Hardware Modeling [VU] (191.011)
– WS24 –
Subprograms

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

In order to facilitate reusability, maintainability, modularity and increased readability, VHDL comes with subprograms in different flavors. While we have already encountered them in previous lectures out of necessity, we will cover them in detail in this dedicated lecture. Prominent examples we already saw are conversion functions like `to_string`, resolution functions, as well as logic and arithmetic operators.

Hardware Modeling [VU] (191.011)
– WS24 –
Subprograms

Florian Huemer & Sebastian Wiedemann & Dylan Baumann

WS 2024/25

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

VHDL features two forms of subprograms, referred to as `function` and `procedure`. Both encapsulate sequential pieces of code and take parameters which must be provided when calling the respective subprogram. You can compare them to functions in C or methods in Java. We will now give a short overview about the differences between them before discussing their respective properties in more detail.

Overview

- Functions
- Procedures

- Functions
  - Call is **expression** returning a value
- Procedures

In VHDL, functions always return a value of a pre-defined base type, resulting in function calls being *expressions*. Therefore, function calls must always be part of some statement, meaning the returned value must **always** be used. There is no option to drop it like in C, or Java. The VHDL standard further differentiates between functions having no side effects and being deterministic, and functions for which this is not the case.

Overview

- Functions
  - Call is **expression** returning a value
- Procedures

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
- Procedures

So-called `pure` functions only use their parameters to compute a return value and are therefore deterministic and free of side effects.

Overview

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
- Procedures

Functions
Call is **expression** returning a value
pure: No side effects, same parameters ⇒ same return value
impure: Side effects possible, return value can vary for identical calls
Procedures

In addition to `pure` functions, there are the so-called `impure` functions, which can have side effects and are allowed to be nondeterministic, meaning that distinct calls with the same parameters may return different results. We will consider examples for both kinds of functions later.

Overview

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
  - `impure`: Side effects possible, return value can vary for identical calls
- Procedures

Functions
Call is **expression** returning a value
pure: No side effects, same parameters ⇒ same return value
impure: Side effects possible, return value can vary for identical calls
Procedures

The other class of subprograms is the so-called `procedure`. Subprograms of this kind do not return a value, therefore only operating via side effects.

Overview

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
  - `impure`: Side effects possible, return value can vary for identical calls
- Procedures

Functions
- Call is **expression** returning a value
- pure: No side effects, same parameters ⇒ same return value
- impure: Side effects possible, return value can vary for identical calls

Procedures
- Call is **statement** with side effects and no return value

Furthermore, without a value being returned, procedure calls are also not expressions but rather statements. This means they are used on their own. A further difference to functions is that the subset of VHDL statements they are allowed to contain is not as restrictive as the one of functions. We will explain this in more detail later.

Overview

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
  - `impure`: Side effects possible, return value can vary for identical calls
- Procedures
  - Call is **statement** with side effects and no return value

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
  - `impure`: Side effects possible, return value can vary for identical calls
- Procedures
  - Call is **statement** with side effects and no return value
- Subprogram call

Before we continue, we want to show the syntax of subprogram calls via two examples.

Overview

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
  - `impure`: Side effects possible, return value can vary for identical calls
- Procedures
  - Call is **statement** with side effects and no return value
- Subprogram call

- Functions
  - Call is **expression** returning a value
  - pure: No side effects, same parameters ⇒ same return value
  - impure: Side effects possible, return value can vary for identical calls
- Procedures
  - Call is **statement** with side effects and no return value
- Subprogram call
  - Parameters passed in parentheses
    report to_string(x); -- function call with one parameter

In the first example, we call the already encountered `to_string` function, which has a single parameter and returns a string. Note how the syntax of the function call is just like the one you know from C and Java. That is, parameters are associated to the respective arguments in parentheses. However, it is also possible to map parameters via named association, similar to entity instantiations.

Overview

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
  - `impure`: Side effects possible, return value can vary for identical calls
- Procedures
  - Call is **statement** with side effects and no return value
- Subprogram call
  - Parameters passed in parentheses

```
report to_string(x); -- function call with one parameter
```

Functions
Call is **expression** returning a value
pure: No side effects, same parameters ⇒ same return value
impure: Side effects possible, return value can vary for identical calls
Procedures
Call is **statement** with side effects and no return value
Subprogram call
Parameters passed in parentheses
report to_string(x); -- function call with one parameter
In case of zero parameters no parentheses
stop; -- procedure call without parameters

The other example shows a call to a procedure called `stop` that has no parameters. Observe how, different to C or Java, that does not pass arguments does not feature parentheses.

Overview

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
  - `impure`: Side effects possible, return value can vary for identical calls
- Procedures
  - Call is **statement** with side effects and no return value
- Subprogram call
  - Parameters passed in parentheses

    ```
    report to_string(x); -- function call with one parameter
    ```
  - In case of zero parameters no parentheses

    ```
    stop; -- procedure call without parameters
    ```

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
  - `impure`: Side effects possible, return value can vary for identical calls
- Procedures
  - Call is **statement** with side effects and no return value
- Subprogram call
  - Parameters passed in parentheses
    `report to_string(x); -- function call with one parameter`
  - In case of zero parameters no parentheses
    `stop; -- procedure call without parameters`

33

We will now continue by discussing functions in more detail.

Overview

- Functions
  - Call is **expression** returning a value
  - `pure`: No side effects, same parameters ⇒ same return value
  - `impure`: Side effects possible, return value can vary for identical calls
- Procedures
  - Call is **statement** with side effects and no return value
- Subprogram call
  - Parameters passed in parentheses
    `report to_string(x); -- function call with one parameter`
  - In case of zero parameters no parentheses
    `stop; -- procedure call without parameters`

■ Simplified declaration syntax

First, let us consider the syntax for declaring functions. Be aware though, that we will only consider a simple subset of possible VHDL functions in this course and that we will restrict the declaration syntax accordingly. In case you are curious though, be invited to have a look at the respective section of the VHDL standard.

Functions

- Simplified declaration syntax

Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```


By now, you should already be quite familiar with the general structure of the function declaration, as the declarations of a `process`, `entity` and `architecture` are quite similar. Nevertheless, we will now quickly go through it.

Functions

33

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

Naturally, a `function` declaration contains a function `designator`, just like as for an entity, an optional list of parameter as well as a return type.

Functions

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

Simplified declaration syntax
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
[declarative_part]
begin
[statement part] -- function body
return ...;
end function;

The `designator` can either be an identifier or an operator symbol, like a + or −. We will consider an example for both soon.

Functions

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

Simplified declaration syntax
Type of the returned value can be scalar or composite

The purpose of the return type between the parameter list and the declarative part is to define the base type of the returned values. This can be any scalar or composite type. It is even possible to specify an unconstrained composite type.

Functions

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

- Type of the returned value can be scalar or composite

Simplified declaration syntax
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
[declarative_part]
begin
[statement part] -- function body
return ...;
end function;
Type of the returned value can be scalar or composite
Default is pure

Of course we also need a means to declare whether a function is supposed to be `pure` or `impure` depending on which kind it is. However, since all functions are `pure` per default this element of the declaration is optional until an `impure` function is desired.

Functions

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

- Type of the returned value can be scalar or composite
- Default is `pure`

33
- Simplified declaration syntax
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part] -- function body
  return ...;
end function;
- Type of the returned value can be scalar or composite
- Default is pure
- Primarily for computing values

Functions in VHDL are primarily supposed to be used for computing values.

Functions

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

- Type of the returned value can be scalar or composite
- Default is `pure`
- Primarily for computing values

- Simplified declaration syntax
```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```
- Type of the returned value can be scalar or composite
- Default is `pure`
- Primarily for computing values
  - ⇒ Does not advance simulation time

This is enforced by disallowing functions to advance the simulation time.

Functions

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

- Type of the returned value can be scalar or composite
- Default is `pure`
- Primarily for computing values
  - ⇒ Does not advance simulation time

Simplified declaration syntax
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
[declarative_part]
begin
[statement part] -- function body
return ...;
end function;
Type of the returned value can be scalar or composite
Default is pure
Primarily for computing values
⇒ Does not advance simulation time
Must not contain wait statements

This is achieved by prohibiting functions to contain `wait` in their statement part, which can, other than that, in general consist of a sequence of almost arbitrary sequential statements.

Functions

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

- Type of the returned value can be scalar or composite
- Default is `pure`
- Primarily for computing values
  - ⇒ Does not advance simulation time
- Must not contain wait statements 179

Simplified declaration syntax
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
[declarative_part]
begin
[statement part] -- function body
return ...;
end function;
Type of the returned value can be scalar or composite
Default is pure
Primarily for computing values
⇒ Does not advance simulation time
Must not contain wait statements
Must also hold for subprograms called inside the body

Consequently, functions must also not call any other subprogram that advances the simulation time within their statement part. While this is satisfied for all function calls per definition, we will later see that for procedures this is not always the case.

Functions

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

- Type of the returned value can be scalar or composite
- Default is `pure`
- Primarily for computing values
  - ⇒ Does not advance simulation time
- Must not contain wait statements 179
  - Must also hold for subprograms called inside the body

Simplified declaration syntax
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
[declarative_part]
begin
[statement part] -- function body
return ...;
end function;
Type of the returned value can be scalar or composite
Default is pure
Primarily for computing values
⇒ Does not advance simulation time
Must not contain wait statements
Must also hold for subprograms called inside the body
Must always end in return

Finally, each path through a function's body **must** end in a `return` statement. This can either be a single one at the end of a function, or multiple ones in case of distinct termination conditions. Naturally, the type of the value returned by these statements must fit the declared return type.

Functions

- Simplified declaration syntax

```
[pure|impure] function designator [(parameter_list)] return TYPE_NAME is
  [declarative_part]
begin
  [statement part]  -- function body
  return ...;
end function;
```

- Type of the returned value can be scalar or composite
- Default is `pure`
- Primarily for computing values
  - ⇒ Does not advance simulation time
- Must not contain wait statements 179
  - Must also hold for subprograms called inside the body
- **Must** always end in `return`

- Declaration syntax

We will now discuss the optional parameter list of functions.

Function Parameters

35

HWMod
WS24

- Declaration syntax

Declaration syntax
parameter_list ::= parameters{; parameters}

A parameter list consists of lists, in the syntax referred to as `parameters`, of parameter identifiers that share class and type. These lists are separated via semicolon.

Function Parameters

- Declaration syntax

```
parameter_list ::= parameters{; parameters}
```

Declaration syntax
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type

Parameters can either be of the class `constant`, `signal` or `file`, with `constant` being the default. While we already discussed the `constant` and `signal` classes by now, the `file` class was not yet thoroughly introduced. This will be the content of an upcoming lecture. However, for now we simply need to know that it allows us to interact with a file of the local filesystem. The class in the declaration of parameters is responsible for determining how the respective parameters are passed to a function and what of it is accessible within the function's statement part.

Function Parameters

- Declaration syntax

```
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
```

The slide shows an example function declaration with a parameter list. This list consists of two lists in turn, one featuring two `integer` parameters, and one featuring a single `natural` parameter.

Function Parameters

- Declaration syntax

```
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
```

- Examples

```
function f1(a,b : integer; signal c : natural) return bit
```

The default type is `constant`. In this case, parameters are simply passed to a function by copying their value. An important consequence of this is that if you pass a `signal` to a `constant` parameter - which is possible - just its value will be copied but **not** its attributes.

Function Parameters

- Declaration syntax

```
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
```

- Default parameter class is `constant`

- Examples

```
function f1(a,b : integer; signal c : natural) return bit
```

Therefore, if you need to access them, you need to declare the parameter to be of the signal class. In this case the attributes will be copied as well. For example, this is done for the parameter `c` in the example shown on the slide.

Function Parameters

- Declaration syntax

```
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
```

- Default parameter class is `constant`
  - Pass-by-copy
  - If signal attributes are required use `signal`

- Examples

```
function f1(a,b : integer; signal c : natural) return bit
```

■ Declaration syntax
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
■ Default parameter class is constant
■ Pass-by-copy
■ If signal attributes are required use signal
■ Not modifiable by function (in general)
■ Examples
function f1(a,b : integer; signal c : natural) return bit

Also note that this pass-by-copy behavior means that parameters are in general not modifiable by functions.

Function Parameters

- Declaration syntax

```
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
```

- Default parameter class is `constant`
  - Pass-by-copy
  - If signal attributes are required use `signal`
  - Not modifiable by function (in general)

- Examples

```
function f1(a,b : integer; signal c : natural) return bit
```

- Declaration syntax
  parameter_list ::= parameters{; parameters}
  parameters ::= [constant|signal|file] identifier_list: type
- Default parameter class is constant
  - Pass-by-copy
  - If signal attributes are required use signal
  - Not modifiable by function (in general)
- Type: scalar or composite (possibly unconstrained)
- Examples
  function f1(a,b : integer; signal c : natural) return bit

The type of a parameter can be an arbitrary scalar or composite types, both constrained and unconstrained.

Function Parameters

- Declaration syntax

```
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
```

- Default parameter class is `constant`
  - Pass-by-copy
  - If signal attributes are required use `signal`
  - Not modifiable by function (in general)
- Type: scalar or composite (possibly unconstrained)

- Examples

```
function f1(a,b : integer; signal c : natural) return bit
```

35
Declaration syntax
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
Default parameter class is constant
Pass-by-copy
If signal attributes are required use signal
Not modifiable by function (in general)
Type: scalar or composite (possibly unconstrained)
Default value possible
Examples
function f1(a,b : integer; signal c : natural) return bit
function f2(a : integer := 42) return bit

We also want to point out that it is possible to give parameters a default value as shown by the second example on the slide. As in other programming languages, this value will be used when no respective parameter is passed during the function call.

Function Parameters

- Declaration syntax

```
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
```

- Default parameter class is `constant`
  - Pass-by-copy
  - If signal attributes are required use `signal`
  - Not modifiable by function (in general)
- Type: scalar or composite (possibly unconstrained)
- Default value possible
- Examples

```
function f1(a,b : integer; signal c : natural) return bit
function f2(a : integer := 42) return bit
```

35
- Declaration syntax
  parameter_list ::= parameters{; parameters}
  parameters ::= [constant|signal|file] identifier_list: type
- Default parameter class is constant
  - Pass-by-copy
  - If signal attributes are required use signal
  - Not modifiable by function (in general)
- Type: scalar or composite (possibly unconstrained)
- Default value possible
- Examples
  function f1(a,b : integer; signal c : natural) return bit
  function f2(a : integer := 42) return bit
  function f3 return string

Finally, for the sake of completeness, the third example shows the declaration of a function without any parameters at all. Note that, similar to calling such a function, no parentheses are used.

Function Parameters

35

HWMod
WS24

Subprograms
Functions
Overview
Pure
Impure
Recommendations
Procedures
Overloading
Packages

- Declaration syntax

```
parameter_list ::= parameters{; parameters}
parameters ::= [constant|signal|file] identifier_list: type
```

- Default parameter class is `constant`
  - Pass-by-copy
  - If signal attributes are required use `signal`
  - Not modifiable by function (in general)
- Type: scalar or composite (possibly unconstrained)
- Default value possible
- Examples

```
function f1(a,b : integer; signal c : natural) return bit
function f2(a : integer := 42) return bit
function f3 return string
```

As already mentioned before, there are two kinds of functions in VHDL, referred to as `pure` and `impure`. We will now discuss pure functions.

Pure Functions

Always return same value when passed same parameters (determinism)

The idea behind pure functions is to make it explicit that a function is deterministic and that it does not have side effects. By determinism, we mean that a function will always return the same value when passed the same parameters.

Pure Functions

- **Always** return same value when passed same parameters (determinism)

■ **Always** return same value when passed same parameters (determinism)
⇒ Can only access its parameters and **constants** from outer scope

To achieve this, pure functions are restricted to **only** use their parameters, as well as constants from the outer scope. A function can thus, for example, use package constants to compute its return value.

Pure Functions

- **Always** return same value when passed same parameters (determinism)
  - ⇒ Can only access its parameters and **constants** from outer scope

■ **Always** return same value when passed same parameters (determinism)
⇒ Can only access its parameters and **constants** from outer scope
■ Only computes value, no side effects

A consequence of a pure function not having access to anything from its outer scope other than constants is that it can also note have any side-effects. While the idea of a pure function might sound odd at this point, this is actually something we already encountered in previous lectures.

Pure Functions

- **Always** return same value when passed same parameters (determinism)
  - ⇒ Can only access its parameters and **constants** from outer scope
- Only computes value, no side effects

- **Always** return same value when passed same parameters (determinism)
  - ⇒ Can only access its parameters and **constants** from outer scope
- Only computes value, no side effects
- Examples
  - Resolution functions (e.g., IEEE-1164's `resolved`)

One example are resolution functions like the one for `std_logic` in the `std_logic_1164` standard. Naturally we want this resolution to be deterministic, as drivers applying the same values at different points in time should always result in the same resolved value.

Pure Functions

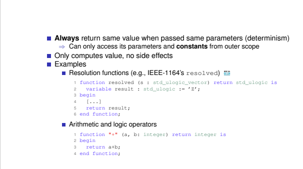
- **Always** return same value when passed same parameters (determinism)
  - ⇒ Can only access its parameters and **constants** from outer scope
- Only computes value, no side effects
- Examples
  - Resolution functions (e.g., IEEE-1164's `resolved`)

The slide shows the declaration of this function. Since there is no explicit `impure` prefix, this function is a pure function. Note how the single parameter is an unconstrained type.

Pure Functions

- **Always** return same value when passed same parameters (determinism)
  - ⇒ Can only access its parameters and **constants** from outer scope
- Only computes value, no side effects
- Examples
  - Resolution functions (e.g., IEEE-1164's `resolved`)

```
function resolved (s : std_ulogic_vector) return std_ulogic is
  variable result : std_ulogic := 'Z';
begin
  [...]
  return result;
end function;
```



Other examples are arithmetic and logic operators, like the integer addition shown on the slide.

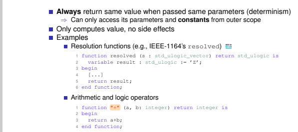
Pure Functions

- **Always** return same value when passed same parameters (determinism)
  ⇒ Can only access its parameters and **constants** from outer scope
- Only computes value, no side effects
- Examples
  - Resolution functions (e.g., IEEE-1164's `resolved`)

```
function resolved (s : std_ulogic_vector) return std_ulogic is
  variable result : std_ulogic := 'Z';
begin
  [...]
  return result;
end function;
```

  - Arithmetic and logic operators

```
function "+" (a, b: integer) return integer is
begin
  return a+b;
end function;
```



Note how the function designator is now an operator symbol instead of an identifier, showing that VHDL features operator overloading like, for example, C++ and many other languages do.

Pure Functions

- **Always** return same value when passed same parameters (determinism)
  - ⇒ Can only access its parameters and **constants** from outer scope
- Only computes value, no side effects
- Examples
  - Resolution functions (e.g., IEEE-1164's `resolved`) IEEE SA OPEN

```
function resolved (s : std_ulogic_vector) return std_ulogic is
  variable result : std_ulogic := 'Z';
begin
  [...]
  return result;
end function;
```

  - Arithmetic and logic operators

```
function "+" (a, b: integer) return integer is
begin
  return a+b;
end function;
```

An impure function is declared just like a pure function with the exception that it starts with the `impure` keyword.

Impure Functions

Might return different value for same parameters (nondeterminism)

In contrast to pure functions, it might return different values for the same parameter values. This essentially means that impure functions are allowed to be non-deterministic.

Impure Functions

- Might return different value for same parameters (nondeterminism)

Might return different value for same parameters (nondeterminism)
Examples
1 impure function now return delay_length;

An example where this nondeterminism is required is the `now` function that always returns the current simulation time when called. The declaration of this function is shown on the slide. Naturally, the return value must be allowed to differ between calls.

Impure Functions

- Might return different value for same parameters (nondeterminism)

- Examples

```
impure function now return delay_length;
```

IEEE SA OPEN

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Examples
impure function now return delay_length;

To achieve this non-determinism, impure functions are not only allowed to access constants of the outer scope, but also signals and variables.

Impure Functions

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope

- Examples

```
impure function now return delay_length;
```

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

```
impure function now return delay_length;
```


Furthermore, in addition to this nondeterminism, impure functions are also allowed to have side effects. A consequence is that a call of an impure function can modify some variable or signal from the outer scope. Note that this allows functions to be stateful.

Impure Functions

HWMod
WS24

Subprograms
Functions
Overview
Pure
Impure
Recommendations
Procedures
Overloading
Packages

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

```
impure function now return delay_length;
```

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples
impure function now return delay_length;

Let us now come to a bigger example, illustrating the need for nondeterminism and side effects. In particular, we will look at a function for generating pseudo-random numbers.

Impure Functions

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

```
impure function now return delay_length;
```

The respective code is shown on the slide. We will now go through it step-by-step.

Impure Functions

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

```
impure function now return delay_length;
```

```
process is
  variable ran : std_ulogic_vector(7 downto 0) := 8d"42";
  impure function prng return integer is
  begin
    ran := ran(6 downto 0) & (ran(7) xor ran(6) xor ran(2));
    return to_integer(unsigned(ran));
  end function;
[...]
```

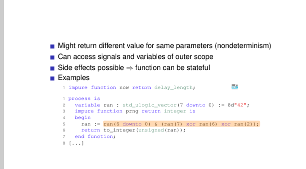
Let us ignore the surrounding code for now and just focus on the function declaration. The function is supposed to return a pseudo-random 8-bit number on each call. Since it does neither require nor take parameters and is supposed to return different results, it must anturally be declared as impure function.

Impure Functions

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

```
impure function now return delay_length;
```

```
process is
  variable ran : std_ulogic_vector(7 downto 0) := 8d"42";
  impure function prng return integer is
  begin
    ran := ran(6 downto 0) & (ran(7) xor ran(6) xor ran(2));
    return to_integer(unsigned(ran));
  end function;
[...]
```



Internally, the current value is always computed out of the previous one, starting with some initial value. To store the previous value, this function clearly requires side effects.

Impure Functions

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

```
impure function now return delay_length;
```

```
process is
  variable ran : std_ulogic_vector(7 downto 0) := 8d"42";
  impure function prng return integer is
  begin
    ran := ran(6 downto 0) & (ran(7) xor ran(6) xor ran(2));
    return to_integer(unsigned(ran));
  end function;
[...]
```

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

In particular, it assigns the result of some computation that is based on the current value of the `ran` variable to this exact variable.

Impure Functions

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

```
impure function now return delay_length;
```

```
process is
  variable ran : std_ulogic_vector(7 downto 0) := 8d"42";
  impure function prng return integer is
  begin
    ran := ran(6 downto 0) & (ran(7) xor ran(6) xor ran(2));
    return to_integer(unsigned(ran));
  end function;
[...]
```

This variable is declared in the outer scope of the function and initialized using a bit-string literal to get the pseudo-random number generation going.

Impure Functions

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

```
impure function now return delay_length;
```

```
process is
  variable ran : std_ulogic_vector(7 downto 0) := 8d"42";
  impure function prng return integer is
  begin
    ran := ran(6 downto 0) & (ran(7) xor ran(6) xor ran(2));
    return to_integer(unsigned(ran));
  end function;
[...]
```

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

We will not go into further detail about how the pseudo-random generation itself works. However, you might recognize this as a linear feedback shift register.

Impure Functions

- Might return different value for same parameters (nondeterminism)
- Can access signals and variables of outer scope
- Side effects possible ⇒ function can be stateful
- Examples

```
impure function now return delay_length;
```

```
process is
  variable ran : std_ulogic_vector(7 downto 0) := 8d"42";
  impure function prng return integer is
  begin
    ran := ran(6 downto 0) & (ran(7) xor ran(6) xor ran(2));
    return to_integer(unsigned(ran));
  end function;
[...]
```

At this point you might ask yourself why you should even use pure functions when impure ones are a more powerful superset of pure functions. This question is justified as there never is a real *requirement* to use a pure function.

Recommendations

■ Use pure functions whenever possible
■ Easier to understand and debug

However, we still recommend you to use them whenever possible. The reasons being that this assists the tools in applying optimizations and makes understanding and debugging your code significantly easier. Just consider yourself debugging code from someone else and think about how comforting it would be when you could rule out whether a function has side effects or not.

Recommendations

- Use `pure` functions whenever possible
  - Easier to understand and debug

Use pure functions whenever possible
Easier to understand and debug
Use impure functions for

Impure functions on the other hand, should only be used when you *really* require their properties.

Recommendations

- Use `pure` functions whenever possible
  - Easier to understand and debug
- Use `impure` functions for

- Use pure functions whenever possible
  - Easier to understand and debug
- Use impure functions for
  - nondeterministic behavior

This is of course the case when your function is nondeterministic.

Recommendations

- Use `pure` functions whenever possible
  - Easier to understand and debug
- Use `impure` functions for
  - nondeterministic behavior

- Use `pure` functions whenever possible
  - Easier to understand and debug
- Use `impure` functions for
  - nondeterministic behavior
  - stateful functions

Another case is when you require your function to store something between calls.

Recommendations

- Use `pure` functions whenever possible
  - Easier to understand and debug
- Use `impure` functions for
  - nondeterministic behavior
  - stateful functions

- Use pure functions whenever possible
  - Easier to understand and debug
- Use impure functions for
  - nondeterministic behavior
  - stateful functions
  - file I/O

Another use case is when you need to interface with the simulation host's file system. Clearly file I/O require side effects.

Recommendations

- Use `pure` functions whenever possible
  - Easier to understand and debug
- Use `impure` functions for
  - nondeterministic behavior
  - stateful functions
  - file I/O

- Use pure functions whenever possible
  - Easier to understand and debug
- Use impure functions for
  - nondeterministic behavior
  - stateful functions
  - file I/O
  - protected types

Finally impure functions are required when implementing protected types. While we have not covered these types yet, they are quite a powerful feature of VHDL as they are somewhat akin to classes in object-oriented programming.

Recommendations

- Use `pure` functions whenever possible
  - Easier to understand and debug
- Use `impure` functions for
  - nondeterministic behavior
  - stateful functions
  - file I/O
  - protected types

Use pure functions whenever possible
Easier to understand and debug
Use impure functions for
nondeterministic behavior
stateful functions
file I/O
protected types
Especially for impure functions with side effects: **use sensible names!**

At this point we also want to stress the need for good function names. This holds especially true if your functions have side effects.

Recommendations

- Use `pure` functions whenever possible
  - Easier to understand and debug
- Use `impure` functions for
  - nondeterministic behavior
  - stateful functions
  - file I/O
  - protected types
- Especially for `impure` functions with side effects: **use sensible names!**

- Use pure functions whenever possible
  - Easier to understand and debug
- Use impure functions for
  - nondeterministic behavior
  - stateful functions
  - file I/O
  - protected types
- Especially for impure functions with side effects: **use sensible names!**
- **Caveat**: File parameters can introduce non-determinism and side effects into pure-functions

Finally, we want to mention a caveat when using pure functions. As you might have noticed in the parameter list syntax before, pure functions are allowed to have file parameters.

Recommendations

- Use `pure` functions whenever possible
  - Easier to understand and debug
- Use `impure` functions for
  - nondeterministic behavior
  - stateful functions
  - file I/O
  - protected types
- Especially for `impure` functions with side effects: **use sensible names!**
- **Caveat**: File parameters can introduce non-determinism and side effects into pure-functions

- Use pure functions whenever possible
  - Easier to understand and debug
- Use impure functions for
  - nondeterministic behavior
  - stateful functions
  - file I/O
  - protected types
- Especially for impure functions with side effects: **use sensible names!**
- **Caveat**: File parameters can introduce non-determinism and side effects into pure-functions
  - ⇒ Recommendation: Do not use file parameters for pure functions

However, we strongly recommend you **not** to do this, as file I/O can always lead to side effects, nondeterminism with respect to the function parameters, or both. This is strictly against the idea behind pure functions and you should therefore strictly use impure functions in such cases.

Recommendations

- Use `pure` functions whenever possible
  - Easier to understand and debug
- Use `impure` functions for
  - nondeterministic behavior
  - stateful functions
  - file I/O
  - protected types
- Especially for `impure` functions with side effects: **use sensible names!**
- **Caveat**: File parameters can introduce non-determinism and side effects into pure-functions
  - ⇒ Recommendation: Do not use file parameters for pure functions

Next, let us talk about procedures, the second kind of subprograms.

Procedures

■ Primarily to encapsulate sequential pieces of code (no return value)

While functions should primarily be used to compute values, procedures should be used to encapsulate sequential pieces of code that do *not* produce a value. To some extent, they can be compared to void functions in C or Java.

Procedures

- Primarily to encapsulate sequential pieces of code (no return value)

Primarily to encapsulate sequential pieces of code (no return value)
Do not return a value

There are two major differences when compared to functions. First, procedures **never** return a value.

Procedures

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, `return` statements without value possible for termination

However, since procedures might terminate at multiple points in their body, they can contain return statements without a value.

Procedures

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, `return` statements without value possible for termination

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, `return` statements without value possible for termination
- Can contain wait statements

Second, procedures are allowed to consume simulation time and therefore to contain wait statements.

Procedures

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, `return` statements without value possible for termination
- Can contain wait statements

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, `return` statements without value possible for termination
- Can contain wait statements
- Work via side effects

With procedures never returning a value, it is intuitively clear that they only work through side effects.

Procedures

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, `return` statements without value possible for termination
- Can contain wait statements
- Work via side effects

Primarily to encapsulate sequential pieces of code (no return value)
Do not return a value
However, return statements without value possible for termination
Can contain wait statements
Work via side effects
Access and modify outer scope

Therefore, procedures can access, and in particular modify, their outer scope. A procedure could, for example, drive the ports of an entity or signals declared in an architecture.

Procedures

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, `return` statements without value possible for termination
- Can contain wait statements
- Work via side effects
  - Access and modify outer scope

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, return statements without value possible for termination
- Can contain wait statements
- Work via side effects
  - Access and modify outer scope
- Simplified declaration syntax
procedure identifier[(parameter_list)] is [declarative_part] begin [statement part] -- procedure body end procedure;

Finally, let us look at the simplified declaration syntax of procedures, shown on the bottom of the slide.

Procedures

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, `return` statements without value possible for termination
- Can contain wait statements
- Work via side effects
  - Access and modify outer scope
- Simplified declaration syntax

```
procedure identifier[(parameter_list)] is
  [declarative_part]
begin
  [statement part]  -- procedure body
end procedure;
```

33

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, return statements without value possible for termination
- Can contain wait statements
- Work via side effects
  - Access and modify outer scope
- Simplified declaration syntax

```
procedure identifier[(parameter_list)] is
  [declarative_part]
begin
  [statement part]  -- procedure body
end procedure;
```


All in all, the syntax is quite similar to the one of functions, except that there is no return type and that a procedure does not have a designator, but rather just an identifier. Thus, operators cannot be implemented by procedures. However, this should already be intuitively clear by now, considering that an operation requires a result and thus a return value. Let us now discuss the properties of procedure parameters;

Procedures

- Primarily to encapsulate sequential pieces of code (no return value)
- Do not return a value
  - However, `return` statements without value possible for termination
- Can contain wait statements
- Work via side effects
  - Access and modify outer scope
- Simplified declaration syntax

```
procedure identifier[(parameter_list)] is
  [declarative_part]
begin
  [statement part]  -- procedure body
end procedure;
```

■ Simplified declaration syntax

Just like functions, procedures can have arbitrary many parameters of a scalar or composite type.

Procedure Parameters

- Simplified declaration syntax

Simplified declaration syntax
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
identifier_list: [in|out|inout] type

However, if we consider the simplified syntax shown on the slide, we can observe two differences compared to function parameter lists. Can you spot them?

Procedure Parameters

- Simplified declaration syntax

```
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
               identifier_list: [in|out|inout] type
```

Simplified declaration syntax
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
identifier_list: [in|out|inout] type

The most important difference is that you can declare a parameter's mode to be `in`, `out` or `inout`.

Procedure Parameters

- Simplified declaration syntax

```
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
                identifier_list: [in|out|inout] type
```

- Simplified declaration syntax
- Similar to entity port declaration

The semantics of these parameter modes are virtually the same as for entity ports of the respective mode.

Procedure Parameters

- Simplified declaration syntax

```
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
              identifier_list: [in|out|inout] type
```

- Similar to `entity` port declaration

In particular, a procedure can drive its parameters if they are of the mode `out` or `inout`.

Procedure Parameters

HWMod
WS24

Subprograms
Functions
Procedures
Overview
Parameters
Example
Overloading
Packages

- Simplified declaration syntax

```
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
                identifier_list: [in|out|inout] type
```

- Similar to `entity` port declaration
  - Procedures can drive `out` and `inout` parameters

Simplified declaration syntax
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
identifier_list: [in|out|inout] type
Similar to entity port declaration
Procedures can drive out and inout parameters

Furthermore, procedures can have parameters of the `variable` class. Since functions cannot modify their parameters they do not support this class and just treat respective instance as constants.

Procedure Parameters

- Simplified declaration syntax

```
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
                identifier_list: [in|out|inout] type
```

- Similar to `entity` port declaration
  - Procedures can drive `out` and `inout` parameters

Simplified declaration syntax
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
identifier_list: [in|out|inout] type
Similar to entity port declaration
Procedures can drive out and inout parameters
Default parameter mode and class are in and constant

Note that the default mode and class are in and constant.

Procedure Parameters

- Simplified declaration syntax

```
parameter_list ::= [parameters]{; parameters}
parameters::=[constant|signal|file|variable]
             identifier_list: [in|out|inout] type
```

- Similar to `entity` port declaration
  - Procedures can drive `out` and `inout` parameters
- Default parameter mode and class are `in` and `constant`

Apply pulse to probe, wait and check if alive is set
- probe, PULSE_WIDTH, alive, alive_cnt from outer scope

Let us now consider the example procedure shown on the slide. Note that `probe`, `PULSE_WIDTH`, `alive` and `alive_cnt` are assumed to be within the procedures scope.

Example - Procedure

Apply pulse to `probe`, wait and check if `alive` is set

- `probe`, `PULSE_WIDTH`, `alive`, `alive_cnt` from outer scope

```
procedure probe_alive(wait_time : time) is
begin
  probe <= '1';
  wait for PULSE_WIDTH;
  probe <= '0';
  wait for wait_time;
  if alive = '0' then
    report "Module not alive";
  else
    alive_cnt := alive_cnt + 1;
  end if;
end procedure;
```

The purpose of the `prob_alive` procedure is to apply a high pulse of a constant width to a signal called `probe` and to then wait and check for a response.

Example - Procedure

Apply pulse to `probe`, wait and check if `alive` is set

- `probe`, `PULSE_WIDTH`, `alive`, `alive_cnt` from outer scope

```
procedure probe_alive(wait_time : time) is
begin
  probe <= '1';
  wait for PULSE_WIDTH;
  probe <= '0';
  wait for wait_time;
  if alive = '0' then
    report "Module not alive";
  else
    alive_cnt := alive_cnt + 1;
  end if;
end procedure;
```

Apply pulse to `probe`, wait and check if `alive` is set
- `probe`, `PULSE_WIDTH`, `alive`, `alive_cnt` from outer scope

In a first step the procedure drives the `probe` signal from the outer scope to 1. Recall that this would not be possible within a function.

Example - Procedure

Apply pulse to `probe`, wait and check if `alive` is set

- `probe`, `PULSE_WIDTH`, `alive`, `alive_cnt` from outer scope

```
procedure probe_alive(wait_time : time) is
begin
  probe <= '1';
  wait for PULSE_WIDTH;
  probe <= '0';
  wait for wait_time;
  if alive = '0' then
    report "Module not alive";
  else
    alive_cnt := alive_cnt + 1;
  end if;
end procedure;
```

Apply pulse to probe, wait and check if alive is set
■ probe, PULSE_WIDTH, alive, alive_cnt from outer scope

```
procedure probe_alive(wait_time : time) is
begin
  probe <= '1';
  wait for PULSE_WIDTH;
  probe <= '0';
  wait for wait_time;
  if alive = '0' then
    report "Module not alive";
  else
    alive_cnt := alive_cnt + 1;
  end if;
end procedure;
```


Next, it uses a `wait` statement to wait for the outer scope's constant `PULSE_WIDTH` and resets the `probe` signal. Again, this is not achievable using a function.

Example - Procedure

Apply pulse to `probe`, wait and check if `alive` is set

- `probe`, `PULSE_WIDTH`, `alive`, `alive_cnt` from outer scope

```
procedure probe_alive(wait_time : time) is
begin
  probe <= '1';
  wait for PULSE_WIDTH;
  probe <= '0';
  wait for wait_time;
  if alive = '0' then
    report "Module not alive";
  else
    alive_cnt := alive_cnt + 1;
  end if;
end procedure;
```

Apply pulse to probe, wait and check if alive is set
■ probe, PULSE_WIDTH, alive, alive_cnt from outer scope
procedure probe_alive(wait_time : time) is
begin
probe <= '1';
wait for PULSE_WIDTH;
probe <= '0';
wait for wait_time;
if alive = '0' then
report "Module not alive";
else
alive_cnt := alive_cnt + 1;
end if;
end procedure;

Next the procedure wait for the value passed via its parameter and then checks if `alive` is active or not. In case of inactivity a message is printed and otherwise a variable incremented.

Example - Procedure

Apply pulse to `probe`, wait and check if `alive` is set

- `probe`, `PULSE_WIDTH`, `alive`, `alive_cnt` from outer scope

```
procedure probe_alive(wait_time : time) is
begin
  probe <= '1';
  wait for PULSE_WIDTH;
  probe <= '0';
  wait for wait_time;
  if alive = '0' then
    report "Module not alive";
  else
    alive_cnt := alive_cnt + 1;
  end if;
end procedure;
```

Apply pulse to probe, wait and check if alive is set
- probe, PULSE_WIDTH, alive, alive_cnt from outer scope

```
procedure probe_alive(wait_time : time) is
begin
  probe <= '1';
  wait for PULSE_WIDTH;
  probe <= '0';
  wait for wait_time;
  if alive = '0' then
    report "Module not alive";
  else
    alive_cnt := alive_cnt + 1;
  end if;
end procedure;
```


Admittedly, this procedure appears a bit artificial. However, it nicely shows that we can access and also modify objects from the outer scope, and use `wait` statements inside a procedure. Later, in chapter 2, we will use procedures to write concise and maintainable testbenches.

Example - Procedure

Apply pulse to `probe`, wait and check if `alive` is set

- `probe`, `PULSE_WIDTH`, `alive`, `alive_cnt` from outer scope

```
procedure probe_alive(wait_time : time) is
begin
  probe <= '1';
  wait for PULSE_WIDTH;
  probe <= '0';
  wait for wait_time;
  if alive = '0' then
    report "Module not alive";
  else
    alive_cnt := alive_cnt + 1;
  end if;
end procedure;
```

VHDL supports subprogram *overloading*

Similar to Java or C, VHDL supports the overloading of subprograms.

Subprogram Overloading

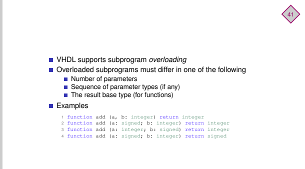
- VHDL supports subprogram *overloading*

VHDL supports subprogram *overloading*
Overloaded subprograms must differ in one of the following
Number of parameters
Sequence of parameter types (if any)
The result base type (for functions)

By that we mean that it is possible to have multiple subprograms with the same identifier or operator symbol, where either the parameter list or return type differ such that the compiler can derive which version of the function is the desired one.

Subprogram Overloading

- VHDL supports subprogram *overloading*
- Overloaded subprograms must differ in one of the following
  - Number of parameters
  - Sequence of parameter types (if any)
  - The result base type (for functions)



For illustration, the slide shows four functions with the same identifier where the compiler can easily determine the correct one.

Subprogram Overloading

- VHDL supports subprogram *overloading*
- Overloaded subprograms must differ in one of the following
  - Number of parameters
  - Sequence of parameter types (if any)
  - The result base type (for functions)
- Examples

```
function add (a, b: integer) return integer
function add (a: signed; b: integer) return integer
function add (a: integer; b: signed) return integer
function add (a: signed; b: integer) return signed
```

41
- VHDL supports subprogram *overloading*
- Overloaded subprograms must differ in one of the following
  - Number of parameters
  - Sequence of parameter types (if any)
  - The result base type (for functions)
- Examples

```vhdl
function add (a, b: integer) return integer
function add (a: signed; b: integer) return integer
function add (a: integer; b: signed) return integer
function add (a: signed; b: integer) return signed
```


Note how the second and the third function only differ in the order of their parameter types.

Subprogram Overloading

- VHDL supports subprogram *overloading*
- Overloaded subprograms must differ in one of the following
  - Number of parameters
  - Sequence of parameter types (if any)
  - The result base type (for functions)
- Examples

```vhdl
function add (a, b: integer) return integer
function add (a: signed; b: integer) return integer
function add (a: integer; b: signed) return integer
function add (a: signed; b: integer) return signed
```

Furthermore, the second and the fourth function declarations only differ in their return type.

Subprogram Overloading

- VHDL supports subprogram *overloading*
- Overloaded subprograms must differ in one of the following
  - Number of parameters
  - Sequence of parameter types (if any)
  - The result base type (for functions)
- Examples

```
function add (a, b: integer) return integer
function add (a: signed; b: integer) return integer
function add (a: integer; b: signed) return integer
function add (a: signed; b: integer) return signed
```

Subprograms can be declared in the declaration sections of

Finally, let us end this lecture by discussing where subprograms can be declared.

Subprograms in Packages

- Subprograms can be declared in the declaration sections of

Subprograms can be declared in the declaration sections of
- entities and architectures
- processes and subprograms

In principle, such declarations can occur in all declaration sections we encountered so far. In particular, this means that you can declare subprograms in the declarative section of an `entity`, an `architecture`, a `process` or even a subprogram itself.

Subprograms in Packages

- Subprograms can be declared in the declaration sections of
  - entities and architectures
  - processes and subprograms

Subprograms can be declared in the declaration sections of
entities and architectures
processes and subprograms
To facilitate reusability also possible in packages

However, none of these really facilitates wide-spread reuse of subprograms as all these declaration sections belong to specific modules. This is where we can make use of packages. Recall that we have already seen packages in previous lectures, where we considered them to be somewhat akin to C libraries or Java modules. However, so far we have only shown you one half of what a package really comprises.

Subprograms in Packages

- Subprograms can be declared in the declaration sections of
  - entities and architectures
  - processes and subprograms
- To facilitate reusability also possible in packages

45

- Subprograms can be declared in the declaration sections of
  - entities and architectures
  - processes and subprograms
- To facilitate reusability also possible in packages
  - The *package declaration* contains declarations (e.g., constants, types, components, subprograms etc.)

In particular, you know about the declarative part of a package. This part contains declarations of constants, types, components and also subprograms to name the most important ones. However, in order to keep packages clean and comprehensible, subprogram declarations in packages are split into their signature and their body, where only the signature will be contained in the package's declarative section.

Subprograms in Packages

- Subprograms can be declared in the declaration sections of
  - entities and architectures
  - processes and subprograms
- To facilitate reusability also possible in packages
  - The *package declaration* contains declarations (e.g., constants, types, components, subprograms etc.)

Subprograms can be declared in the declaration sections of
entities and architectures
processes and subprograms
To facilitate reusability also possible in packages
The *package declaration* contains declarations (e.g., constants, types, components, subprograms etc.)
The *package body* contains the subprogram bodies

The bodies of subprograms, meaning their declarative and statement parts, must be contained in the so-called package body.

Subprograms in Packages

- Subprograms can be declared in the declaration sections of
  - entities and architectures
  - processes and subprograms
- To facilitate reusability also possible in packages
  - The *package declaration* contains declarations (e.g., constants, types, components, subprograms etc.)
  - The *package body* contains the subprogram bodies

45
- Subprograms can be declared in the declaration sections of
  - entities and architectures
  - processes and subprograms
- To facilitate reusability also possible in packages
  - The *package declaration* contains declarations (e.g., constants, types, components, subprograms etc.)
  - The *package body* contains the subprogram bodies
  - ⇒ Similar to C function prototype and definition

While this separation might seem odd initially, this is exactly what the C language does as well, with its function prototypes and definitions. We will now consider an example to illustrate the declaration of subprograms in packages.

Subprograms in Packages

- Subprograms can be declared in the declaration sections of
  - entities and architectures
  - processes and subprograms
- To facilitate reusability also possible in packages
  - The *package declaration* contains declarations (e.g., constants, types, components, subprograms etc.)
  - The *package body* contains the subprogram bodies
  - ⇒ Similar to C function prototype and definition

Consider the example package shown on the slide.

Example - Package with Body


```
package math_pkg is
  constant WIDTH, HEIGHT : natural := 100;
  function max3(value1, value2, value3 : integer) return integer;
end package;

package body math_pkg is
  -- package-local auxiliary function; must be declared before use in max3
  function max(value1, value2 : integer) return integer is
  begin
    if value1 > value2 then
      return value1;
    end if;
    return value2;
  end function;

  function max3(value1, value2, value3 : integer) return integer is
  begin
    return max(max(value1, value2), value3);
  end function;
end package body;
```

The first part is the declarative one we already know from previous lectures. It declares two constants, `width` and `height`, and a function named `max3` that computes the maximum of three `integer` numbers.

Example - Package with Body


```
package math_pkg is
  constant WIDTH, HEIGHT : natural := 100;
  function max3(value1, value2, value3 : integer) return integer;
end package;

package body math_pkg is
  -- package-local auxiliary function; must be declared before use in max3
  function max(value1, value2 : integer) return integer is
  begin
    if value1 > value2 then
      return value1;
    end if;
    return value2;
  end function;

  function max3(value1, value2, value3 : integer) return integer is
  begin
    return max(max(value1, value2), value3);
  end function;
end package body;
```

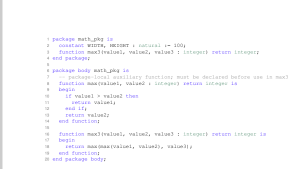
Below this declarative part, we can see a package body that contains the implementation of all functions declared in the declarative part of the package. Since the compiler is aware of all subprograms signatures due to the package declarations, the implementations inside the body of these subprograms can be in arbitrary order.

Example - Package with Body


```
package math_pkg is
  constant WIDTH, HEIGHT : natural := 100;
  function max3(value1, value2, value3 : integer) return integer;
end package;

package body math_pkg is
  -- package-local auxiliary function; must be declared before use in max3
  function max(value1, value2 : integer) return integer is
  begin
    if value1 > value2 then
      return value1;
    end if;
    return value2;
  end function;

  function max3(value1, value2, value3 : integer) return integer is
  begin
    return max(max(value1, value2), value3);
  end function;
end package body;
```



Finally, be aware that is also possible to have subprograms inside the package body only, meaning without their signature occurring in the declarative part. This is useful if you require some package-internal auxiliary functions.

Example - Package with Body


```
package math_pkg is
  constant WIDTH, HEIGHT : natural := 100;
  function max3(value1, value2, value3 : integer) return integer;
end package;

package body math_pkg is
  -- package-local auxiliary function; must be declared before use in max3
  function max(value1, value2 : integer) return integer is
  begin
    if value1 > value2 then
      return value1;
    end if;
    return value2;
  end function;

  function max3(value1, value2, value3 : integer) return integer is
  begin
    return max(max(value1, value2), value3);
  end function;
end package body;
```

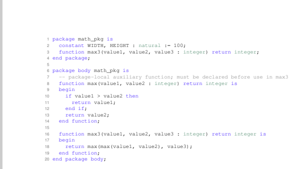
The example shows such a package-local auxiliary function, named `max`, which is used by `max3`.

Example - Package with Body


```
package math_pkg is
  constant WIDTH, HEIGHT : natural := 100;
  function max3(value1, value2, value3 : integer) return integer;
end package;

package body math_pkg is
  -- package-local auxiliary function; must be declared before use in max3
  function max(value1, value2 : integer) return integer is
  begin
    if value1 > value2 then
      return value1;
    end if;
    return value2;
  end function;

  function max3(value1, value2, value3 : integer) return integer is
  begin
    return max(max(value1, value2), value3);
  end function;
end package body;
```



Be aware that subprograms which are only declared in the package body are not visible outside this body. Therefore, if you would import the shown math package into a program of yours, `max` would not be visible. Finally, we want to point out that such local subprograms must be declared *before* their use, as is the case in the example.

Example - Package with Body


```
package math_pkg is
  constant WIDTH, HEIGHT : natural := 100;
  function max3(value1, value2, value3 : integer) return integer;
end package;

package body math_pkg is
  -- package-local auxiliary function; must be declared before use in max3
  function max(value1, value2 : integer) return integer is
  begin
    if value1 > value2 then
      return value1;
    end if;
    return value2;
  end function;

  function max3(value1, value2, value3 : integer) return integer is
  begin
    return max(max(value1, value2), value3);
  end function;
end package body;
```

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!