#### UHDL Type System

In this lecture we take a first look at the type system of VHDL. We go through commonly used basic built-in types and discuss how to define custom ones. Upcoming lectures then go into further details on composite types and other more advanced topics.



Types Scalar Types Attributes Subtypes

#### Hardware Modeling [VU] (191.011) - WS24 -VHDL Type System

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

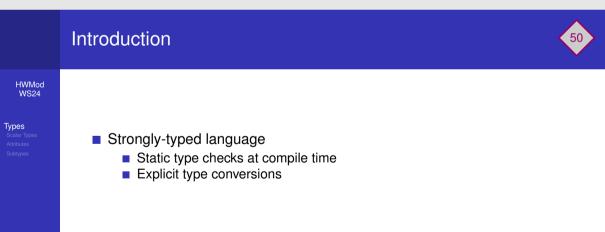
#### WS 2024/25

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

Strongly-typed language
 Static type checks at compile time
 Explicit type conversions

50

VHDL has quite a rich and powerful type system. It is a strongly typed language, which means that strict type rules are enforced at compile time, similar to what is done in languages like Java, Rust or Ada and unlike C or JavaScript. Type conversions are always explicit. This means that it is, for example, not possible to assign a value of type boolean to a variable of type integer without first converting the value. This allows for many errors to be caught early during compile time.



Strongly-typed language

 Static type checks at compile time
 Explicit type conversions

 Type diclaration syntax

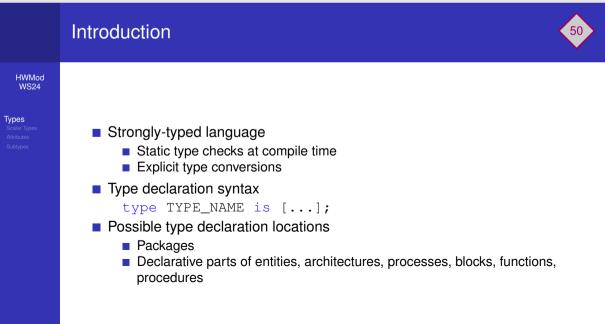
 type TYPE\_NAME is [...];

50

New custom types are declared using the type keyword. In this course, we use the suffix underscore-t to name custom types. We employ this naming convention to clearly mark identifiers as type names and distinguish them from built-in types.



Often times, type declarations are put into packages, such that they can be used in multiple modules and on interfaces between modules. However, depending on the scope you want your custom type to have, it can also make sense to declare them in the declarative parts of entities, architectures, processes or sub-programs.





50

#### Introduction (cont'd)

VHDL contains several basic built-in or predefined types, that we have already encountered in the previous lecture. The declarations of all predefined types can be found in the standard package, which we have also linked in the slides. Note, however, that this package is not loaded or parsed as regular VHDL packages - if you follow the link you can see that a lot of the code is actually commented out. You can rather think of it as a formal specification of all the built-in types and the operations that can be performed on them.



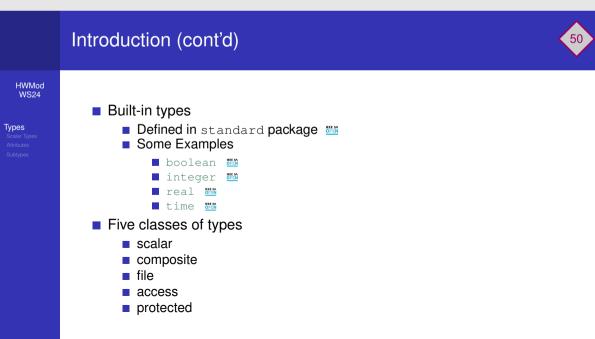
50

#### Introduction (cont'd)

Built-in types
 Control in standard package
 Sometin take
 Sometin take

50

VHDL distinguishes five classes of types, namely scalar, composite, file, access and protected types. Built-in types like integer, boolean or real fall into the scalar category. Composite types comprise arrays and records, representing homogeneous and heterogeneous collections, respectively. Composite types as well as File, access and protected types will be discussed in upcoming lectures.



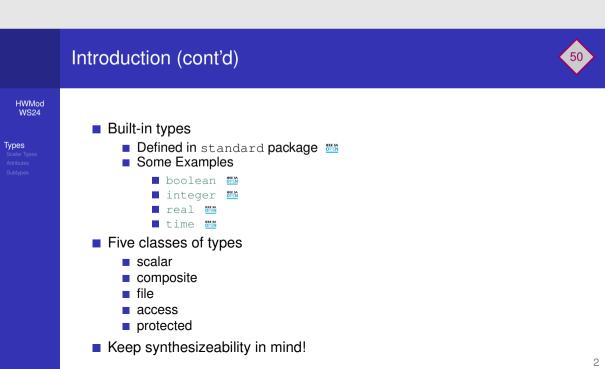
UHDL Type System

#### Introduction (cont'd)

Buthin types
 Uditrian (in strandsref package)
 Some Examples
 Some Examples
 Some Transition
 Some Transi

50

As you might have already suspected, not all types can be used when describing hardware. Naturally, it is not possible to simply write to a file, when you are effectively describing a digital circuit. However, also some of the seemingly unsuspicious built-in types like real are also not synthesizeable. In the following slides, we will now take a closer look at scalar types.



#### └─VHDL Type System └─Scalar Types └─Integer Types

Scalar types are further divided into integer, floating-point, enumeration and physical types. Let's start with integer types.

#### Integer Types

HWMod WS24

Types Scalar Types Integers Floats Enums Physical Type Attributes Subtypes



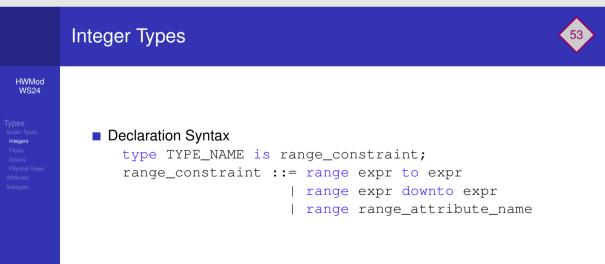
53

**\$3** 

#### └─VHDL Type System └─Scalar Types └─Integer Types

(33)

Integer types are declared using a simple range constraint, that specifies the values the declared type can hold. A range constraint is introduced using the range keyword, followed by two expressions separated by either the to or the downto keyword, representing ascending and descending ranges, respectively. Furthermore, it is also possible to define a range using a range attribute. However, this is a language construct, which is yet to be discussed in an upcoming lecture.



#### └─VHDL Type System └─Scalar Types └─Integer Types

Deckration Syntax
 type ITTZ\_NAME is range\_column raiser;
 r

Note that for ascending ranges the numerical value of the left expression must be smaller or equal to the value of the right expression. For descending ranges the situation is reversed.

## Integer Types 53 HWMod WS2d ■ Declaration Syntax type TYPE\_NAME is range\_constraint; range\_constraint ::= range expr to expr | range expr downto expr | range range\_attribute\_name ■ Left/right expressions must be integers ■ Ranges are either ascending or descending ■ Ascending (to): left ≤ right ■ Descending (downto) left ≥ right

#### └─VHDL Type System └─Scalar Types └─Integer Types (cont'd)

Examples
 B bit unsigned integer
 type uintE\_t is range 0 to 255;
 the bit signed integer
 type 11tE\_t is range -2\*15 to 2\*\*15-1;
 integer that can only hold values between 42 and 13
 type descending\_int\_t is range 44 to 4000 13

This slide shows some examples for integer type declarations. The first two examples define types able to represent the typical value ranges for unsigned 8-bit and a signed 16-bit integer values. They both use ascending ranges while the last example uses a descending one.

# State S

#### └─VHDL Type System └─Scalar Types └─Integer Types (cont'd)

Example
 End to depart integr
 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

 End to depart integr

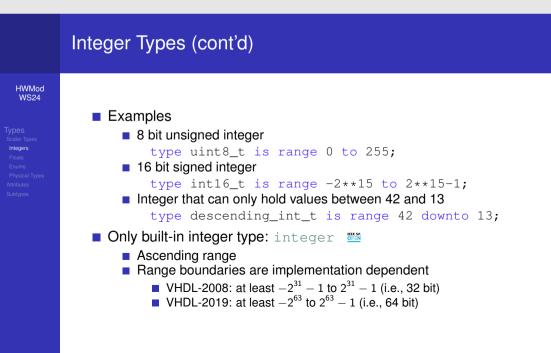
 End to depart integr

 End to depart integr

 End to depart integr

 End to

In VHDL, there is only one predefined integer type, simply called integer. While it always has an ascending range, its left and right range boundaries are implementation dependent, which means that they may vary between simulation and synthesis tools of different vendors. However, the different version of the VHDL standard guarantee certain minimum ranges, which are listed on this slide.



#### └─VHDL Type System └─Scalar Types └─**Range Contraints**

Range constraints are checked statically (i.e., at compile time) are checked dynamically during simulation (i.e., runtime) and checked in hardware

Range constraints are a strong safety feature of VHDL, as they are checked both at compiler and run time by the simulator. This allows to catch possible errors early during development, before going to actual hardware. We will see that range constraints also come up for other type classes later. However, keep in mind that run time checks are not performed in hardware. In hardware the result of incrementing an integer that is already at its maximum value will lead to an overflow, possibly even resulting in a value that is outside of the defined range for the respective type.

#### Range Contraints

HWMod WS24 Range constraints

- are checked statically (i.e., at compile time)
- are checked dynamically during simulation (i.e., runtime)
- not checked in hardware

Types Scalar Types Integers Floats Enums Physical Typ Attributes Subtypes

#### └─VHDL Type System └─Scalar Types └─**Range Contraints**

Range constraints
 are checked statically (i.e., at comple time)
 are checked synamically during simulation (i.e., runtime)
 are to checked in hardware
 Example: statication cange errors
 associate A = ultrat\_c = -1+ - range errors -1 < 0
 associate A = ultrat\_c = -1+10+ - range errors -1 < 0

This slide, shows some code examples that lead to range errors. The problems with the constant and variable declarations can easily be caught by the compiler, as the initial value is clearly outside the range of the used type.

#### **Range Contraints**

2

#### HWMod WS24

#### Range constraints

- are checked statically (i.e., at compile time)
- are checked dynamically during simulation (i.e., runtime)
- not checked in hardware

#### Example: static range errors

```
1 constant A : uint8_t := -1; -- range error: -1 < 0</pre>
```

variable b : uint8\_t := 16\*16; -- range error: 256 > 255

Types Scalar Typ Integers Floats Enums Physical

#### └─VHDL Type System └─Scalar Types └─**Range Contraints**

Range constraints
 endowed dynamically data is completely that
 endowed dynamically data guardiator (i.e., unstance)
 endowed have
 endowed and have
 endowed

The overflow error in the process in the second code snippet will only be detected during runtime, and will cause the simulation to be stopped immediately. This means that the second report statement will not be executed.

#### **Range Contraints** HWMod Range constraints **WS24** are checked statically (i.e., at compile time) are checked dynamically during simulation (i.e., runtime) not checked in hardware Integers Example: static range errors constant A : uint8\_t := -1; -- range error: -1 < 0</pre> 1 variable b : uint8\_t := 16\*16; -- range error: 256 > 255 2 Example: dynamic range error 1 process 2 variable x : uint8\_t := 254; 3 begin 4 x := x + 1; report "everything fine!"; 5 6 x := x + 1;7 report "not fine" -- never executed because of the range error 8 end process;

#### └─VHDL Type System └─Scalar Types └─**Floating-Point Types**

Declaration Syntax
 type TYPE\_NAME is range\_constraint;
Example
 type my\_fp\_t is range 0.0 to 1.0;

Floating-point types are quite similar to integers, as they are also declared using a range constraint.

## Figure 1 and the second sec

#### └─VHDL Type System └─Scalar Types └─**Floating-Point Types**

 Deckration Syntax type TTPE\_NAME is range\_constraint;
 Example
 type my\_fp\_t is range 0.0 to 1.0;
 Ordy built moding point type: real the
 ange is implementation dependent
 most Heavy a 64 to (a. double precision) floating point 57

Again, there is only one predefined floating-point type named real. The range of this type is defined by the tool implementation and depends on the actual floating-point types available on a given system.

# Floating-Point Types Floating-Point Types Prove Type Prove T

#### └─VHDL Type System └─Scalar Types └─**Floating-Point Types**

 Declaration Syntax type TTPE\_UNAME is range\_constraint;
 Example
 Example
 type my\_fp\_ts is range 0.0 to 1.0;
 Cody built holdsrippionit type: cmal 
 a range is implementation dependent most likely set bit (0.4 codule precision) floating-point
 Range checks as with integers 57

Exactly as with integers, floating-point values are also checked if they are within the range constraint of the underlying type declaration.

#### Floating-Point Types Float F

#### -VHDL Type System -Scalar Types -Floating-Point Types

 Declaration Syntax
 THE MAKE is range\_com Example xample
type my\_fp\_t is range 0.0 to 1.0;
nly built-in floating-point type: real 
#
range is implementation dependent
most likely a 64 bit (i.e., double precision) floati Range checks as with integers Cannot be used in synthesizeable code

Keep in mind, that floating-point types are not synthesizeable! They can only be used in static compile time expressions - for example as a generic that controls some behavior of a module - or in simulation code!

### Floating-Point Types HWMod **WS24** Declaration Syntax

type TYPE\_NAME is range\_constraint;

#### Example

Floats

type my\_fp\_t is range 0.0 to 1.0;

- Only built-in floating-point type: real
  - range is implementation dependent
  - most likely a 64 bit (i.e., double precision) floating-point
- Range checks as with integers
- Cannot be used in synthesizeable code!





57

Let's now take a look at enumeration types. Enumeration types – often simply referred to as enums – are present in many programming languages, such as C, Java or Rust. They represent finite sets or sequences of named constant values. Generally, enums make code more readable and maintainable and reduce the potential for errors, as they allow to replace arbitrary numeric or string constants with meaningful names.

#### **Enumeration Types**

HWMod WS24

Types Scalar Types Integers Floats Enums Physical Typ Attributes 52

B Declaration Syntax type TYPE\_NAME is (enum\_literal {, enum\_literal});

In VHDL Enumeration types are declared using a list of comma-separated enumeration literals in parentheses. An enumeration literal can be an identifier or a character literal in simple quotes – it is also possible to combine both variants in a single enumeration.

#### Enumeration Types WMod Ws24 Types State Types Types Types Types Types Types Types\_NAME is (enum\_literal {, enum\_literal});

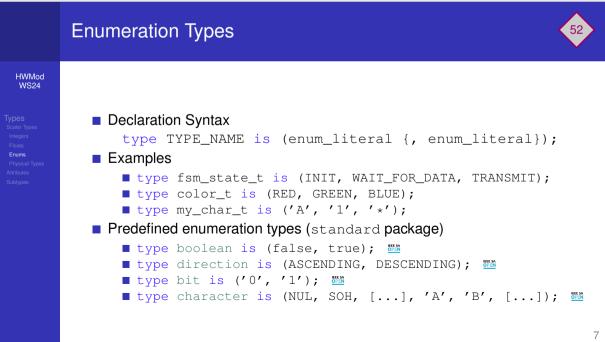
Declaration Syntax type TITE\_NAMME is (enum\_literal {, enum\_literal}); bramples = type fm\_state\_t is (INIT\_VALT\_TOR\_DATA, TRANSMIT); = type color\_t is (RED\_ GREEN, BLUE); = type my\_lmate\_t is (X^\*, Y^\*, Y^\*);

One important use-case for enums, that we will encounter in this course, is in the implementation of state machines. Here, enums are used to give descriptive names to the various states of an FSM, which – as already mentioned – makes code more readable and maintainable.

# Fundamentation Types 52 HWMod WS24 Image: Declaration Syntax type TYPE\_NAME is (enum\_literal {, enum\_literal}); Exams Prove Types Image: type TYPE\_NAME is (enum\_literal {, enum\_literal}); Examples Image: type fsm\_state\_t is (INIT, WAIT\_FOR\_DATA, TRANSMIT); Image: type my\_char\_t is ('A', '1', '\*');

Decknotich Syntax Decknotich Syntax Decknotic Syntax (second, literal); Syntax (second, literal); Syntax (second, literal); Professional (second, literal); Pro

However, also a lot of important predefined types in VHDL are actually enums. On this slide, we have listed some of them from the standard package. As you can see, the boolean type is defined as an enum, which simply contains the literals true and false. This means, that unlike in programming languages like C or Python, booleans and integers are fundamentally different types. Because of the strict type system, it is, hence, not possible to assign a boolean value to an integer or vice versa, without a proper type conversion.



The last class of the scalar types are physical types. As the name already suggests, they are used in VHDL to represent some form of physical unit. Hence, expressions with physical types always involve a numeric value as well as a unit.

#### Physical Types



Types Scalar Types Integers Floats Enums Physical Types Attributes Subtypes





Declaration Syntax type TTPE\_NAME is range\_constraint units primary\_unit {secondary\_init} end units: To declare a physical type we need an integer range, similar to what is done for an integer type declaration, followed by the units keyword. First the primary unit is declared, followed by an arbitrary number secondary units, that define how the different units relate to each other.

#### Physical Types



Types Scalar Types Integers Floats Enums Physical Types Attributes Subtypes

#### Declaration Syntax

type TYPE\_NAME is range\_constraint units
 primary\_unit
 {secondary\_unit}
end units;



B Occursion Syntax type TTFX\_UNKE is range\_constraint units primary\_unit end units\_primat end units\_primat may applicate unit may applicate unit may applicate unit may applicate unit may applicate units end units; end units;

The example on this slide shows how this can look like for a physical type storing distance values. Note that every value you want to express using the physical type must be representable using the primary unit under the given range constraint. This means that the highest value that can be expressed in the distance type is one meter, because one meter is equal to 1 billion nanometers.

#### Physical Types

Declaration Syntax

#### HWMod WS24

Types Scalar Types Integers Floats Enums Physical Types Attributes Subtypes

### 54

## primary\_unit {secondary\_unit} end units; Example type distance\_t is range 0 to 1\_000\_000\_000 units nm; -- primary unit um = 1000 nm; mm = 1000 um; -- secondary units cm = 10 mm; m = 1000 mm; -- secondary units end units;

type TYPE\_NAME is range\_constraint units



The only physical type that we are going to need during this course is the predefined type time.

#### 

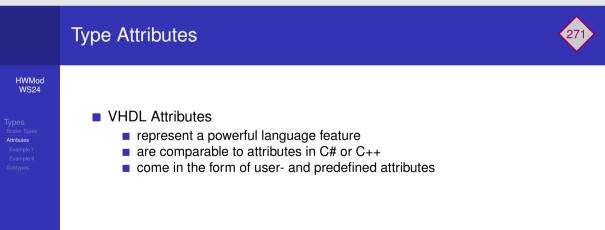
Predefined physical type (standard Package): time

#### └─VHDL Type System └─Type Attributes └─**Type Attributes**

VHDL Attributes
 represent a powerful language feature
 are comparable to attributes in C# or C++
 come in the form of user- and predefined attributes

271

Alright, now that we've covered all four scalar type categories, we can move on to an important type-related feature in VHDL known as attributes. Attributes in VHDL can be roughly compared to annotations in Java or attributes in C# or C++, although they work quite differently in VHDL. VHDL distinguishes user-defined and predefined attributes. User-defined attributes are defined to be constants of arbitrary type that can be attached to almost every object in VHDL – for example entities, signals, types, subprograms and many more. Future lectures will present some examples for that.



#### └─VHDL Type System └─Type Attributes └─**Type Attributes**

VHDL Attributes represent a powerful language teature are comparable to attributes in C# or C++ come in the form of user- and predefined attribute Type attributes allow to e obtain meta-information about a type perform operations with values of a type 271

This lecture focuses on predefined attributes and more specifically predefined type attributes. Predefined type attributes can either be simple constants that contain meta-information about a type, but can also be special functions that perform some operation with values of a specific type.

# Type Attributes Wide <

#### └─VHDL Type System └─Type Attributes └─**Type Attributes**

VHDL Althouts
 Properties a provint language leasure
 encore in the order second productions attracts
 to come in the order second productions attracts
 to come in the common deal of a type
 encore in the common deal of a type
 for the common deal
 for the common

271

A single quotation symbol is used to access or invoke an attribute. If an attribute is a function additional parameters are passed in parentheses.

#### **Type Attributes** HWMod **WS24** VHDL Attributes represent a powerful language feature Attributes are comparable to attributes in C# or C++ come in the form of user- and predefined attributes Type attributes allow to obtain meta-information about a type perform operations with values of a type Syntax Accessing attribute a of type T: T' a Example: integer' low Invoking attribute function f of type T with argument x: T' f (x) Example: integer' image(x)

#### -VHDL Type System -Type Attributes -Type Attributes

represent a powerful language fei are comparable to attributes in Ci come in the form of user- and pre a attributes allow to c of type IT with aroument x: IT' IT (x) 271

OK, so far this all sounds very abstract. Let's now see what we can do with type attributes. We will do this by looking at some code examples that showcase how attributes are used in practice.

#### Type Attributes HWMod **WS24** VHDL Attributes represent a powerful language feature Attributes are comparable to attributes in C# or C++ come in the form of user- and predefined attributes Type attributes allow to obtain meta-information about a type perform operations with values of a type Syntax Accessing attribute a of type T: T' a Example: integer' low Invoking attribute function f of type T with argument x: T' f (x) Example: integer' image(x)

Example code ; yearsa ; typestar, to (...); ; back ; topettar, topettar,

Consider the code snippet of a simple VHDL process on the left side of the slide.

#### Example: low, high, left, right, ascending Example code HWMod **WS24** 1 process 2 type int\_t is [...]; 3 begin 4 report "low/high: " & Example I to\_string(int\_t'low) & "/" & 5 6 to\_string(int\_t'high); 7 report "left/right: " & 8 to\_string(int\_t'left) & "/" & to\_string(int\_t'right); 9 10 report "asc: " & to\_string(int\_t'ascending); wait; 11 12 end process;

In its declarative part an integer type called  $int_t$  is declared.

#### Example: low, high, left, right, ascending Example code HWMod **WS24** 1 process type int\_t is [...]; 2 3 begin 4 report "low/high: " & 5 to\_string(int\_t'low) & "/" & 6 to\_string(int\_t'high); 7 report "left/right: " & 8 to\_string(int\_t'left) & "/" & to\_string(int\_t'right); 9 10 report "asc: " & to\_string(int\_t'ascending); wait; 11 12 end process;

Note that for the purpose of this demonstration we've substituted the type's range constraint with a placeholder, as we intend to execute this process with two different range configurations.

#### Example: low, high, left, right, ascending Example code HWMod **WS24** 1 process type int\_t is [...]; 2 3 begin 4 report "low/high: " & Example I 5 to\_string(int\_t'low) & "/" & 6 to\_string(int\_t'high); 7 report "left/right: " & 8 to\_string(int\_t'left) & "/" & to\_string(int\_t'right); 9 report "asc: " & to\_string(int\_t'ascending); 10 wait; 11 12 end process;

Example code

 protect
 protect

The function of the process is to simply print the values of the predefined attributes low, high, left, right and ascending of the defined integer type.

#### Example: low, high, left, right, ascending Example code HWMod **WS24** 1 process 2 type int\_t is [...]; 3 begin 4 report "low/high: " & to\_string(int\_t'low) & "/" & 5 6 to\_string(int\_t'high); 7 report "left/right: " & 8 to\_string(int\_t'left) & "/" & to\_string(int\_t'right); 9 report "asc: " & to\_string(int\_t'ascending); 10 wait; 11 12 end process;



On the right side we can now see the output of the simulator for the two range scenarios. In the first case a descending range constraint is used, while the second case uses an ascending one with the same limits. For both cases the attributes low and high return the lowest and highest values in the range constraint. However, since left and right refer to the left and right values of the range constraint as they are used in the type declaration, the respective attributes return opposite values for the two cases.

#### Example: low, high, left, right, ascending Example code Output HWMod **WS24** range 3 downto -4 1 process 2 type int\_t is [...]; [...]: low/high: -4/3 [...]: left/right: 3/-4 3 begin 4 report "low/high: " & [...]: asc: false Example I to\_string(int\_t'low) & "/" & 5 ■ range -4 to 3 6 to\_string(int\_t'high); [...]: low/high: -4/3 7 report "left/right: " & [...]: left/right: -4/3 8 to\_string(int\_t'left) & "/" & [...]: asc: true to\_string(int\_t'right); 9 report "asc: " & to\_string(int\_t'ascending); 10 11 wait; 12 end process;

10



In general we can deduce that for ascending ranges the low attribute is always equal to the left limit, while the high and the right attribute are also equal. For descending ranges the situation is reversed. Note, that these attributes are not only defined for integer types, but can also be used on floating-point, enumeration and physical types. The type of the attributes low, high, left and right are always equal to the type that the attributes are invoked on.

#### Example: low, high, left, right, ascending Example code Output HWMod **WS24** range 3 downto -4 1 process 2 type int\_t is [...]; [...]: low/high: -4/3 [...]: left/right: 3/-4 3 begin report "low/high: " & 4 [...]: asc: false Example I to\_string(int\_t'low) & "/" & 5 range -4 to 3 6 to\_string(int\_t'high); [...]: low/high: -4/3 7 report "left/right: " & [...]: left/right: -4/3 8 to\_string(int\_t'left) & "/" & [...]: asc: true to\_string(int\_t'right); 9

```
Note
```

11 wait; 12 end process;

10 11

For ascending types: T'low = T'left and T'high = T'right
For descending types: T'low = T'right and T'high = T'left

report "asc: " & to\_string(int\_t'ascending);

On this slide we want to present the attribute functions image and value, which can be used to convert between scalar types and strings.

#### Example: image, value

Example code

#### HWMod WS24

Types Scalar Types Attributes Example I Example II Subtypes

```
1 process
2 variable str : string(1 to 2) := "42";
3 variable val : integer;
4 begin
5 report str;
6 val := integer'value(str); -- convert string to integer
7 report integer'image(val); -- convert integer to string and print it
8 wait;
9 end process;
```

#### Note

The code snippet shows a process that declares two variables. The variable called str is a string and is initialized with the decimal string representation of the number 42. Strings in VHDL are actually array types, which will be covered in a future lecture. The second variable is a simple integer.

#### Example: image, value HWMod **WS24** Example code 1 process variable str : string(1 to 2) := "42"; 2 3 variable val : integer; Example II 4 begin 5 report str; 6 val := integer'value(str); -- convert string to integer 7 report integer 'image (val); -- convert integer to string and print it wait: 8 9 end process;

#### Note



The process first prints the string 42 using a report statement.

#### Example: image, value HWMod **WS24** Example code 1 process 2 variable str : string(1 to 2) := "42"; 3 variable val : integer; Example II 4 begin 5 report str; 6 val := integer'value(str); -- convert string to integer 7 report integer 'image (val); -- convert integer to string and print it wait; 8 9 end process;

#### Note



It then uses the predefined value function-attribute to convert the string to an integer.

#### Example: image, value HWMod **WS24** Example code 1 process 2 variable str : string(1 to 2) := "42"; 3 variable val : integer; Example II 4 begin 5 report str; val := integer'value(str); -- convert string to integer 6 7 report integer 'image (val); -- convert integer to string and print it wait; 8

```
9 end process;
```

#### Note



Using the image attribute, it then converts the integer back into a string and again prints it out – resulting in the string "42" being output once more.

#### Example: image, value

Example code

#### HWMod WS24

```
Types
Scalar Types
Attributes
Example I
Example II
Subtypes
```

```
1 process
2 variable str : string(1 to 2) := "42";
3 variable val : integer;
4 begin
5 report str;
6 val := integer'value(str); -- convert string to integer
7 report integer'image(val); -- convert integer to string and print it
8 wait;
9 end process;
```

#### Note

Please note that instead of using the image attribute, it is often easier and more readable to simply invoke the to-string function, which is also used in the example on the previous slide. This function is implicitly declared for all scalar types.

#### Example: image, value

Example code

#### HWMod WS24

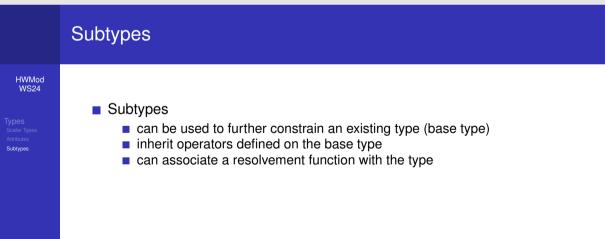
Types Scalar Types Attributes Example I Example II Subtypes

```
1 process
2 variable str : string(1 to 2) := "42";
3 variable val : integer;
4 begin
5 report str;
6 val := integer'value(str); -- convert string to integer
7 report integer'image(val); -- convert integer to string and print it
8 wait;
9 end process;
```

#### Note

#### └─VHDL Type System └─Subtypes └─**Subtypes**

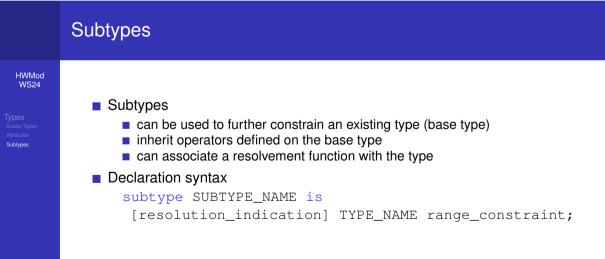
Another VHDL concept we want to touch upon, before we end this lecture are subtypes. Subtypes can be used to further constrain the value range of an already defined type – which is then referred to as the base type. Additionally, it is possible to also attach a so-called resolvement function to the subtype. How this feature can be used, is subject to an upcoming lecture.



#### └─VHDL Type System └─Subtypes └─**Subtypes**

Subtypes
a can be used to kirther constrain an existing type (base type)
in their operators added on the base type
can associate a resolvement function with the type
Declaration syntax
subtype SUBTYPE\_UNAX: is
[resolution\_indication] TYPE\_UNAXE range\_constraint,

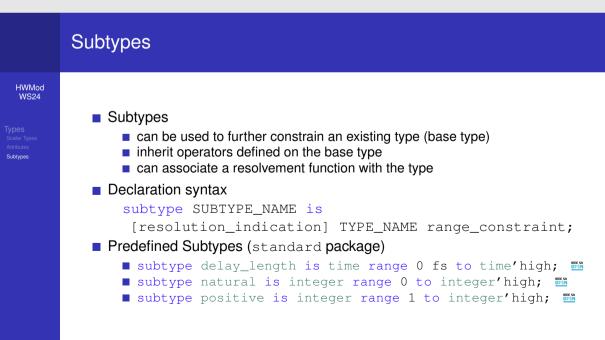
For the declaration of subtypes the subtype keyword is used. Besides the actual name of the subtype, the name of the base type as well as a new – possibly more restrictive – range constraint must be specified.



└─VHDL Type System └─Subtypes └─**Subtypes** 

Bubyon
 Bathyon
 <

The standard package already contains some predefined subtype declarations. Let's consider the natural type. It uses integer as its base type and restricts the range to all values greater or equal to zero. In these examples we can also see the VHDL high attribute in action, which is used to refer to the largest possible value of the respective base type.



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.



Types Scalar Types Attributes Subtypes

### Lecture Complete!

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