Hardware Modeling [VU] (191.011)
– WS25 –
VHDL Type System

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2025/26

Modified: 2025-12-16, 16:03 (f8a58e9)

Introduction

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

Introduction

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

```
type TYPE_NAME is [...];
```

Introduction

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

  ```
  type TYPE_NAME is [...];
  ```
- Possible type declaration locations
  - Packages
  - Declarative parts of entities, architectures, processes, blocks, functions, procedures

Introduction (cont'd)

- Built-in types
  - Defined in `standard` package
  - Some Examples
    - `boolean`
    - `integer`
    - `real`
    - `time`

Introduction (cont'd)

- Built-in types
  - Defined in `standard` package
  - Some Examples
    - `boolean`
    - `integer`
    - `real`
    - `time`
- Five classes of types
  - scalar
  - composite
  - file
  - access
  - protected

Introduction (cont'd)

- Built-in types
  - Defined in `standard` package
  - Some Examples
    - `boolean`
    - `integer`
    - `real`
    - `time`
- Five classes of types
  - scalar
  - composite
  - file
  - access
  - protected
- Keep synthesizeability in mind!

Integer Types

Types
Scalar Types
Integers
Floats
Enums
Physical Types
Attributes
Subtypes

Integer Types

- Declaration Syntax

```
type TYPE_NAME is range_constraint;
range_constraint ::= range expr to expr
                   | range expr downto expr
                   | range range_attribute_name
```

Integer Types

- 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 $\leq$ right
  - Descending (`downto`) left $\geq$ right

Integer Types (cont'd)

- Examples
  - 8 bit unsigned integer

```
type uint8_t is range 0 to 255;
```

  - 16 bit signed integer

```
type int16_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 42 downto 13;
```

Integer Types (cont'd)

- Examples
  - 8 bit unsigned integer

```
type uint8_t is range 0 to 255;
```

  - 16 bit signed integer

```
type int16_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 42 downto 13;
```

- Only built-in integer type: `integer` IEEE SA OPEN
  - Ascending range
  - Range boundaries are implementation dependent
    - VHDL-2008: at least $-2^{31}-1$ to $2^{31}-1$ (i.e., 32 bit)
    - VHDL-2019: at least -2^{63} to $2^{63}-1$ (i.e., 64 bit)

Range Contraints

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

Range Contraints

- 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
2    variable b : uint8_t := 16*16; --  range error: 256 > 255
```

Range Contraints

- 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
2    variable b : uint8_t := 16*16; --  range error: 256 > 255
```

- Example: dynamic range error

```
1    process
2      variable x : uint8_t := 254;
3    begin
4      x := x + 1;
5      report "everything fine!";
6      x := x + 1;
7      report "not fine" -- never executed because of the range error
8    end process;
```

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

- Declaration Syntax

```
type TYPE_NAME is range_constraint;
```

- Example

```
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

Floating-Point Types

- Declaration Syntax

```
type TYPE_NAME is range_constraint;
```

- Example

```
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

Floating-Point Types

- Declaration Syntax

```
type TYPE_NAME is range_constraint;
```

- Example

```
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!

Enumeration Types

Types
Scalar Types
Integers
Floats
Enums
Physical Types
Attributes
Subtypes

Enumeration Types

- Declaration Syntax

```
type TYPE_NAME is (enum_literal {, enum_literal});
```

Enumeration Types

- Declaration Syntax

```
type TYPE_NAME is (enum_literal {, enum_literal});
```

- Examples
  - `type fsm_state_t is (INIT, WAIT_FOR_DATA, TRANSMIT);`
  - `type color_t is (RED, GREEN, BLUE);`
  - `type my_char_t is ('A', '1', '*');`

Enumeration Types

- Declaration Syntax

```
type TYPE_NAME is (enum_literal {, enum_literal});
```

- Examples
  - `type fsm_state_t is (INIT, WAIT_FOR_DATA, TRANSMIT);`
  - `type color_t is (RED, GREEN, BLUE);`
  - `type my_char_t is ('A', '1', '*');`
- Predefined enumeration types (`standard` package)
  - `type boolean is (false, true);`
  - `type direction is (ASCENDING, DESCENDING);`
  - `type bit is ('0', '1');`
  - `type character is (NUL, SOH, [...], 'A', 'B', [...]);`

Physical Types

Types
Scalar Types
Integers
Floats
Enums
Physical Types
Attributes
Subtypes

Physical Types

- Declaration Syntax

```
type TYPE_NAME is range_constraint units
  primary_unit
  {secondary_unit}
end units;
```

Physical Types

- Declaration Syntax

```
type TYPE_NAME is range_constraint units
  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;
```

Physical Types

- Declaration Syntax

```
type TYPE_NAME is range_constraint units
  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;
```

- Predefined physical type (`standard` Package): `time`

Type Attributes

Types
Scalar Types
Attributes
Example I
Example II
Subtypes

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

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
- 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)`

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
- 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: low, high, left, right, ascending

Example code

```
1   process
2     type int_t is [...];
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) & "/" &
9       to_string(int_t'right);
10    report "asc: " & to_string(int_t'ascending);
11    wait;
12  end process;
```

Example: low, high, left, right, ascending

Example code

```
1   process
2     type int_t is [...];
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) & "/" &
9       to_string(int_t'right);
10    report "asc: " & to_string(int_t'ascending);
11    wait;
12  end process;
```

Example: low, high, left, right, ascending

Example code

```
1   process
2     type int_t is [...];
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) & "/" &
9       to_string(int_t'right);
10    report "asc: " & to_string(int_t'ascending);
11    wait;
12  end process;
```

Example: low, high, left, right, ascending

Example code

```
1   process
2     type int_t is [...];
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) & "/" &
9       to_string(int_t'right);
10    report "asc: " & to_string(int_t'ascending);
11    wait;
12  end process;
```

Example: low, high, left, right, ascending

Example code

```
1  process
2    type int_t is [...];
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) & "/" &
9      to_string(int_t'right);
10   report "asc: " & to_string(int_t'ascending);
11   wait;
12 end process;
```

Output

- `range 3 downto -4`

```
[...]: low/high: -4/3
[...]: left/right: 3/-4
[...]: asc: false
```

- `range -4 to 3`

```
[...]: low/high: -4/3
[...]: left/right: -4/3
[...]: asc: true
```

Example: low, high, left, right, ascending

Example code

```
1  process
2    type int_t is [...];
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) & "/" &
9      to_string(int_t'right);
10   report "asc: " & to_string(int_t'ascending);
11   wait;
12 end process;
```

Output

- `range 3 downto -4`

```
[...]: low/high: -4/3
[...]: left/right: 3/-4
[...]: asc: false
```

- `range -4 to 3`

```
[...]: low/high: -4/3
[...]: left/right: -4/3
[...]: asc: true
```

Note

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

Example: image, value

Example code

```
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

For all scalar types `T` and values of that type `V`: `V = T'value(T'image(V))`

Example: image, value

Example code

```
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

For all scalar types `T` and values of that type `V`: `V = T'value(T'image(V))`

Example: image, value

Example code

```
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

For all scalar types `T` and values of that type `V`: `V = T'value(T'image(V))`

Example: image, value

Example code

```
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

For all scalar types `T` and values of that type `V`: `V = T'value(T'image(V))`

Example: image, value

Example code

```
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

For all scalar types `T` and values of that type `V`: `V = T'value(T'image(V))`

Example: image, value

Example code

```
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

For all scalar types `T` and values of that type `V`: `V = T'value(T'image(V))`

Subtypes

- Subtypes
  - can be used to further constrain an existing type (base type)
  - inherit operators defined on the base type
  - can associate a resolvement function with the type

Subtypes

- Subtypes
  - can be used to further constrain an existing type (base type)
  - inherit operators defined on the base type
  - can associate a resolvement function with the type
- Declaration syntax

```
subtype SUBTYPE_NAME is
 [resolution_indication] TYPE_NAME range_constraint;
```

Subtypes

- Subtypes
  - can be used to further constrain an existing type (base type)
  - inherit operators defined on the base type
  - can associate a resolvement function with the type
- Declaration syntax

```
subtype SUBTYPE_NAME is
  [resolution_indication] TYPE_NAME range_constraint;
```

- Predefined Subtypes (`standard` package)
  - `subtype delay_length is time range 0 fs to time'high;`
  - `subtype natural is integer range 0 to integer'high;`
  - `subtype positive is integer range 1 to integer'high;`

Lecture Complete!