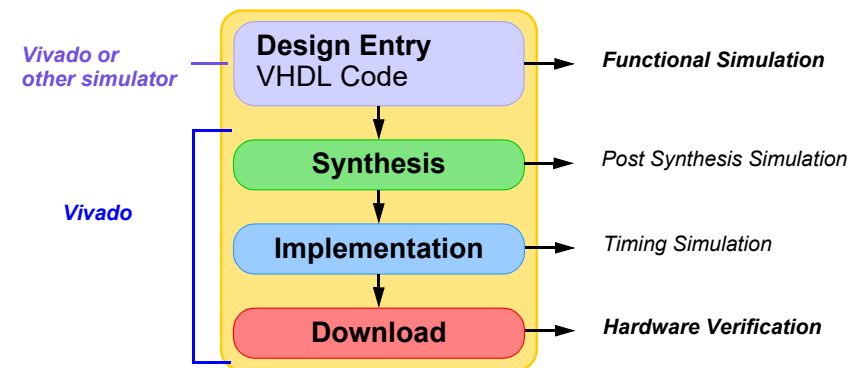
Ver 21.314

Notes:

Introduction to Digital Devices and Design Flows

As part of this course we will design and simulate using VHDL, using the design entry and simulation tools within Vivado. Vivado will also be used to synthesise and implement the design, for a target FPGA device.

The FPGA design flow involves testing and verifying the system along the way!



Ver 21.314

Design Entry Alternatives for FPGAs

13.16

- In this course we cover a **Hardware Description Language (VHDL)**.
- There are other methods of design entry:
 - **Schematic capture** — placing and connecting symbols of low-level components. This method is seldom used now.
 - **Block based design** — most FPGA companies offer a tool for designing circuits by placing blocks and connecting them together. These offer higher levels of abstraction than schematic capture (bigger, more sophisticated blocks!).
 - **High Level Synthesis** — Generation of HDL Code from code written in a software language (e.g. C, C++).
 - Research is also ongoing into **new and efficient** methods.
- Note that in most of the above, the first step is to convert the design into an HDL representation! These techniques can also be combined.



Conclusions

13.17

- Several types of digital device exist. **ASICs** and **FPGAs** are very prominent and have different characteristics and therefore are suitable for different systems. **ASSPs** can be bought off-the-shelf too.
- **ASICs** are entirely customisable at manufacture and low power, but economic only in large quantities (*think mobile phones*).
- **FPGAs** have programmable components and can be reconfigured as desired, but they consume more power (*think mobile base stations*).
- **Processors** are another option. Functionality is implemented solely by writing software for sequential execution (on one or a few cores).
- **Hardware description languages** like **VHDL** are needed to define the functions implemented on an FPGA or ASIC.
- In implementing a design, several steps are required and collectively this is known as a **design flow**. It involves several software tools. **Testing and verification** are important and integral tasks.



Notes:

The methods described on the main slide are often used to produce functional blocks which are then integrated into a complete system. Design tools are available to help with this. In Vivado, there is a tool called IP Integrator which can be used for system-level design.

It is possible to design complete systems using an HDL like VHDL or Verilog, but time consuming. Therefore, design methods that “raise the abstraction level” and allow designs to be created more quickly, are gaining in popularity. However, even if using one of these methods, it is valuable to understand principles of HDL description — HDL is usually generated at some stage!

Ver 21.314

Notes:

Introduction to Digital Devices and Design Flows



Introduction to Digital Devices and Design Flows

Ver 21.314

