Hardware Modeling [VU] (191.011)
– WS24 –
Composite Types

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

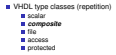
In this lecture we take a look at composite types in VHDL and will, thus, extend our knowledge about the VHDL type system.

Hardware Modeling [VU] (191.011)
– WS24 –
Composite Types

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

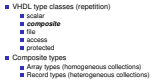
Modified: 2025-03-12, 16:24 (b25118c)



In the introduction lecture to the VHDL type system we already learned that there are five classes of types and discussed the scalar type in detail.

Introduction

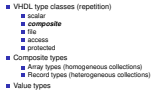
- VHDL type classes (repetition)
  - scalar
  - ***composite***
  - file
  - access
  - protected



In this lecture, we now take a closer look at composite types. The composite type class comprises array and record types, which we will simply refer to as arrays and records.

Introduction

- VHDL type classes (repetition)
  - scalar
  - ***composite***
  - file
  - access
  - protected
- Composite types
  - Array types (homogeneous collections)
  - Record types (heterogeneous collections)



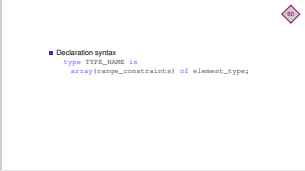
Please note that all types we have discussed so far and all types we will discuss in this lecture are value types. When a value type variable is assigned to another variable, a copy of the value is made. This means changes to one variable do not affect the other. This also holds true for composite types like arrays discussed in this lecture. Note that this is in contrast to many other programming languages, where arrays are often handled using pointers or references.

Introduction

- VHDL type classes (repetition)
  - scalar
  - ***composite***
  - file
  - access
  - protected
- Composite types
  - Array types (homogeneous collections)
  - Record types (heterogeneous collections)
- Value types

You should already be familiar with the concept of arrays from other programming languages. Arrays are data structures used to store multiple values of the same type. Individual elements in an array are accessed using an index. In many programming languages such as C or Java this index starts at zero. However, as we will learn in this lecture, VHDL is much more flexible in this regard, allowing array indices to be arbitrary continuous ranges of integer values.

Array Types

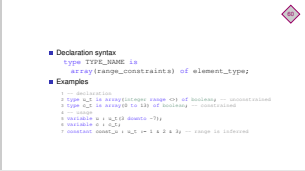


Array types are declared using the `array` keyword followed by one or more comma-separated integer range constraints. Specifying multiple range constraints allows the declaration of multidimensional arrays. Finally, we have to declare the actual element type of the array. This can be any scalar type, another array or a record type.

Array Types

- Declaration syntax

```
type TYPE_NAME is
  array(range_constraints) of element_type;
```



We don't want to get into too much detail regarding the formal specification of the array declaration syntax, as it is quite complex. Rather we want to show some example declarations and discuss their semantics. Array types in VHDL can be classified as either constrained or unconstrained. The code snippet on this slide shows an example for both variants. The first type named `u_t` is unconstrained, indicated by the symbol formed by the opening and closing angled brackets. Whenever this type is used in a variable declaration, it has to be constrained explicitly. In this case, this range constraint may use arbitrary integer-type limits and can be either ascending or descending. For constants, such as `const_u` in the example code, the story is a little different. Here, the range constraint can be omitted, as the compiler can infer it from the constant value itself. Constrained array types, such as `c_t`, can only be used with one particular range – the range they have been declared for. The array range constraints basically define the indices that can be accessed on arrays of that type. This means that arrays in VHDL don't always start with the index zero, as is the case for many other programming languages. For the variable "u", we can for example access the indices three, two all the way down to negative 7. Let us quickly revisit the declaration of the constant const-u. Its initialization value is a 3-element array comprising the integers 1, 2 and 3, specified using the concatenation operator. However, be aware that contrary to what you might expect, this code will not produce an array with the range 0 to 2. Since, the range of the underlying type u-type is specified as an integer range, the compiler will let the array start from the index given by the smallest value representable by the built-in integer type. This is something you should keep in mind when working with unconstrained types.

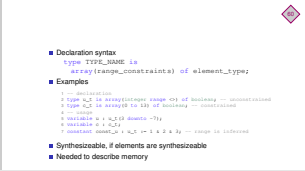
Array Types

- Declaration syntax

```
type TYPE_NAME is
  array(range_constraints) of element_type;
```

- Examples

```
1 -- declaration
2 type u_t is array(integer range <>) of boolean; -- unconstrained
3 type c_t is array(0 to 13) of boolean; -- constrained
4 -- usage
5 variable u : u_t(3 downto -7);
6 variable c : c_t;
7 constant const_u : u_t := 1 & 2 & 3; -- range is inferred
```



A VHDL array is synthesizeable if the element type it contains is synthesizeable. Indeed, arrays are heavily used in synthesizeable code to describe memories, such as RAMs, ROMs or FIFOs. However, this topic will be discussed in a dedicated upcoming lecture.

Array Types

- Declaration syntax

```
type TYPE_NAME is
  array(range_constraints) of element_type;
```

- Examples

```
1 -- declaration
2 type u_t is array(integer range <>) of boolean; -- unconstrained
3 type c_t is array(0 to 13) of boolean; -- constrained
4 -- usage
5 variable u : u_t(3 downto -7);
6 variable c : c_t;
7 constant const_u : u_t := 1 & 2 & 3; -- range is inferred
```

- Synthesizeable, if elements are synthesizeable
- Needed to describe memory

Let's now look at some of the predefined array types that can be found in the standard package.

Predefined Array Types

We have already encountered the built-in string type in previous lectures. In VHDL, strings are simply defined as arrays of characters, comparable to what is done in the C programming language. The string type is another example for an unconstrained array type. Note that the string range limits are of the integer-subtype positive, which only includes numbers grater and equal to one. This means that a string never has a character at index zero! For the example string constant shown on this slide, this results in an inferred range of 1 to 11.

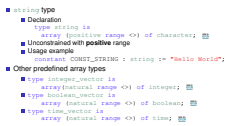
Predefined Array Types

- `string` type
  - Declaration

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

  - Unconstrained with **positive** range
  - Usage example

```
constant CONST_STRING : string := "Hello World";
```



Conveniently, starting with VHDL-2008 the standard package already defines array types for the predefined scalar types integer, real, boolean, bit, and time. The naming convention for these array types is the name of the scalar type underscore-vector. Note, that they are all unconstrained arrays with natural range limits.

Predefined Array Types

- `string` type
  - Declaration
    ```
    type string is
      array (positive range <>) of character;
    ```
  - Unconstrained with **positive** range
  - Usage example
    ```
    constant CONST_STRING : string := "Hello World";
    ```
- Other predefined array types
  - `type integer_vector is array(natural range <>) of integer;`
  - `type boolean_vector is array (natural range <>) of boolean;`
  - `type time_vector is array (natural range <>) of time;`

Now that we know, how arrays are declared, we can turn our attention to the question of how array elements are accessed.

Element Access

Element access syntax
Not via square brackets, i.e., [ ]
Parentheses with integer index as parameter
Similar to function calls

In contrast to many other programming languages, such as C, Java or Python, VHDL does not use square brackets to access array elements – in fact square brackets are not used at all in the VHDL syntax. It rather uses simple round parentheses which enclose the integer index of the element that is accessed. This, unfortunately, means that an array access is not always directly obvious in code, as a call to a function with a single parameter uses the exact same syntax.

Element Access

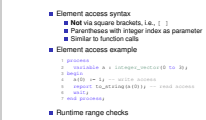
- Element access syntax
  - **Not** via square brackets, i.e., `[ ]`
  - Parentheses with integer index as parameter
  - Similar to function calls

The example code on this slide demonstrates how this works. The process first write the integer value 1 to the first index of the array ”a” and then uses a report statement to print it out.

Element Access

- Element access syntax
  - **Not** via square brackets, i.e., `[ ]`
  - Parentheses with integer index as parameter
  - Similar to function calls
- Element access example

```
1 process
2   variable a : integer_vector(0 to 3);
3 begin
4   a(0) := 1; -- write access
5   report to_string(a(0)); -- read access
6   wait;
7 end process;
```



As with the value ranges for integer and floating-point types, VHDL also performs static and runtime range checks when accessing arrays. This means that accessing an element that is outside the declared range of an array leads to a compilation error or the immediate termination of the simulator. However, please keep in mind that runtime range checks are only performed in simulation and not in hardware.

Element Access

- Element access syntax
  - **Not** via square brackets, i.e., `[ ]`
  - Parentheses with integer index as parameter
  - Similar to function calls
- Element access example

```
1 process
2   variable a : integer_vector(0 to 3);
3 begin
4   a(0) := 1; -- write access
5   report to_string(a(0)); -- read access
6   wait;
7 end process;
```

- Runtime range checks

Range access syntax: parentheses with integer range expression

In VHDL, it is also possible to index whole ranges of values. Again, parentheses are used, but instead of a single integer, an integer range expression has to be specified.

Element Access (cont'd)

- Range access syntax: parentheses with integer range expression

How this works in actual VHDL code is shown in the example snippet on this slide. The code first declares an array variable A, whose indices zero and one are set to the numbers 4 and 2, respectively. Now consider the right-hand-side of the assignment in line 6. The value range zero to one of variable A is read two times, each time returning an array of length two, which are then combined to an array of length 4 using the concatenation operator. The left-hand-side of the assignment, specifies a range of the variable A. comprising four elements as the target for the assignment. Hence, the text output of this process are the numbers 4 and 2 repeated three times.

Element Access (cont'd)

- Range access syntax: parentheses with integer range expression
- Range access example

```
1  process
2    variable a : integer_vector(0 to 5);
3  begin
4    a(0) := 4;
5    a(1) := 2;
6    a(2 to 5) := a(0 to 1) & a(0 to 1);
7    for i in 0 to 5 loop
8      report to_string(a(i));
9    end loop;
10   wait;
11 end process;
```

As already mentioned when we talked about the declaration syntax of arrays, it is possible to specify more than one range constraint, which results in multidimensional arrays. The first example on this slide shows how this can look like for a two-dimensional array. `a2d_t` represents an unconstrained two-dimensional array. Hence, when the array variable `a2d` is declared, the range limits have to be specified – note that it is valid to mix ascending and descending ranges. However, what is not valid is to mix constrained and unconstrained ranges in array declarations. To access elements of multidimensional arrays, multiple comma-separated indices are used in parentheses.

Multidimensional Array Examples

2D Array

```
1 process
2  type a2d_t is array(integer range <>, integer range <>) of boolean;
3  variable a2d : a2d_t(3 downto -4, 0 to 7);
4 begin
5  a2d(-4, 0) := true;
6  report to_string(a2d(-4, 0)); -- output: true
7  wait;
8 end process;
```

It is also possible to declare a multidimensional array, by declaring an array type whose element type is another array. This is demonstrated in the second code snippet on this slide. Notice, however, that in this case the individual array elements are accessed slightly differently, requiring two parenthesis expressions, instead of one with a comma.

Multidimensional Array Examples

2D Array

```
1 process
2  type a2d_t is array(integer range <>, integer range <>) of boolean;
3  variable a2d : a2d_t(3 downto -4, 0 to 7);
4 begin
5  a2d(-4, 0) := true;
6  report to_string(a2d(-4, 0)); -- output: true
7  wait;
8 end process;
```

Array of an Array

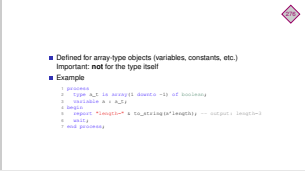
```
1 process
2  type aoa_t is array(integer range <>) of boolean_vector(0 to 7);
3  variable aoa : aoa_t(3 downto -4);
4 begin
5  aoa(-4)(0) := true;
6  report to_string(aoa(-4)(0)); -- output: true
7  wait;
8 end process;
```

Defined for array-type objects (variables, constants, etc.)
Important: **not** for the type itself

To obtain meta-information about arrays VHDL offers several predefined array attributes. These attributes are defined for all array-type objects such as variables or constants. This is in contrast to the attributes that we have seen for scalar types, which are defined for the respective types and not the objects of that type.

Array Attributes

- Defined for array-type objects (variables, constants, etc.)
  Important: **not** for the type itself



The example code on this slide shows how the length attribute of the array ”a” is accessed. The length attribute always returns the number of elements contained in an array, which in the case of the example is 3.

Array Attributes

- Defined for array-type objects (variables, constants, etc.)
  Important: **not** for the type itself
- Example

```
1 process
2   type a_t is array(1 downto -1) of boolean;
3   variable a : a_t;
4 begin
5   report "length=" & to_string(a'length); -- output: length=3
6   wait;
7 end process;
```

The attributes low, high, left, right and ascending have a similar function as the corresponding attributes for scalar types. For arrays, they return the respective range limits the array was defined for. For an exhaustive list on all array attributes, please refer to the VHDL standard.

Array Attributes

- Defined for array-type objects (variables, constants, etc.)
  Important: **not** for the type itself
- Example

```
1 process
2   type a_t is array(1 downto -1) of boolean;
3   variable a : a_t;
4 begin
5   report "length=" & to_string(a'length); -- output: length=3
6   wait;
7 end process;
```

- Other attributes: `low`, `high`, `left`, `right`, `ascending`, etc. (see VHDL standard)

Access an array's declared range

Let's take a closer look at one quite important attribute. As the name suggests, the range attribute allows to refer to the range an array is defined for.

Array Attributes – Range

- Access an array's declared range

In the code snippet it is used to iterate over all indices of the array A and print them alongside the actual value stored at that location. This way of iterating over the elements of an array is obviously preferable to hard-coding an explicit range in the for loop.

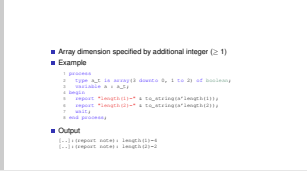
Array Attributes – Range

- Access an array's declared range
- Example

```
1 process
2   variable a : integer_vector(2 downto 0) := 1 & 2 & 3;
3 begin
4   report "range";
5   for i in a'range loop
6     report "index=" & to_string(i) & ", value=" & to_string(a(i));
7   end loop;
8   wait;
9 end process;
```

- Output

```
[...]: index=2, value=1
[...]: index=1, value=2
[...]: index=0, value=3
```



In case of multidimensional arrays, all array attributes can be passed an additional parameter in parentheses that selects the dimension for which the respective attribute should be read. The example code shows how this works for the length attribute. Note, that the first dimension has the index 1.

Array Attributes – Multidimensional Arrays

- Array dimension specified by additional integer (≥ 1)
- Example

```
1 process
2   type a_t is array(3 downto 0, 1 to 2) of boolean;
3   variable a : a_t;
4 begin
5   report "length(1)=" & to_string(a'length(1));
6   report "length(2)=" & to_string(a'length(2));
7   wait;
8 end process;
```

- Output

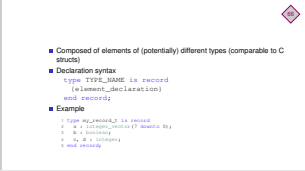
```
[..]:(report note): length(1)=4
[..]:(report note): length(2)=2
```

Composed of elements of (potentially) different types (comparable to C structs)

OK, now let's turn our attention to the other class of composite types – records. Record types are the VHDL-version of heterogeneous composite data types and can be compared to structs in the C programming language. A record can be composed of an arbitrary number of elements of different data types, such as built-in types like integer, boolean or time, enums, arrays or even other records. They allow to group data that logically belongs together and, hence, make code more structured and maintainable.

Record Types

- Composed of elements of (potentially) different types (comparable to C structs)



Record types are declared using the `record` keyword, followed by a list of element declarations. An element declaration is given by an identifier – the element name – followed by a colon, a type and a semicolon. If multiple elements with the same type have to be declared it is also possible to use a comma-separated list of identifiers, instead of a single one. This is also demonstrated by the elements c and d in the example record shown on this slide.

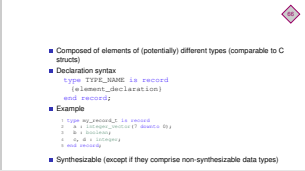
Record Types

- Composed of elements of (potentially) different types (comparable to C structs)
- Declaration syntax

```
type TYPE_NAME is record
  {element_declaration}
end record;
```

- Example

```
1 type my_record_t is record
2   a : integer_vector(7 downto 0);
3   b : boolean;
4   c, d : integer;
5 end record;
```



Generally, records are fully synthesizable. However, if they contain elements of data types that are not synthesizable, like time or real, then the resulting record is also not synthesizable.

Record Types

- Composed of elements of (potentially) different types (comparable to C structs)
- Declaration syntax

```
type TYPE_NAME is record
  {element_declaration}
end record;
```

- Example

```
1 type my_record_t is record
2   a : integer_vector(7 downto 0);
3   b : boolean;
4   c, d : integer;
5 end record;
```

- Synthesizable (except if they comprise non-synthesizable data types)

Element access syntax: dot operator

To access the elements of a record the dot operator is used.

Element Access

- Element access syntax: dot operator

In the example code snippet the record type `vec2_t` is declared, which represents vector in 2D space. Then this type is used to declare the variable `v` whose elements `x` and `y` are then assigned the values one and two.

Element Access

- Element access syntax: dot operator
- Example

```
1  process
2    type vec2_t is record
3      x, y : real;
4    end record;
5    variable v : vec2_t;
6  begin
7    v.x := 1.0;
8    v.y := 2.0;
9    report "x=" & to_string(v.x) & "," &
10          "y=" & to_string(v.y); -- output: x=1.0, y=2.0
11   wait;
12 end process;
```

Records can have elements of unconstrained types

It is possible to declare a record with unconstrained element types – for example an integer vector without a range constraint.

Unconstrained Elements

- Records can have elements of unconstrained types

Records can have elements of unconstrained types
Constrain types when record is used

In such a case, the range of these unconstrained elements have to be specified when a variable of that record type is declared.

Unconstrained Elements

- Records can have elements of unconstrained types
- Constrain types when record is used

This is shown in the example code snippet on this slide. The record `ur_t` contains two unconstrained elements – A and B. Hence, when the variable v is declared, range limits for the elements A and B have to be provided. If those range constraints would be omitted the compiler outputs an error. For constants the range constraints are not necessary, as they are inferred automatically.

Unconstrained Elements

- Records can have elements of unconstrained types
- Constrain types when record is used
- Example

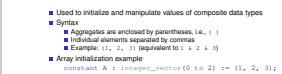
```
1  process
2    type ur_t is record
3      a : integer_vector; -- unconstrained
4      b : boolean_vector; -- unconstrained
5      c : real_vector(1 downto 0); --constrained
6    end record;
7    variable v : ur_t(a(1 downto 0), b(0 to 7));
8  begin
9    wait;
10 end process;
```

- Used to initialize and manipulate values of composite data types

Before we close this lecture, we want to introduce an important and powerful VHDL language feature named aggregate expressions. Aggregate expressions – for short just aggregates – can be used to initialize and manipulate composite data structures – such as arrays and records – in a quite compact and efficient way.

Aggregate Expressions

- Used to initialize and manipulate values of composite data types

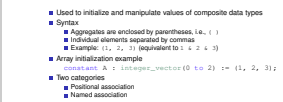


Again, we don't aim to discuss the formal syntax specification for aggregate expressions in detail. Hence, in the following we look at some examples to give you a general "feel" for aggregates and how they can be used. Generally aggregates are always enclosed by parentheses, and the individual elements are separated by commas. The simple example on this slide could be used to initialize a 3-element integer array.

Aggregate Expressions

- Used to initialize and manipulate values of composite data types
- Syntax
  - Aggregates are enclosed by parentheses, i.e., `( )`
  - Individual elements separated by commas
  - Example: `(1, 2, 3)` (equivalent to `1 & 2 & 3`)
- Array initialization example

```
constant A : integer_vector(0 to 2) := (1, 2, 3);
```



Aggregates can be classified, depending on whether they use positional or named associations. The example on this slide uses positional association. On the following slides we will take a closer look at both categories.

Aggregate Expressions

- Used to initialize and manipulate values of composite data types
- Syntax
  - Aggregates are enclosed by parentheses, i.e., `( )`
  - Individual elements separated by commas
  - Example: `(1, 2, 3)` (equivalent to `1 & 2 & 3`)
- Array initialization example

```
constant A : integer_vector(0 to 2) := (1, 2, 3);
```

- Two categories
  - Positional association
  - Named association

Positional association for arrays is quite straight-forward to understand. Each element in the aggregate expression is assigned to the respective item in the array starting from the left. Many programming languages have similar language constructs to initialize arrays. In the first example block, the element at index 1 of the boolean vector A gets assigned the value true, while index 0 gets assigned false. For the constant B the left-most index is 2, hence, it is assigned the value 1. Note that for the initialization of arrays of enum types that use character literals, such as string or bit-vector, VHDL supports a shorthand notation using double quotes. For string types we have already seen this notation on previous slides and lectures. We have included this equivalent shorthand notation for the constants C and D of the example.

Positional Association

- Array examples

```
1 constant A : boolean_vector(1 downto 0) := (true, false) -- a(1)=true
2 constant B : integer_vector(2 to 4) := (1, 2, 3); -- b(2)=1
3 constant C : string(1 to 3) := ('a', 'b', 'c'); -- equivalent to "abc"
4 constant D : bit_vector(2 downto 0) :=
5     ('1', '0', '1'); -- equivalent to "101"
```

Positional association also works for records, as shown in the second example block. Here the values are simply assigned based on their position in the record type declaration. Hence, the element X in the constant V is assigned the value 1, while Y and Z are set to zero. The example of the constant named demo also shows that aggregate expressions can also be nested. However, for complex records with many heterogeneous elements – such as demo-type – we would rather discourage you from using positional association, and rather use the named association that will be discussed on the following slides.

Positional Association

Array examples

```
1 constant A : boolean_vector(1 downto 0) := (true, false) -- a(1)=true
2 constant B : integer_vector(2 to 4) := (1, 2, 3); -- b(2)=1
3 constant C : string(1 to 3) := ('a', 'b', 'c'); -- equivalent to "abc"
4 constant D : bit_vector(2 downto 0) :=
5     ('1', '0', '1'); -- equivalent to "101"
```

Record examples

```
1 type vec3d_t is record
2   x, y, z : real;
3 end record;
4 constant V : vec3d_t := (1.0, 0.0, 0.0);
5
6 type demo_t is record
7   a : integer_vector(1 downto 0);
8   v : vec3d_t;
9   b : boolean;
10 end record;
11 constant DEMO : demo_t := ((1, 2), V, true);
```

With named association, values are explicitly assigned by specifying the names of the fields in records or the indices in an array. In code the right arrow operator is used for this purpose.

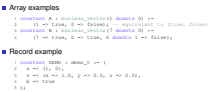
Named Association

Let's first look at some examples how this works for array types. For the constant "A" the first index is set to true, while the second one is set to false. On first sight this might seem unnecessarily cumbersome, especially when we compare it to the code that would be required for a positional association. However, as shown for the constant B, the left part of the arrow operator can also be a range expression. In this example, the first and the last array element are set to true, while the ones in the middle are set to false.

Named Association

- Array examples

```
1 constant A : boolean_vector(1 downto 0) :=
2     (1 => true, 0 => false); -- equivalent to (true, false)
3 constant B : boolean_vector(7 downto 0) :=
4     (7 => true, 0 => true, 6 downto 1 => false);
```



For record types the name of the element is used on the left side of the arrow. In the example code, we use the same record type declarations, as shown on the previous slide and initialize the constant demo to the same value. Notice that now it is much clearer and more explicit which element of the record gets assigned which value. Moreover, should the sequence of elements be changed in the record type declaration, the named association is still valid and won't break as a positional one would.

Named Association

- Array examples

```
1 constant A : boolean_vector(1 downto 0) :=
2     (1 => true, 0 => false); -- equivalent to (true, false)
3 constant B : boolean_vector(7 downto 0) :=
4     (7 => true, 0 => true, 6 downto 1 => false);
```

- Record example

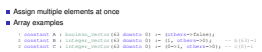
```
1 constant DEMO : demo_t := (
2   a => (1, 2),
3   v => (x => 1.0, y => 0.0, z => 0.0),
4   b => true
5 );
```

Assign multiple elements at once

Finally, let's take a quick look at the "others" clause that can be used in both named and positional associations – but only as the last element. Using the `others` keyword, it is possible to assign all elements that have not yet been assigned using a positional or named association in a composite data structure.

Others Clause

- Assign multiple elements at once

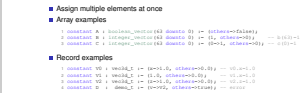


The constant A in the array example code uses the ”others” clause to initialize all elements of the array to false. The constant B and C only set the first and last elements to one, respectively. All other values are set to zero. Notice, that all these expressions are still valid if the left limit of the range expression would be changed.

Others Clause

- Assign multiple elements at once
- Array examples

```
1 constant A : boolean_vector(63 downto 0) := (others=>false);
2 constant B : integer_vector(63 downto 0) := (1, others=>0);     -- b(63)=1
3 constant C : integer_vector(63 downto 0) := (0=>1, others=>0); -- c(0)=1
```



For record types, the ”others” clause does not always make a lot of sense, since for it to be valid the unassigned elements must all have the same type. The last example on the slide, where a constant ”D” of the demo type is declared, illustrates this. Since its other fields are of different types, trying to set them via others results in an error. However, for some records like the 3-element vector type, already used on the previous slides, it can be helpful.

Others Clause

- Assign multiple elements at once
- Array examples

```
1 constant A : boolean_vector(63 downto 0) := (others=>false);
2 constant B : integer_vector(63 downto 0) := (1, others=>0);     -- b(63)=1
3 constant C : integer_vector(63 downto 0) := (0=>1, others=>0); -- c(0)=1
```

- Record examples

```
1 constant V0 : vec3d_t := (x=>1.0, others=>0.0); -- v0.x=1.0
2 constant V1 : vec3d_t := (1.0, others=>0.0);    -- v1.x=1.0
3 constant V2 : vec3d_t := (z=>1.0, others=>0.0); -- v2.z=1.0
4 constant D  :  demo_t := (v=>V2, others=>true); -- error
```

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

Lecture Complete!