

**Advanced VLSI Design**  
**Prof. D. K. Sharma**  
**Department of Electrical Engineering**  
**Indian Institute of Technology – Bombay**

**Lecture - 20**  
**Basic Components in VHDL**

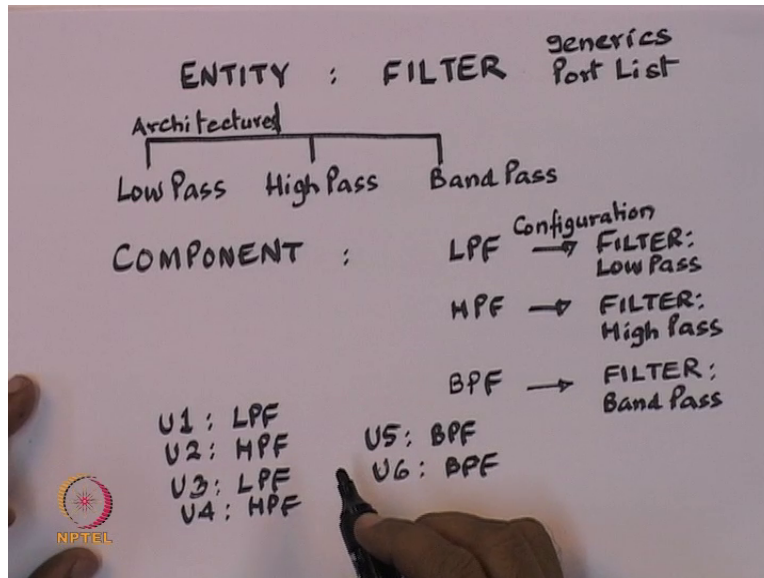
In our last lecture on Hardware Description Language and VHDL in particular, we had started discussion on the design elements which constitute the language. we had looked at entities, we had looked at architecture, we had seen that an entity architecture pair constitutes a template for a piece of hardware, this template is then fixed as a component type by the term component and then it is actually instantiated as one specific piece of hardware by a component instance.

Binding a particular component to an entity architecture pair is done by configuration in fact configuration can be in line during the description of a component itself or it could be a standalone unit which binds a particular component type to an entity architecture pair, we had also seen that we can place various definitions, various components and so on in packages and several packages can be put together in a library.

We had also seen the syntax and the usage of configuration, packages and libraries. There is one point which we will probably take up in some detail today and that has to do with this somewhat confusing terminology of entity architecture pairs, component types and actual instances. As we had talked about earlier we had said that modern versions of VHDL permit you to instantiate entity architectures directly, this has weakened somehow the usage of the component type.

And therefore sometimes this hierarchy of design description is not made very clear, let us just make it clear with one example.

**(Refer Slide Time: 02:53)**



Let us say that we have an entity called filter, this is probably a digital unit which carries out filtering of the input data using whatever algorithm, now notice that as far as the entity is concerned it only describes the interconnection of this object with the outside world it does not care what the insides of this could be, that means it will provide a connection to the input data stream to the output data stream set of coefficients and so on.

So those will be the items which will be described by the entity in its port list and the generics might for example specify the critical frequency and so on of this filter. This entity can now have several architectures and let us say that we have three architectures for this entity, these are the architectures for this entity notice that an architecture is defined only with respect to an entity, now our entity is a filter.

And now we can have separate inside mechanisms in short of this filter and that may describe the behaviour of this filter and you could have an architecture, for example which could be Low pass, you could have an architecture which is High pass, and you could have an architecture which is Band pass.

Notice that all these architectures are compatible with the interconnection of this piece of hardware with the outside world what this architectures determine is how is the input data stream handled inside this piece of hardware in order to produce an output stream which is a low pass

version or high pass person or a band pass version of the input data stream, around which frequency is will low pass, high pass or band pass will occur is in fact decided by the generics of this entity, so this entity has only the generics and the port list.

Next you might define components, so for example you bind a particular pair of entity architecture to a component type, so I might have a component called LPF which is then bound to the entity filter with architecture low pass, I could have a different component called HPF which is bound to the same entity but with a different architecture and finally I could have yet another component called BPF.

These are names of components and this is connected to the same entity but with a third architecture which is band pass. So using an entity and its three architectures I have actually described three different component types these are not yet specific components, now I can instantiate these components and maybe in my hardware there are six pieces is a hardware, six actual instances of components.

And it is possible that U1 is of type LPF, U2 is of the type HPF, U3 is again LPF, U4 is again HPF and U5 is BPF, U6 is BPF. now notice the hierarchy of the use of these design elements, the entity is the top unit it specifies that I am describing a class of hardware called filters, the entity describes only the generics that means a manifest constants which describe the properties of this filter and port list which describes how this class of hardware will be connected to other pieces of hardware and entity can have several architectures.

In this case actually it is somewhat unusual, the different architectures actually provide different functionality a much more common use is when different architectures provide different implementations of the same functionality, but to take the most general case we may have different architectures after all our entity is the generic filter and the architectures decides whether it is a low pass filter, a high pass filter or a band pass filter.

So we have three architectures and at this stage this entity with three architectures is not specific hardware it is not an integrated circuit on a board so to speak, it is a template for the kind of

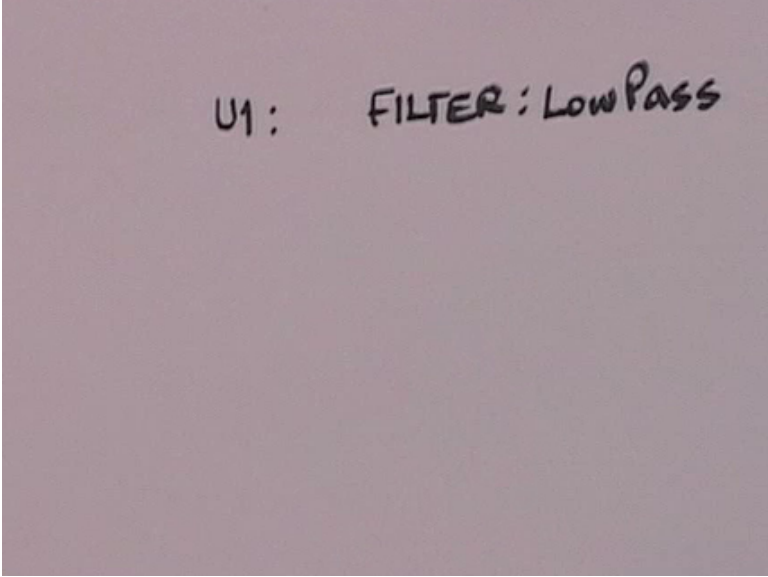
hardware that we can use, we now describe component types and LPF, HPF and BPF are component types, these are still not specific pieces of component, these are component types which are available to us.

And then each component type is then bound in fact this binding is done by configurations, so you have a configuration which binds LPF to entity filter with the architecture low pass, HPF to entity filter with architecture high pass and the entity BPF to the entity filter with band pass, it is only after I have declared these components types using a configuration and entity architecture pairs, that I can actually use these component types of U1, U2, U3, U4, U5, U6 are the actual pieces of hardware which describe my design.

So U1 is then bound to a component type, U2 is also bound to a component type and so on. So each instance of component needs a component type to which it will bind and each component type must then be configured to an entity architecture pair, I hope that explains this somewhat confusing hierarchies which is in fact a bit too detailed if you ask me, but this is the hierarchy that we have in the full-blown hierarchy that we have in VHDL.

However most of the time we do not need this full-blown hierarchy most of the time generally just one level or two are good enough in that case in modern versions of VHDL you are permitted to skip the component stage all together, that means a particular instance can be bound not to a component type but directly to an entity architecture pair that means it will be possible in the modern version of VHDL.

**(Refer Slide Time: 12:29)**



U1: FILTER: LowPass

To say that component instance U1 is bound in fact to entity filter with architecture low pass this direct binding is a somewhat late edition to VHDL earlier the binding could only be done to an object type component, so only a design unit component could be instantiated in the earlier versions of VHDL modern versions of VHDL allow instantiation of entity architecture pairs directly.

Once we have this done, now it is relatively easier to see how hardware could be described structurally, you have interconnections defined by signals, you have hardware components which are instances of component types and component types are bound using configurations to entity architecture pair, so this is the super structure which is assumed in a VHDL based hardware description.

Modern versions of VHDL allow short circuiting the component stage if that is considered convenient it still respects the old hierarchy and therefore if you describe components that is also okay, with the modern versions of architecture but it also allows you take the shortcut of directly instantiating entity architecture pairs, so this is the hierarchy of design elements in modern version of VHDL.

**(Refer Slide Time: 14:21)**

## Design Elements in VHDL: Libraries

Many design elements such as packages, definitions and entire entity architecture pairs can be placed in a library.

The description invokes the library by first declaring it:  
For example, `Library IEEE;`

Objects in the Library can then be incorporated in the design by a 'use' clause.

For example, `Use IEEE.std_logic_1164.all`

In this example, IEEE is a library and std\_logic\_1164 is a package in the library.

In addition to that of course you have libraries and packages and we had seen this in the last lecture that you have this syntax of library dot package dot component and you can make any component from any package from any library visible to your design by this kind of declaration which first declares the library and then you use what is called use clause which allows you to select which component of library package etc. will become visible to your design.

**(Refer Slide Time: 15:00)**

## Object and Data Types in VHDL

VHDL defines several types of **objects**. These include **constants, variables, signals** and **files**.

The types of values which can be assigned to these objects are called *data types*.

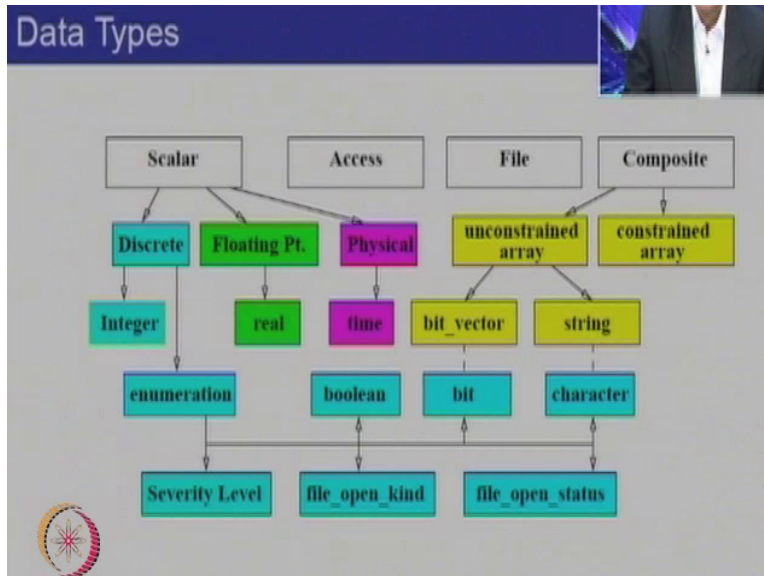
Same *data types* may be assigned to different *object types*.  
For example, a **constant**, a **variable** and a **signal** can all have values which are of *data type BIT*.

Declarations of objects include their *object type* as well as the *data type* of values that they can acquire.

For example `signal Enable: BIT;`

We had also seen object and data types, object types are constants, variables, signals, files and in fact that are pointers types as well and then the data types could be of type bit, of type std logic define later in library and so on.

**(Refer Slide Time: 15:21)**



Let us look at data types in much more detail now, notice now we are talking of data types and not of object types different object types can use the same data type, for example a constant or variable or a signal could all be of type bit. Now let us see the classification of the data types which we can handle in VHDL, you have types which are scalar, a Scalar is a single object, you have a type which is Access this is like pointers in programming languages.

You have a data type like File and then you have Composite data types which are collections of scalar types. Let us now look at the scalar types, the scalar types could be of the type discrete shown here in blue, they could be floating points that means which are inherently continuous type variables of course in actual implementation because of the size of the variable is limited these are also somewhat discrete but they are meant to be continuous.

And finally you have a scalar type called physical these are used to describe physical quantities and they have not only a value but also a unit and different sized units can be interrelated in a physical type, so essentially we had three scalar types, we have things which are inherently discrete like integers, we have things which are theoretically continuous though in practical implementation with fixed size representation they are also discrete.

But they are meant to represent continuous quantities like floating point and finally there are physical types which these two carry only values but you have physical types which carry values

as well as units and then the values can be transformed, because you can define several units for the same physical unit, physical quantity. Now let us look at all three of this one by one.

Discrete kind of scalar values can be either of type integer, integers are well known to us these are positive or negative numbers inclusive of 0, they could be of type enumeration in which you enumerate a set of value that this discrete type can take, so for example a scalar could be a discrete type but it cannot take any old value.

It can take any value out of a list that we shall enumerate, various enumerated types are inbuilt into the language as you follow the language you will become familiar with these but at least three of these are quite common and enumerated type Boolean as the value false or true, so if you have a scalar which is discrete and enumeration type then a pre-defined enumeration type is a Boolean type and the Boolean has enumerated values true and false it can take only these two values.

Similarly, we have an enumerated type called bit, the type bit can take values 0 and 1, so this is also a two valued enumerated type the enumerated type is therefore 0 or 1. Notice that the integers 0 or 1 is therefore quite distinct from the enumerated type 0 or 1 in VHDL and VHDL is a very strongly typed language it insist on this distinctions being made.

The third built-in type which is enumerated is character essentially these are ASCII values or whatever values which represent text characters and these are all enumerated, therefore the type of characters known to the language is limited by the enumeration and this is a longer list then Boolean or bit that we had seen which are two value.

So these are essentially all the alphabet characters which are enumerated and a scalar type which is of type character can take any one of these specified values, so a character type can take any one of those specified values. Notice that internally all the discrete types and indeed all the floating points are represented using bits that means 0's and 1's.



But as far as the syntactical structure of the language is concerned it checks that the value is either false or true in case of Boolean either 0 or 1 in case of bit or any one of those specified text characters in types - a type of characters, what it does internally to represent these is of no concern to the person who is using VHDL.

Apart from these three you have other specialized enumeration kinds these are severity level in case of an assertion which is a statement that we shall see later, there is a file open kind which essentially is associated with the type file and file open status, file open kind could be are you reading a file for read-only or for read write etc. etc. and file open status whether this file is open whether it is successfully open in the specified mode or not etc.

So the file open status is also of an enumerated type, so these are the pre-defined enumerated type but the option remains with the user to define enumerated types of your own, these are the ones which the language provides to you, but you do have the option of defining your own type which is enumerated.

For example you might design an ALU and define an enumerated type called ALU command and the ALU command could have any one of the enumerated values add, subtract, multiply or divide. So enumeration types which are shown here are those which are already built into the language but you can expand this types by defining your own enumerated types. As far as floating points are concerned the built-in type for that is called real.

As far as physical types are concerned for scalar there is only one pre-defined physical quantity and that is time of course we have seen the time is crucial to a hardware description language and therefore time is pre-defined in VHDL, so that is the physical type which is pre-defined in VHDL, however as in other cases you can define your own physical type which will carry its values and its units that you can declare.

Having look at scalars it is now meaningful to look at composites, so composites are collections of scalars and you can have unconstrained arrays or constrained arrays, unconstrained arrays are things which can have any number of scalar quantities which are associated with them,

constrained arrays are those which are restricted to a particular range of scalar values, to unconstrained arrays are pre-defined in the language.

A bit vector is an unconstrained array of bits which we had earlier seen in the enumerated types, the bit vector can be of any size that is why it is unconstrained and therefore the size has to be defined by the user, so when you use a bit vector which is a collection of bits for example you might have a data bus it is for you to decide that this bit vector will have 16 elements whose index will run from 15 to 0.

Similarly, unconstrained array of the type string a pre-defined strings are collection of the type character which is an enumerated type in VHDL, so in short the bit type and the character type have associated unconstrained arrays declared already in the language, so therefore bit vector is in fact a an element of the language you can directly use and declare some collection of bits to be bit vectors without having to describe a new type.

Similarly, you can declare a collection of characters to be a string which is an unconstrained array, whenever you use an unconstrained array, then you have to fix the size of this unconstrained array, the type does not constrained the size, the user when invoking the type will fix the size, on the other hand in case of a constrained array the size is previously fixed, for example you might declared a type called byte and byte might be a constrained array of bits.

And now the size is pre-defined in the type itself and therefore a byte has to be an collection of 8 bits such arrays are called constrained arrays, so now we have seen the kind of data types which constitute this language many data types are pre-defined we have had a look at many of the important types.

We have not looked at accessing file types which are which we will see later if we cover this in this lecture, but as you use VHDL in actual usage you will become familiar with those types as well, but these are more fundamental types which you must be familiar with before you start using VHDL.

**(Refer Slide Time: 27:23)**

## Enumeration Type

VHDL *enumeration* types allow us to define a set of values that a variable of this type can acquire. For example, we can define a data type by the following declaration:

```
type instr is (add, sub, adc, sbb, rotl, rotr);
```

Now a variable or a signal defined to be of type `instr` can only be assigned values enumerated above – that is: `add`, `sub`, `adc`, `sbb`, `rotl` and `rotr`.

In actual implementation, these values may be mapped to a 3 bit value. However, an attempt to assign, say, '010' to a variable of type `instr` will result in an error. Only the enumerated values can be assigned to a variable of this type.

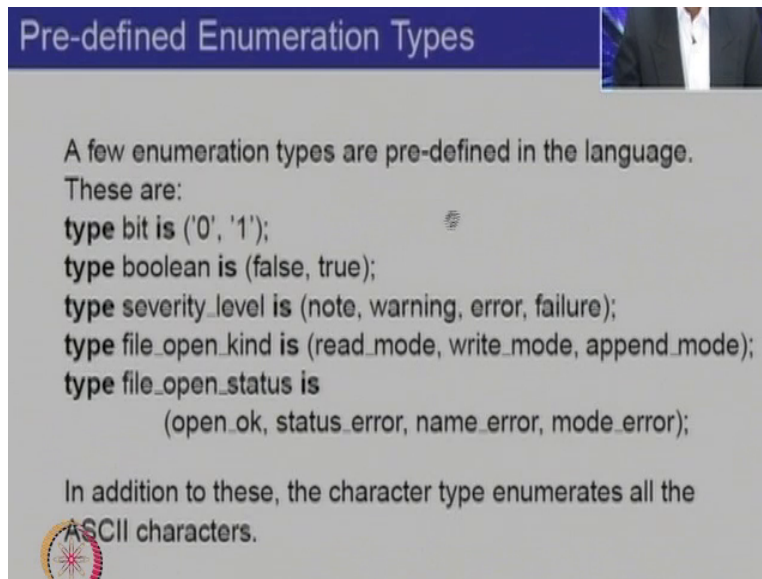
So enumeration type allows us to define a set of values that a variable of this type can acquire, for example we can define a data type by the following declaration, the keyword `type` must be used first so you say `type instr` instruction is and then you give a list of values which are enumerated, so for example I might say `type instr is add, subtract, add with carry, subtract with borrow, rotate left, rotate right`, this limited set of commands now has been enumerated.

And now variable or a signal which is defined to be of type of `instr` instruction can only be assigned values enumerated above, that means a particular signal says which is of type instruction can only acquire values which are `add`, `subtract`, `add with carry`, `subtract with borrow`, `rotate left` and `rotate right`, in actual implementation as I had said earlier these values maybe internally mapped to a 3 bit value.

Because we have only 6 possibilities however an attempt to assign a 3 bit value say 010 to a variable of type instruction will result in an error, it is for the VHDL to use 010 for one of these possibilities internally, but externally you must respect the enumeration that you had declared, so if you want the fourth here so assuming that this is 0, 1, 2, 3, 4 and 5 and internally rotate left might use the bit combination 010.

But direct assignment of 010 when you mean rotate left is wrong as for as the language is concerned, you must only assign the value `rotl` to this kind of variable, so only the enumerated values can be assigned to a variable of this type.

**(Refer Slide Time: 29:58)**



**Pre-defined Enumeration Types**

A few enumeration types are pre-defined in the language. These are:

```
type bit is ('0', '1');
type boolean is (false, true);
type severity_level is (note, warning, error, failure);
type file_open_kind is (read_mode, write_mode, append_mode);
type file_open_status is
    (open_ok, status_error, name_error, mode_error);
```

In addition to these, the character type enumerates all the ASCII characters.

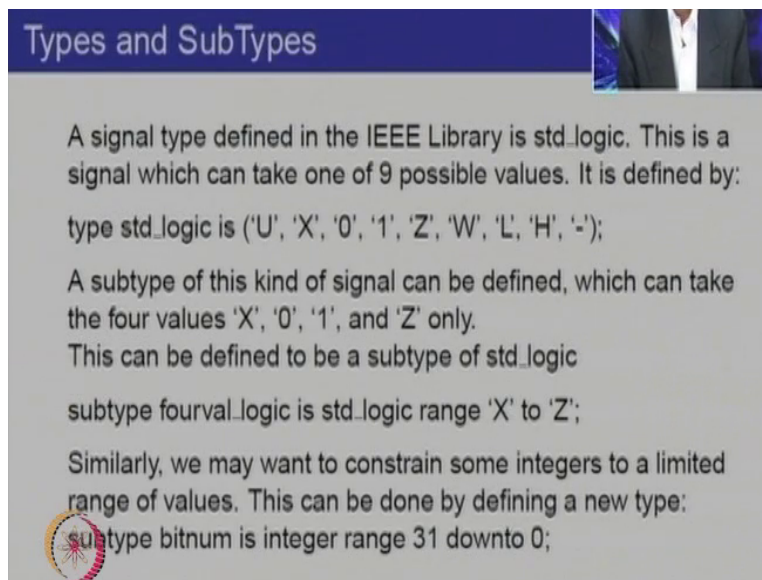
As we had said earlier, if you enumeration types are pre-defined you do not have to define this using a type statement, if you and enumeration types of pre-defined in the language these are this is effectively the declaration that you might have had to use had these not been pre-defined, type `bit` is 0 and 1, notice the quotation marks around 0 and 1, this distinguishes it from the integer 0 and 1.

When there is no cause of confusion then you can skip the quotation marks otherwise the quotation marks must be used, because 0 and 1 can be confused with the integers 0 and 1, the language requires you when you use them for a bit to put this quotation marks around 0 and 1, type `Boolean` is pre-defined and it can have the values `false` and `true`.

Type `severity level` is also pre-defined and it can only take these values `note`, `warning`, `error` and `failure` these are the four severity levels, these are severities of failure of assertion which are used by the language, type `file open kind` is also pre-defined as we had seen earlier and it can have one of these three values `read mode`, `write mode` or `append mode`.

The difference between write mode and append mode is that write mode begins from the start of the data structure, append mode starts adding from the end, file open status is also pre-defined and it can only take values open ok, status error, name error or more error. In addition to these the character type enumerates all the ASCII characters which exist, so these are the pre-defined enumeration types which are available to you.

**(Refer Slide Time: 32:15)**



The slide, titled "Types and SubTypes", contains the following text:

A signal type defined in the IEEE Library is `std_logic`. This is a signal which can take one of 9 possible values. It is defined by:

```
type std_logic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-');
```

A subtype of this kind of signal can be defined, which can take the four values 'X', '0', '1', and 'Z' only. This can be defined to be a subtype of `std_logic`

```
subtype fourval_logic is std_logic range 'X' to 'Z';
```

Similarly, we may want to constrain some integers to a limited range of values. This can be done by defining a new type:

```
subtype bitnum is integer range 31 downto 0;
```

Now notice that for each type you can define a subtype which does not use all the possible values defined in the main type, for example if you use the IEEE library it defines a new signal type called `std logic` which is quite commonly used, `std logic` is a signal which can take one of 9 possible values, it is defined by a statement of the type.

Type `std logic` is U which stands for undefined, X which is unknown, 0 1 obvious binary values, Z which is open circuit, W which is a weak version of X, L low which is the week version of 0, H or high which is a week version of 1 and do not care which is represented by a dash. So a signal can in fact take one of these 9 values, now suppose we do not want to use all 9 of these in our design, then you can define a subtype of this kind of signal.

The subtype let us say uses only these four values X, 0, 1 and Z indeed these are the four values which are used in Verilog as you might learn later, this can be defined to be a subtype of `std logic`, we might use the statement subtype we must give a name to this new subtype and what we

have chosen here is fourval logic, so we will say subtype fourval logic is std logic. Because it is a subtype it must refer to the parent type which is std logic, range X to Z that means it will take this subrange X to Z of the main type which is std logic.

Similarly, we may want to constraint some integers to a limited range of values this can be defined by a new subtype, for example we might say subtype bit num is integer is range 31 down to 0. So that means bit number is actually an integer bit number is actual integer subtype which has all the properties of integers with the additional constraint that it can only be in this range 31 to 0, when do we describe a new type and when should we describe a new subtype.

The advantage of using a subtype is that all the procedures which are defined for the main type are than inherited by the subtype. Otherwise you have to define your own procedures for a new type, for example end of std logic is defined by the type, now if you define a subtype then the end is inherited from the main type this inheritance is quite convenient and when you want to use this inheritance.

Then it is better to define a subtype if it is a completely independent type then there is no need to force a subtype declaration then you can declare a new type. But now you have the responsibility of declaring all the functions which are valid to operate on this type, so for example the bit num that we had seen with a restricted range of 31 to 0 was a subtype of integer and therefore operations defined an integer like addition, subtraction, multiplication and so on are directly inherited we do not have to define them all over again.

This makes it clear of about when to use a subtype and when to use a type. We should now look at physical types we had seen, so now we know that discrete types which are used.

**(Refer Slide Time: 33:10)**

## Physical Types

Objects which are declared to be of Physical type, carry a value as well as a unit. These are used to represent physical quantities such as time, resistance and capacitance.

The Physical type defines a basic unit for the quantity and may define other units which are multiples of this unit.

Time is the only Physical type, which is pre-defined in the language. The user may define other Physical types.



The physical types or objects which carry a value as well as a unit. So physical types are a data type which carry a value as well as a unit these are used to represent physical quantities such as time, resistance and capacitance, time is pre-defined, but we do have the option of defining our own physical type called resistance or capacitance. The physical type first defines a basic unit for the quantity and then may define other units which are multiples of this unit.

Time is the only physical type which is pre-defined in the language the user may define other physical types.

**(Refer Slide Time: 38:02)**

## Pre-defined Physical Type: Time

```
type time is range 0 to ...
```

```
  units
```

```
    fs;
```

```
    ps = 1000 fs;
```

```
    ns = 1000 ps;
```

```
    us = 1000 ns;
```

```
    ms = 1000 us;
```

```
    sec = 1000 ms;
```

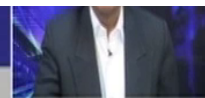
```
    min = 60 sec;
```

```
    hr = 60 min;
```

```
  end units time;
```



The user may define other physical types as required.

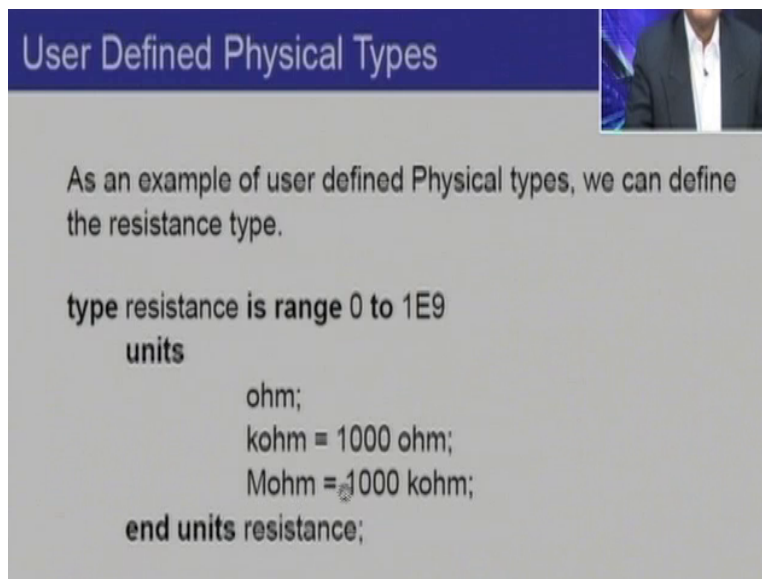


This is the pre-defined type physical and the pre-defined unit is femtoseconds, so you can say type this is how it must have been described internally, it is equivalent to a definition of this kind, type time is range 0 to some maximum value, then you have a declaration units and the base unit is declared first, this one is compulsory others are optional. So the base unit then is femtosecond.

And then we declare that a picosecond is in fact 1000 femtoseconds, a nanosecond is then 1000 picoseconds, a microsecond is 1000 nanoseconds, a millisecond is 1000 microseconds, second is 1000 milliseconds, a minute is 60 seconds and an hour is 60 minutes and this ends units for time. This is a complete definition of the type time internally again it will be kept as femtoseconds you may however assign a value to a time type object of let us say 500 microseconds.

This declaration makes it possible for the language to establish and equivalent between microseconds and the base unit which is femtoseconds it will be automatically converted to the base unit and then stored in base units, at the time of reporting it will be reported in the convenient units which you can choose, this is the pre-defined physical type the user may define other physical types as and when required.

**(Refer Slide Time: 40:16)**



The slide is titled "User Defined Physical Types" and features a small inset image of a person in a suit. The main content is a code snippet for defining a resistance type in a programming language. The code is as follows:

```
As an example of user defined Physical types, we can define
the resistance type.

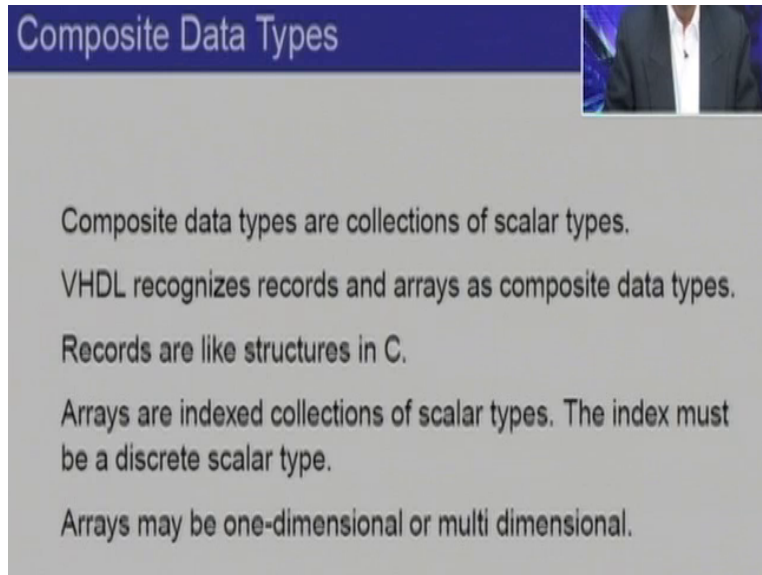
type resistance is range 0 to 1E9
  units
    ohm;
    kohm = 1000 ohm;
    Mohm = 1000 kohm;
  end units resistance;
```

Just to take an example, let us say we want to define a new type new physical type called resistance, we can declare it as saying type resistance is range 0 to 1E9 with units ohm that is the base unit and derived units which are Kohm or kilo ohm equal to 1000 ohm and Mohm or



megaohm to be 1000 kilohms and units resistance, with this block of declarations we now can use types which are in fact of type resistance.

**(Refer Slide Time: 41:08)**



Composite Data Types

Composite data types are collections of scalar types.

VHDL recognizes records and arrays as composite data types.

Records are like structures in C.

Arrays are indexed collections of scalar types. The index must be a discrete scalar type.

Arrays may be one-dimensional or multi dimensional.

Having done the scalar types let now look at composite data types, these are the types which consists of a base unit type and then what we declare is a collection of composite types, so composite data types are collection of scalar types, VHDL recognizes records and arrays as composite data types. Records are like structures in C, they permit a collection of heterogeneous kinds of scalars. Arrays are indexed collection of scalar types.

The index must be discrete scalar type, notice it need not be an integer, it can be any discrete scalar type and arrays maybe one-dimensional or multi-dimensional, so essentially if you have to put lots of scalar types together if they are all the same type then you can describe this collection as an array. If they are not of the same type, then you have to declare a record and describe what combination of dissimilar types will constitute that record.

And now you can put scalars of those dissimilar types into composite data types called records.

**(Refer Slide Time: 42:44)**

## Arrays

Arrays can be constrained or unconstrained.

- In constrained arrays, the type definition itself places bounds on index values. For example:

**type byte is array (7 downto 0) of bit;**

**type rotmatrix is array (1 to 3, 1 to 3) of real;**

- In unconstrained arrays, no bounds are placed on index values. Bounds are established at the time of declaration.

**type bus is array (natural range <>) of bit;**

The declaration could be:

**signal addr\_bus: bus(15 downto 0);**

**signal data\_bus: bus(7 downto 0);**

Arrays are particularly important and they can be constrained or unconstrained as we had seen earlier, in constrained arrays the type definition itself places bounds on index values, for example we might say type byte is array 7 down to 0 of bit or type rotation matrix rotmatrix is array 1 to 3, 1 to 3 of real, so it is a two dimensional array for a rotation matrix which will have sin theta, cos theta, sin pi, cos pi kind of entries therefore those are real.

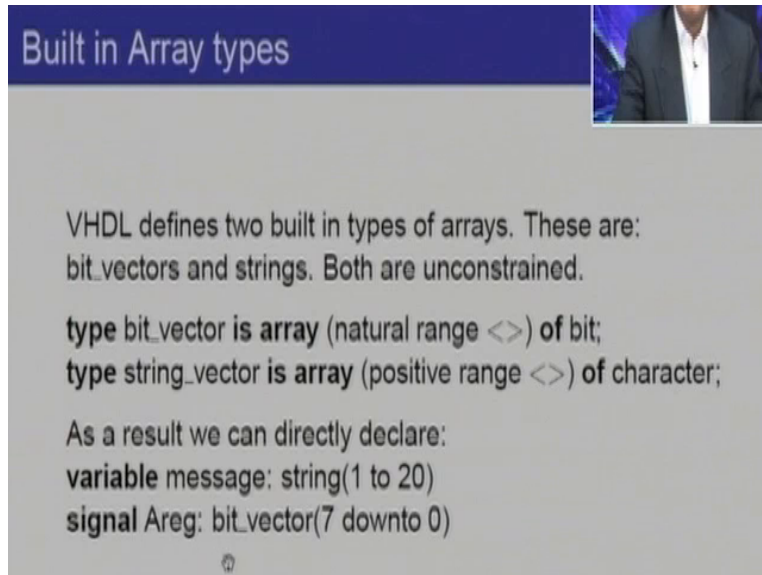
And we have defined a 3 by 3 array of this as a rotation matrix, notice that this size of these objects is now fixed. In unconstrained arrays no bounds are placed on index values, bounds are established at the time of declaration, for example we might declare a type bus to be an array unconstrained array of bits, then we will say type bus is array natural range this empty symbol which is essentially < any > concatenated of bit.

So here we are saying that the type bus is an array of bits it will be indexed by a natural number that means the index cannot be negative, however the range is left undefined in the type, when this type will be used the user will declare the range, for example when using it the declaration could be signal address bus is of type is bus 15 down to 0 that means the address bus is 16 bits wide.

However, the same type namely bus can be used for data bus and the declaration will be signal data bus is bus 7 down to 0, so it is at the time of usage that the size of this array has been

declared by intrinsic property this is unconstrained, the constraints are placed when it is actually used such arrays are called unconstrained arrays.

**(Refer Slide Time: 45:38)**



**Built in Array types**

VHDL defines two built in types of arrays. These are: bit\_vectors and strings. Both are unconstrained.

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

As a result we can directly declare:

```
variable message: string(1 to 20)  
signal Areg: bit_vector(7 downto 0)
```

There are as we had said earlier built-in array types, where bit vector is an unconstrained array of bit and string is an unconstrained array of character, as a result you can directly declare variables or signals of this kind for example you might have a declaration which says variable message is a string 1 to 20, notice string is an unconstrained array of characters and here we are saying that message will be a collection of 20 characters whose index will go from 1 to 20.

Similarly, we might declare signal A register Areg is a bit vector 7 down to 0, bit vector is a type already known to the language and it is an unconstrained array we are putting the constraint here 7 down to 0 and saying that this constraint type should be used for Areg, so this is how we use the built-in array types.

**(Refer Slide Time: 46:59)**

## Records



While an array is a collection of the same type of objects, a record can hold components of different types and sizes.

This is like a struct in C.

The syntax of a record declaration contains a semicolon separated list of fields, each field having the format name, ..., name : subtype

For example:

```
type resource is record  
(P_reg, Q_reg : bit_vector(7 downto 0); Enable: bit)  
end record resource;
```

Records are on the other hand collections of different types, so while an array is a collection of the same type of objects, record can hold components of different types and sizes, this is like a structure in C. And the syntax of a record declaration contains a semicolon separated list of fields, each field having the format name name name subtype.

For example type resource is record and then within brackets P reg, Q reg each of which is a bit vector 7 down to 0 and enable which is of the type bit, so this record resource now will always have two 8 bit registers and a bit called enable, so these types are in fact of different types and you can collect them together in a type called a record and you give it a name call resource.

And now you can declare objects of the type resource, where each one of those objects will contain two 8 bit registers and a 1 bit enable, so this can be used indeed later and we have seen one example of this, the port list is in fact a record it has different of signals some are input, others are output, some could be bits, others could be bit vectors and so on, and that collection is in fact a semicolon separated list of different types and each type of the same type is a comma separated list.

So that is how we have used the port list earlier and now we can recognize that in fact a port list is actually a record. With this I think we have reviewed the basic elements which constitute VHDL we have looked at object types, we have looked at data types, we have seen the various

in-built scalar data types, we have looked at the collections which are pre-defined, we have looked at physical types and we have seen also how you can declare your own types.

The hardware itself we have understood has templates which are entities with their corresponding architectures, a specific template can be chosen to define a component and once you have defined a component then you can instantiate actual instances of these components, various components can be interconnected using signals and signals can have various kinds they can be of scalar types or they can be of arrays like buses and so on and you have seen examples of all of these.

So armed with all these basic types, we can now see a few examples of hardware description language, I must emphasize here that in a brief course of lectures it is impossible to cover all the nuances of a language what we have done here is introduced you to the basic types and it is important to understand what these types represent and what they find distinctions among them is. This part is essentially required to be done in a lecture series.

We will use a few examples in a very brief survey of the VHDL language, but in order to learn VHDL descriptions you must use some standard text and you must use some actual programs which allow you to describe hardware using VHDL and we shall do this in the following lectures.