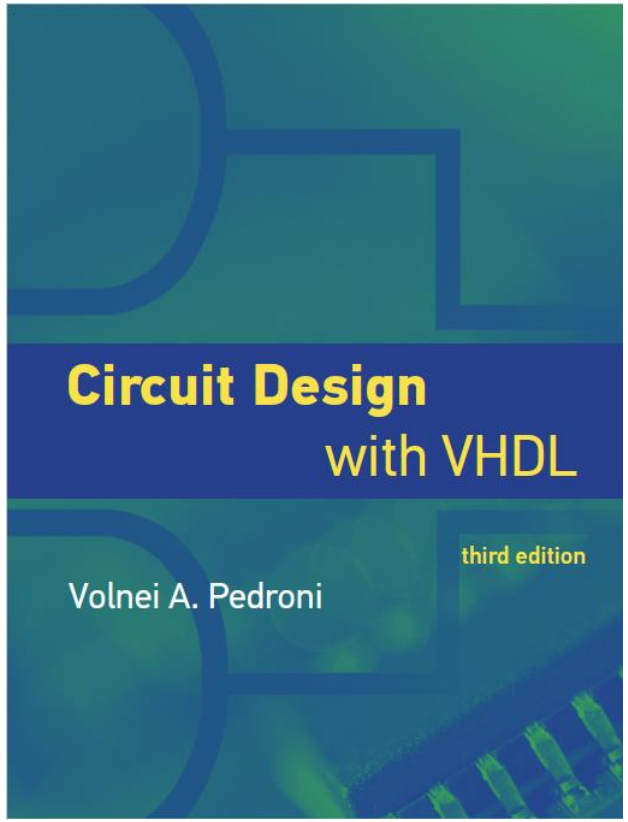


Circuit Design with VHDL

3rd Edition

Volnei A. Pedroni

MIT Press, 2020

Slides Chapter 14

Packages and Subprograms

Revision 1

Book Contents

Part I: Digital Circuits Review

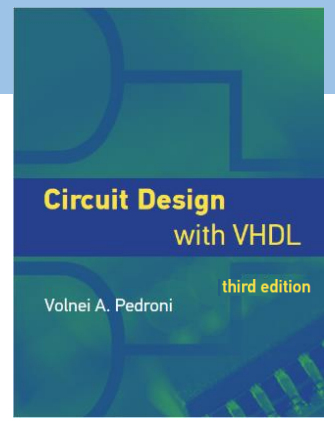
1. Review of Combinational Circuits
2. Review of Combinational Circuits
3. Review of State Machines
4. Review of FPGAs

Part II: VHDL

5. Introduction to VHDL
6. Code Structure and Composition
7. Predefined Data Types
8. User-Defined Data Types
9. Operators and Attributes
10. Concurrent Code
11. Concurrent Code – Practice
12. Sequential Code
13. Sequential Code – Practice
14. Packages and Subprograms
15. The Case of State Machines
16. The Case of State Machines – Practice
17. Additional Design Examples
18. Intr. to Simulation with Testbenches

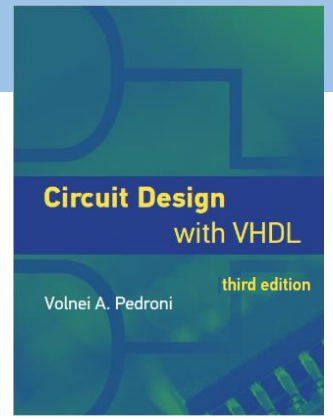
Appendices

A. Vivado Tutorial
B. Quartus Prime Tutorial
C. ModelSim Tutorial
D. Simulation Analysis and Recommendations
E. Using Seven-Segment Displays with VHDL
F. Serial Peripheral Interface
G. I2C (Inter Integrated Circuits) Interface
H. Alphanumeric LCD
I. VGA Video Interface
J. DVI Video Interface
K. TMDS Link
L. Using Phase-Locked Loops with VHDL
M. List of Enumerated Examples and Exercises



VHDL for Synthesis Slides

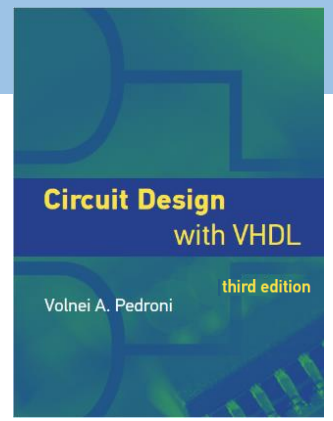
Chapter	Title
5	Introduction to VHDL
6	Code Structure and Composition
7	Predefined Data Types
8	User-Defined Data Types
9	Operators and Attributes
10	Concurrent Code
11	Concurrent Code – Practice
12	Sequential Code
13	Sequential Code – Practice
14	Packages and Subprograms



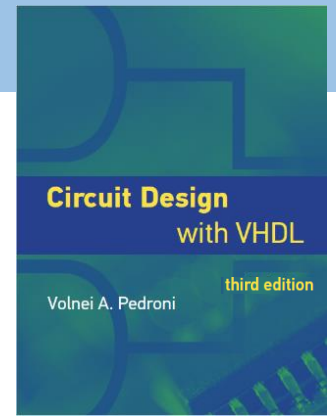
Chapter 14

Packages and Subprograms

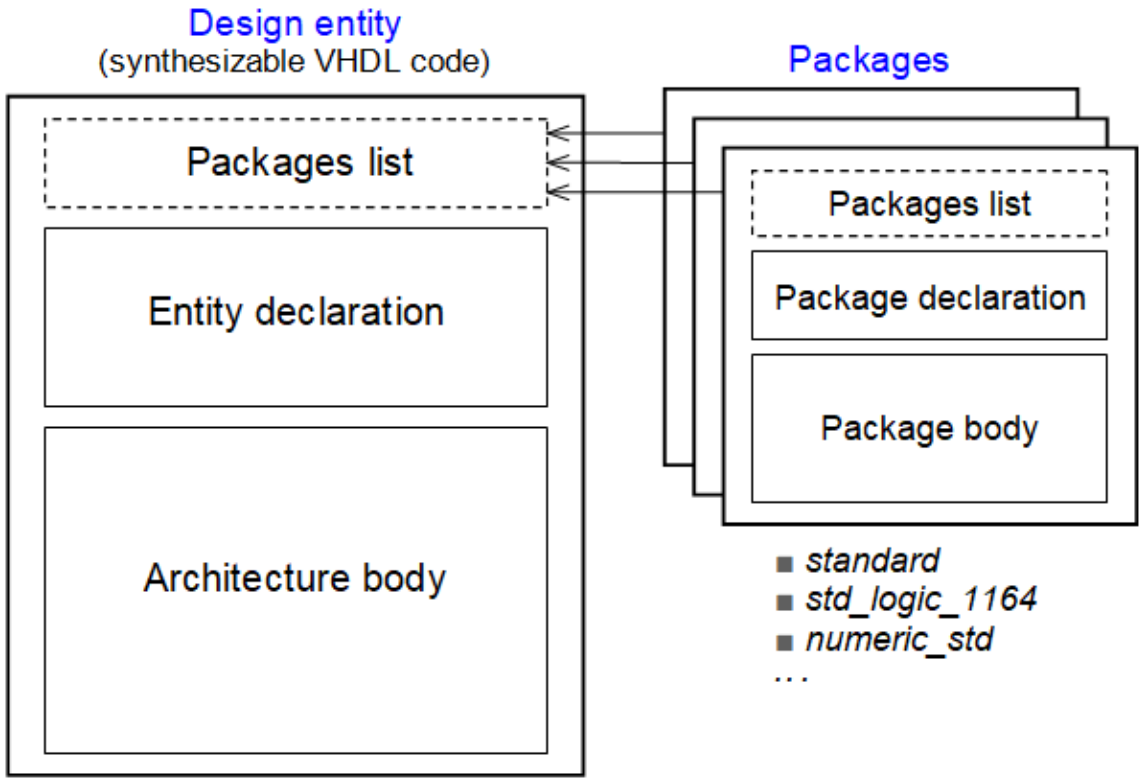
1. Package
2. Package with generics
3. Subprograms
4. Function
5. Procedure
6. Subprogram with generics and generic subprograms
7. The *assert* statement
8. The *report* statement



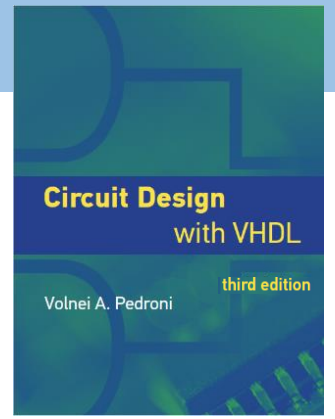
1. Package



1. Package



- Package = Collection of declarations that are common to a model
- Prevents common solutions from being rewritten
- Helps code organization, simplification, and reusability

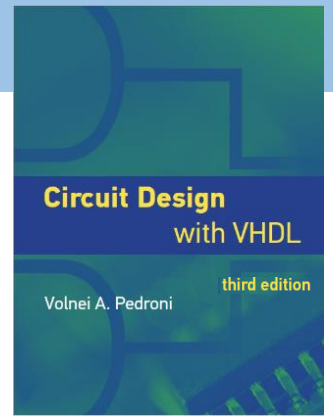


1. Package

Syntax:

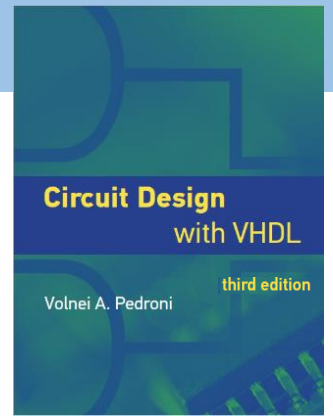
```
[libraries/packages list]

package package_name is
   declarative_part
end [package] [package_name];

-----------------------------------

[package body package_name is
   [subprogram_body]
   [deferred_constant_specification]
end [package body] [package_name]];
```

- Package declaration:
  Can contain declarations of subprogram, type, constant, signal, variable, package, etc.
- Package body:
  - Needed only when the package contains a declaration of subprogram or deferred constant
  - Must show the subprogram body and/or the constant's value-defining mechanism



1. Package

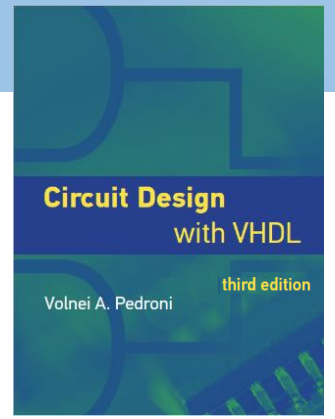
Example: A package with type and subprogram declarations



1. Package

Example: A package with type and subprogram declarations

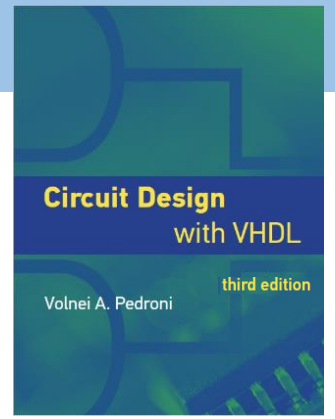
```
-------------------------------------------------------------
package example_pkg is
   type signed_array is array (1 to 8) of signed(7 downto 0);
   function negative_edge(signal s: bit) return boolean;
end package;

package body example_pkg is

      ???

end package body;
-------------------------------------------------------------
```



1. Package

Example: A package with type and subprogram declarations

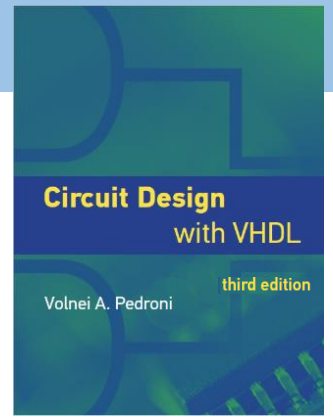
```
-------------------------------------------------------------
package example_pkg is
   type signed_array is array (1 to 8) of signed(7 downto 0);
   function negative_edge(signal s: bit) return boolean;
end package;

package body example_pkg is
   function negative_edge(signal s: bit) return boolean is
   begin
      return (s'event and s='0');
   end function negative_edge;
end package body;
-------------------------------------------------------------
```



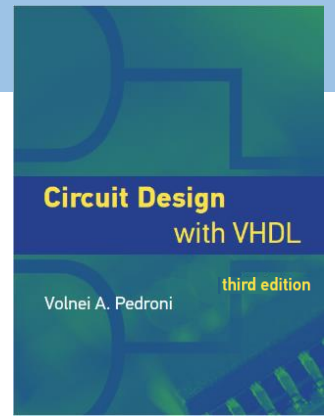
Chapter 14

Packages and Subprograms

1. Package
2. Package with generics
3. Subprograms
4. Function
5. Procedure
6. Subprogram with generics and generic subprograms
7. The *assert* statement
8. The *report* statement



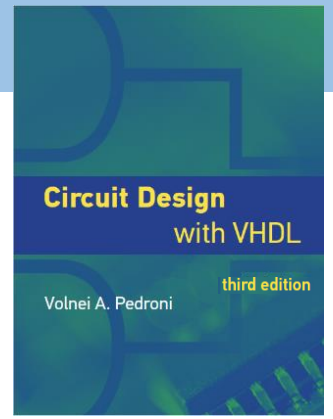
2. Package with generics



2. Package with generics

- VHDL-2008 allows the inclusion of a generic list in a package:

```
[libraries/packages list]

package package_name is
   generic (generic_list);
   package_declarative_part
end [package] [package_name];

-----------------------------------------

[package body package_name is
   [subprogram_body]
   [deferred_constant_specification]
end [package body] [package_name]];
```

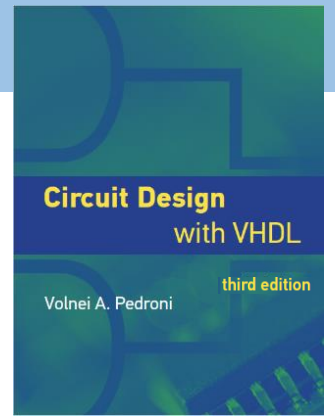


2. Package with generics

- Such a package is said to be:

  a) A *generic-mapped package* if a generic map association is included

  b) An *uninstantiated package* otherwise, i.e.
  - without a *generic map* list or
  - with default values for all generics



2. Package with generics

- Such a package is said to be:

  a) A *generic-mapped package* if a generic map association is included

  b) An *uninstantiated package* otherwise, i.e.
  - without a *generic map* list or
  - with default values for all generics

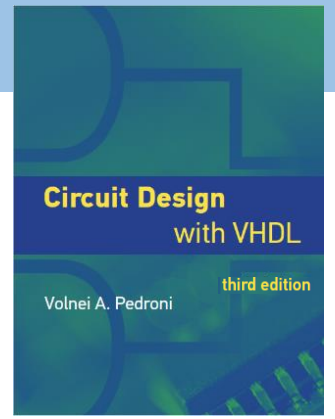
- To use the latter, it must first be instantiated, using the *new* keyword:

When without a generic list:

```
package package_name is new work.uninstantiated_pkg_name
   generic map (association_list);
```

When with default values:

```
package package_name is new work.uninstantiated_pkg_name
   generic map (default);
```



2. Package with generics

- Such a package is said to be:

  a) A *generic-mapped package* if a generic map association is included

  b) An *uninstantiated package* otherwise, i.e.
  - without a *generic map* list or
  - with default values for all generics

- To use the latter, it must first be instantiated, using the *new* keyword:

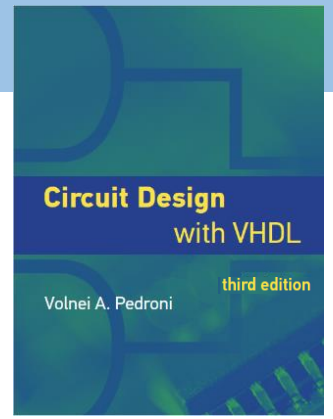
When without a generic list:

```
package package_name is new work.uninstantiated_pkg_name
   generic map (association_list);
```

When with default values:

```
package package_name is new work.uninstantiated_pkg_name
   generic map (default);
```

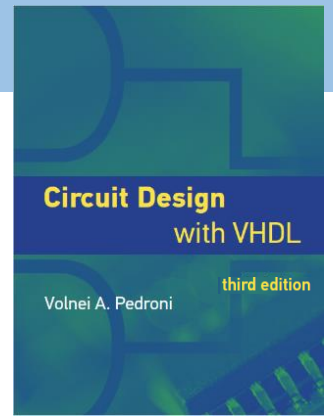
- Full details are presented in section 14.2



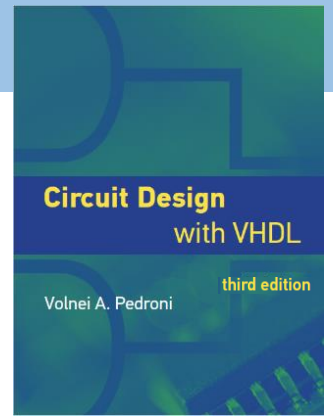
Chapter 14

Packages and Subprograms

1. Package
2. Package with generics
3. Subprograms
4. Function
5. Procedure
6. Subprogram with generics and generic subprograms
7. The *assert* statement
8. The *report* statement



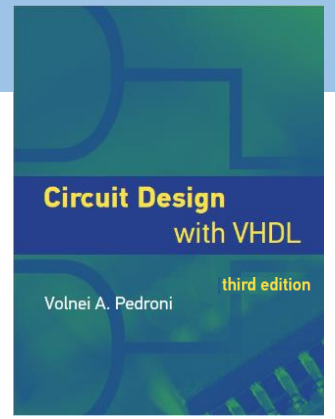
3. Subprograms



3. Subprograms

Subprogram concepts:

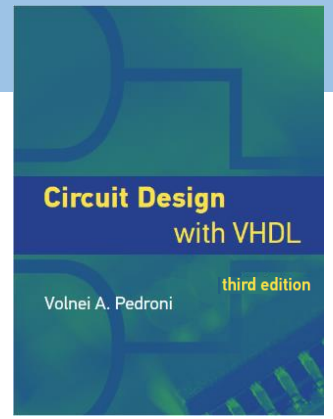
- function and procedure are called subprograms
- process and subprograms are the only regions of sequential code
- But while process is for the code proper, subprograms are separate units, to solve common problems, hence intended to be reused and often located in libraries (inside packages)



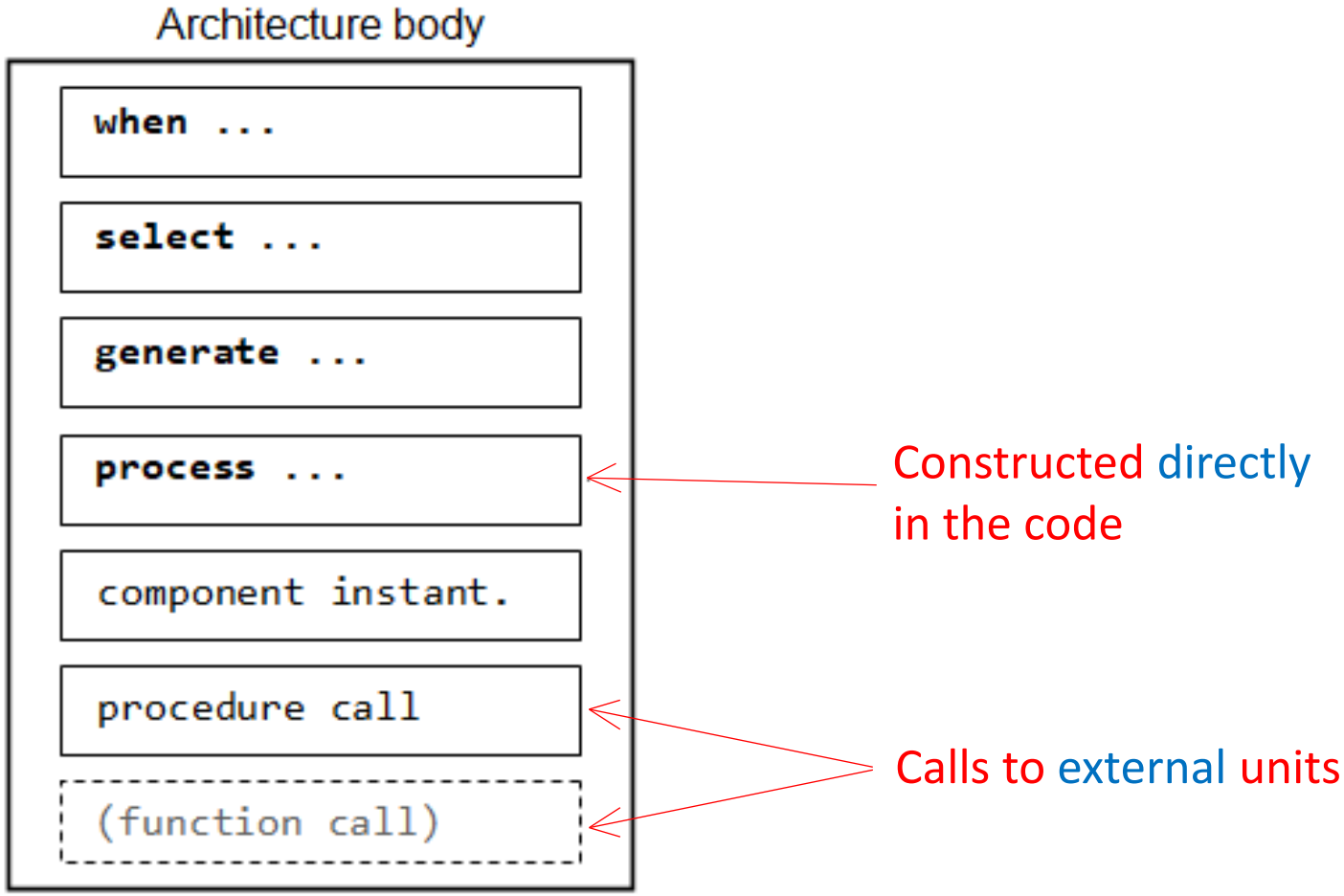
3. Subprograms

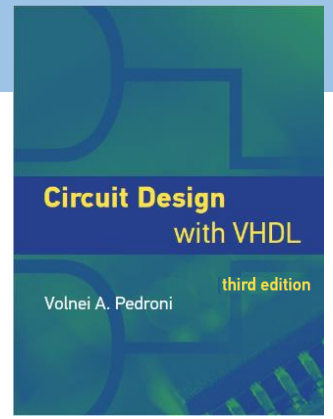
Subprogram concepts:

- function and procedure are called subprograms
- process and subprograms are the only regions of sequential code
- But while process is for the code proper, subprograms are separate units, to solve common problems, hence intended to be reused and often located in libraries (inside packages)
- Example: Nearly all predefined operators (chapter 9) are functions



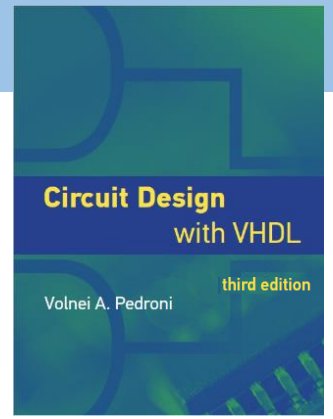
3. Subprograms





3. Subprograms

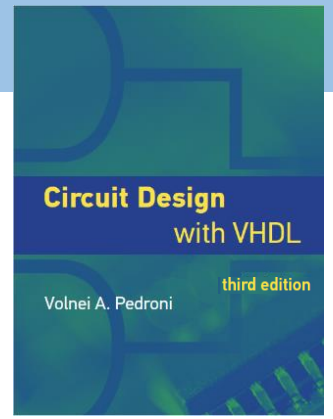
Subprogram construction:



3. Subprograms

Subprogram construction:

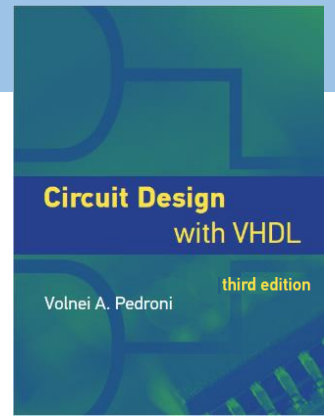
- Allowed places: package, entity, architecture, process, or another subprogram (block not used)



3. Subprograms

Subprogram construction:

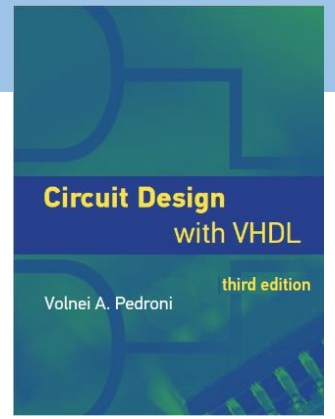
- Allowed places: package, entity, architecture, process, or another subprogram (block not used)
- Most *common* places: package and architecture (declarative region, of course)



3. Subprograms

Subprogram construction:

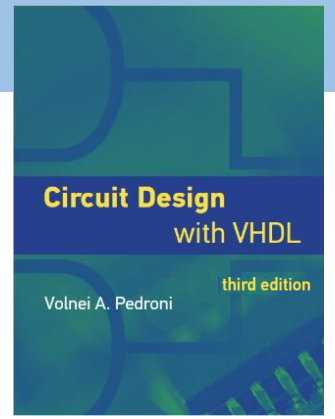
- Allowed places: package, entity, architecture, process, or another subprogram (block not used)
- Most *common* places: package and architecture (declarative region, of course)
- Statements allowed in procedure: All sequential statements (*if*, *case*, *loop*, *wait*, plus sequential versions of *when* and *select*)



3. Subprograms

Subprogram construction:

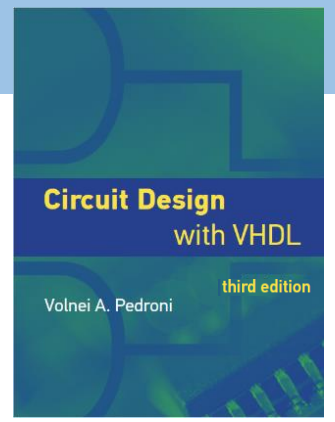
- Allowed places: package, entity, architecture, process, or another subprogram (block not used)
- Most *common* places: package and architecture (declarative region, of course)
- Statements allowed in procedure: All sequential statements (*if*, *case*, *loop*, *wait*, plus sequential versions of *when* and *select*)
- Statements allowed in function: All above, except *wait*



3. Subprograms

Subprogram construction:

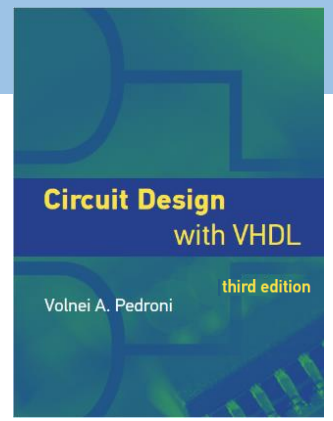
- Allowed places: package, entity, architecture, process, or another subprogram (block not used)
- Most *common* places: package and architecture (declarative region, of course)
- Statements allowed in procedure: All sequential statements (*if*, *case*, *loop*, *wait*, plus sequential versions of *when* and *select*)
- Statements allowed in function: All above, except *wait*
- Subprogram calls can be made in both concurrent and sequential codes



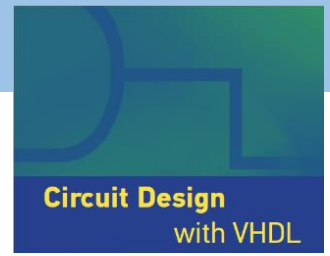
Chapter 14

Packages and Subprograms

1. Package
2. Package with generics
3. Subprograms
4. Function
5. Procedure
6. Subprogram with generics and generic subprograms
7. The *assert* statement
8. The *report* statement



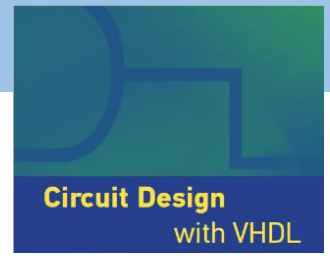
4. Function



4. Function

```
[pure|impure] function function_name [(input_list)] return return_type is
   [declarative_part]
begin
   sequential_statement_part
end [function] [function_name];
```

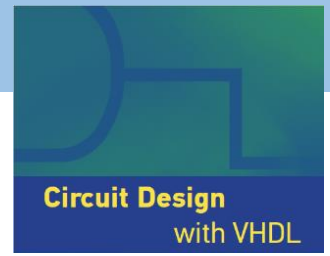
- It is "pure" when it does not rely on external objects, so it always returns the same result when called with the same parameter values
- Don't use "impure" functions for synthesis (and even for simulation, in general)



4. Function

```
[pure|impure] function function_name [(input_list)] return return_type is
   [declarative_part]
begin
   sequential_statement_part
end [function] [function_name];
```

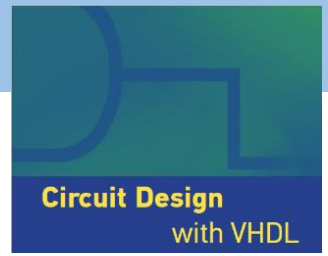
- It is "pure" when it does not rely on external objects, so it always returns the same result when called with the same parameter values
- Don't use "impure" functions for synthesis (and even for simulation, in general)
- A function can receive any number of parameters, but must return exactly one



4. Function

```
[pure|impure] function function_name [(input_list)] return return_type is
   [declarative_part]
begin
   sequential_statement_part
end [function] [function_name];
```

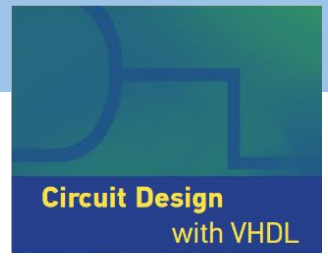
- It is "pure" when it does not rely on external objects, so it always returns the same result when called with the same parameter values
- Don't use "impure" functions for synthesis (and even for simulation, in general)
- A function can receive any number of parameters, but must return exactly one
- Such parameters can be constant (default), signal, or file (the word variable is forbidden in the input_list)



4. Function

```
[pure|impure] function function_name [(input_list)] return return_type is
   [declarative_part]
begin
   sequential_statement_part
end [function] [function_name];
```

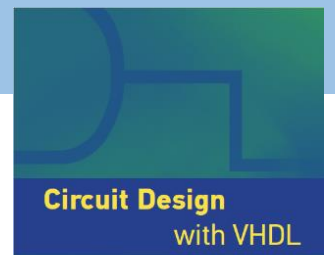
- The declarative_part accepts declarations of variables but not of signals



4. Function

```
[pure|impure] function function_name [(input_list)] return return_type is
   [declarative_part]
begin
   sequential_statement_part
end [function] [function_name];
```

- The declarative_part accepts declarations of variables but not of signals
- The statement_part cannot contain *wait* statements and component instantiations

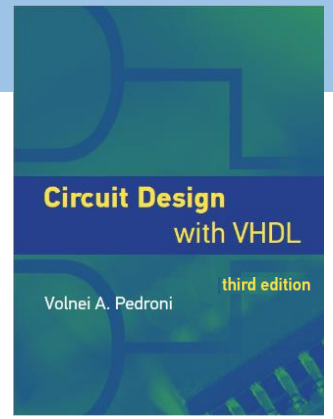


4. Function

```
[pure|impure] function function_name [(input_list)] return return_type is
   [declarative_part]
begin
   sequential_statement_part
end [function] [function_name];
```

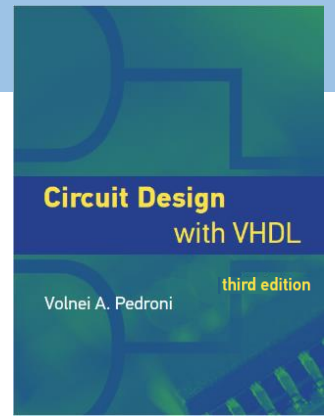
- The declarative_part accepts declarations of variables but not of signals
- The statement_part cannot contain *wait* statements and component instantiations
- A function call is always part of an expression; for example:

```
y <= mean(a, b);                 --mean is a function
y <= a + b;                      --"+" is a function
if rising_edge(clk) then ...     --rising_edge is a function
```



4. Function

Example: Function *ceil_log2*, constructed in a package

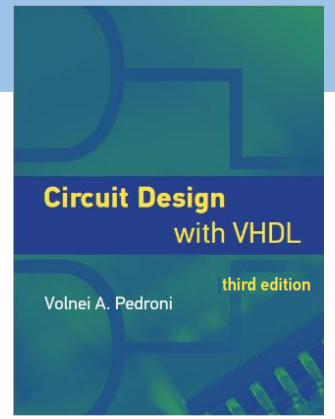


4. Function

Example: Function *ceil_log2*, constructed in a package

a) Function construction:

```
-------------------------------------------------------
package subprograms_pkg is
   function ceil_log2 (input: positive) return natural;
end package;

package body subprograms_pkg is
   function ceil_log2 (input: positive) return natural is
      variable result: natural := 0;
   begin
      while 2**result < input loop
         result := result + 1;
      end loop;
      return result;
   end function ceil_log2;
end package body;
-------------------------------------------------------
```

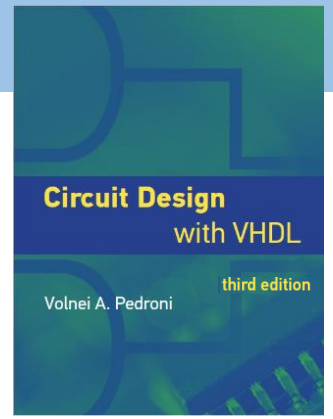


4. Function

Example: Function *ceil_log2*, constructed in a package

b) Function call:

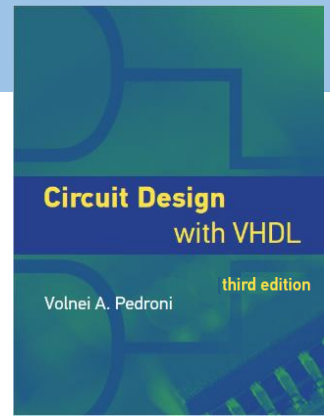
```
-------------------------------------------------
use work.subprograms_pkg.all;

entity test_circuit is
   generic (
      BITS: natural := 8);
   port (
      inp: in positive range 1 to 2**BITS-1;
      outp: out natural range 0 to BITS);
end entity;

architecture test_circuit of test_circuit is
begin
   outp <= ceil_log2(inp);
end architecture;
-------------------------------------------------
```



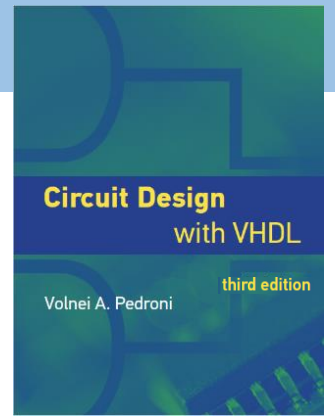
Chapter 14

Packages and Subprograms

1. Package
2. Package with generics
3. Subprograms
4. Function
5. Procedure
6. Subprogram with generics and generic subprograms
7. The *assert* statement
8. The *report* statement



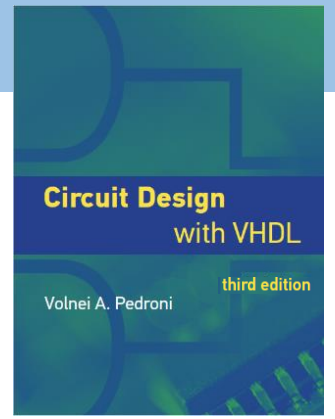
5. Procedure



5. Procedure

```
procedure procedure_name [(input_output_list)] is
   [declarative_part]
begin
   sequential_statement_part
end [procedure] [procedure_name]
```

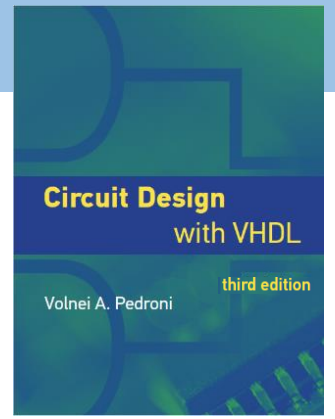
- A procedure can receive and return any number of parameters (multi-output problems)
- The input_output_list can contain objects of any class (constant, signal, variable, or file)
- Default for mode *in* is constant; default for modes *out* and *inout* is variable



5. Procedure

```
procedure procedure_name [(input_output_list)] is
   [declarative_part]
begin
   sequential_statement_part
end [procedure] [procedure_name]
```

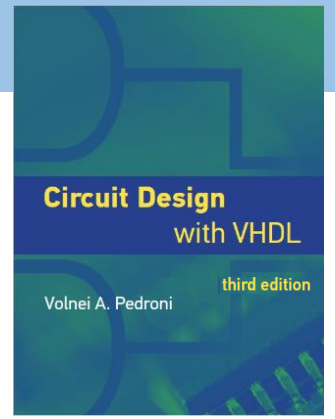
- A procedure can receive and return any number of parameters (multi-output problems)
- The input_output_list can contain objects of any class (constant, signal, variable, or file)
- Default for mode *in* is constant; default for modes *out* and *inout* is variable
- Like functions, it cannot contain signal declarations and component instantiations



5. Procedure

```
procedure procedure_name [(input_output_list)] is
   [declarative_part]
begin
   sequential_statement_part
end [procedure] [procedure_name]
```

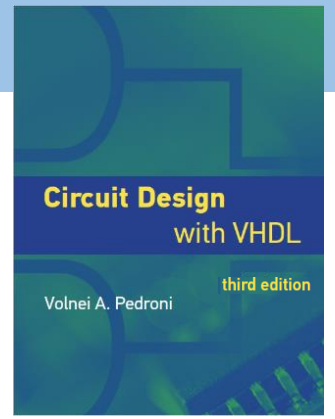
- A procedure can receive and return any number of parameters (multi-output problems)
- The input_output_list can contain objects of any class (constant, signal, variable, or file)
- Default for mode *in* is constant; default for modes *out* and *inout* is variable
- Like functions, it cannot contain signal declarations and component instantiations
- But contrary to functions, the *wait* statement is allowed



5. Procedure

```
procedure procedure_name [(input_output_list)] is
   [declarative_part]
begin
   sequential_statement_part
end [procedure] [procedure_name]
```

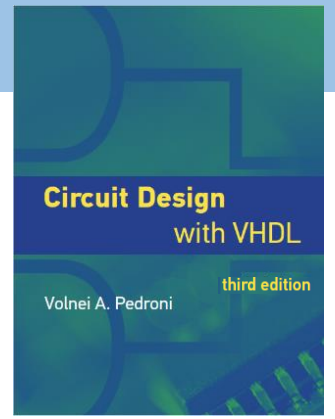
- Procedures can deal with time (with *wait* statement)



5. Procedure

```
procedure procedure_name [(input_output_list)] is
   [declarative_part]
begin
   sequential_statement_part
end [procedure] [procedure_name]
```

- Procedures can deal with time (with *wait* statement)
- And can easily deal (read and write) with files (important for simulation)

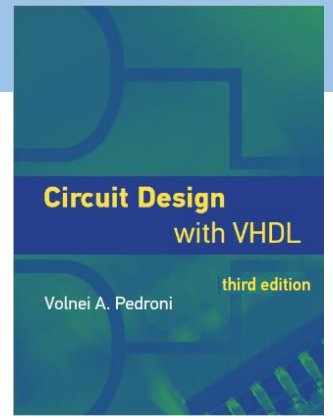


5. Procedure

```
procedure procedure_name [(input_output_list)] is
   [declarative_part]
begin
   sequential_statement_part
end [procedure] [procedure_name]
```

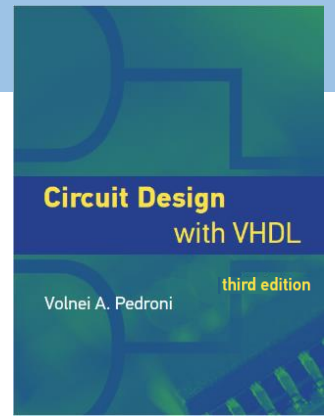
- Procedures can deal with time (with *wait* statement)
- And can easily deal (read and write) with files (important for simulation)
- A procedure call is a standalone statement; for example:

```
sort (a1, a2, a3, b1, b2, b3);                    --sort is a procedure
divide (dividend, divisor, quotient, remainder);  --divide is a procedure
```



5. Procedure

Example: Procedure *divide*, constructed in a package

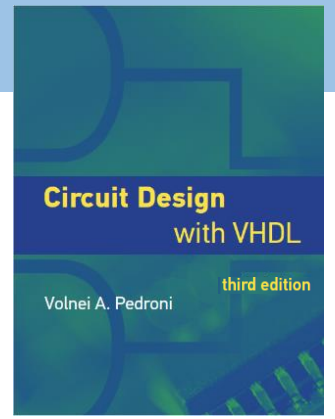


5. Procedure

Example: Procedure *divide*, constructed in a package

a) Procedure construction:

```
-------------------------------------------------------------------
library ieee;
use ieee.numeric_std.all;

package subprograms_pkg is
   procedure divide (dividend, divisor: in unsigned;
      signal quotient, remainder: out unsigned);
end package;

package body subprograms_pkg is
   procedure divide (dividend, divisor: in unsigned;
      signal quotient, remainder: out unsigned) is
   begin
      quotient <= dividend / divisor;
      remainder <= dividend rem divisor;
   end procedure;
end package body;
-------------------------------------------------------------------
```

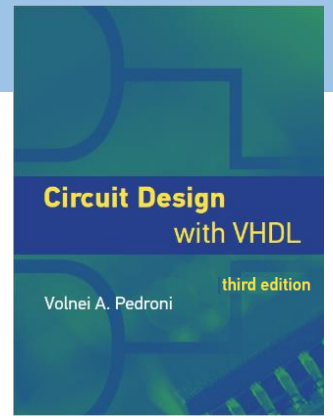


5. Procedure

Example: Procedure *divide*, constructed in a package

b) Procedure call:

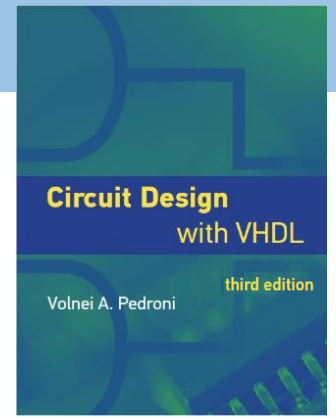
```
-----------------------------------------
library ieee;
use ieee.numeric_std.all;
use work.subprograms_pkg.all;

entity test is
   generic (
      BITS: positive := 8);
   port (
      a, b: in unsigned(BITS-1 downto 0);
      q, r: out unsigned(BITS-1 downto 0));
end entity;

architecture rtl of test is
begin
   divide (a, b, q, r);
end architecture;
-----------------------------------------
```



5. Procedure

Example: Procedure *divide*, constructed in a package

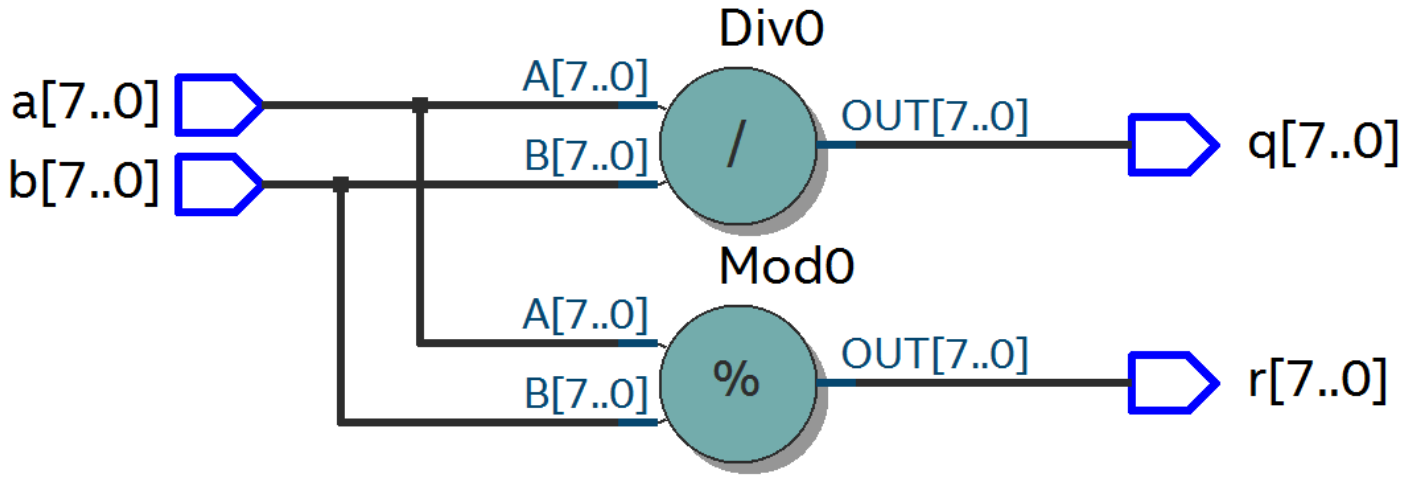
RTL view:

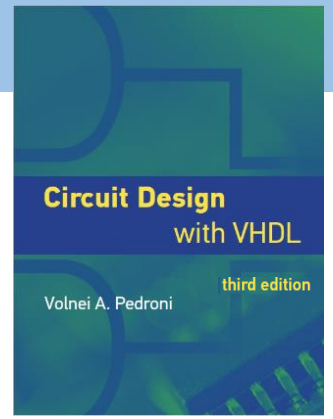


5. Procedure

Example: Procedure *divide*, constructed in a package

RTL view:

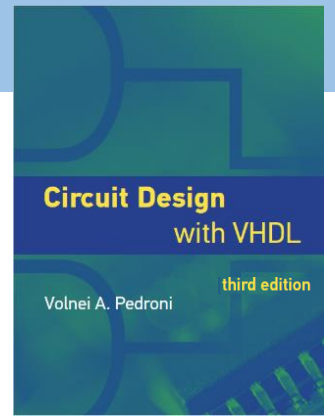




5. Procedure

Example: Procedure *divide*, constructed in a package

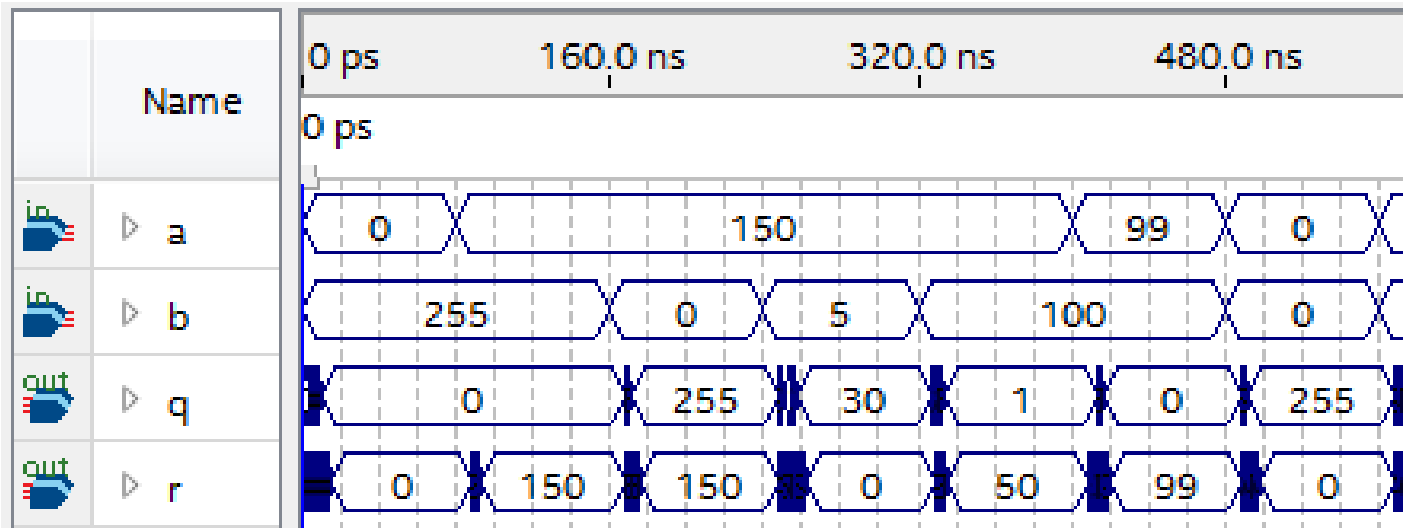
Simulation results:



5. Procedure

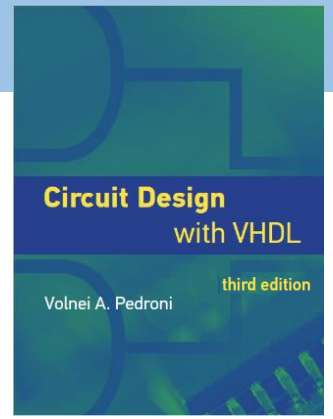
Example: Procedure *divide*, constructed in a package

Simulation results:



Notes:

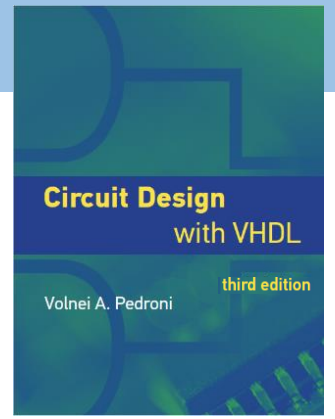
- Division-by-zero not checked in the code
- Simulator's default: Division-by-zero saturates the output and makes remainder = divider



5. Procedure

Example: Procedure *divide*, constructed in a package

What we didn't check in the code:



5. Procedure

Example: Procedure *divide*, constructed in a package

What we didn't check in the code:

- Division-by-zero
- What table 9.4 says about the "/" (division) operator
- What table 9.4 says about the "rem" (remainder) operator

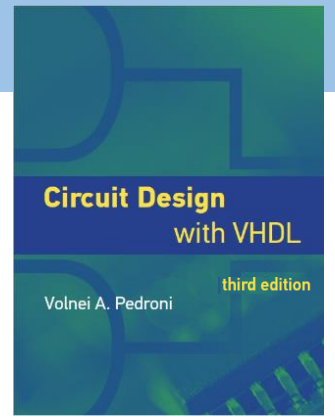
Table 9.4

Arithmetic operators (including output sizes or ranges)

Binary operations							Unary operations		
Left, Right	Outp	+, –	*	/	**	rem, mod	Inp	Outp	–, abs
UNS, UNS		(8)	(11)	(14)	–	(15)	—	—	—

(14) L'length-1 downto 0 (output length = length of input on the left)

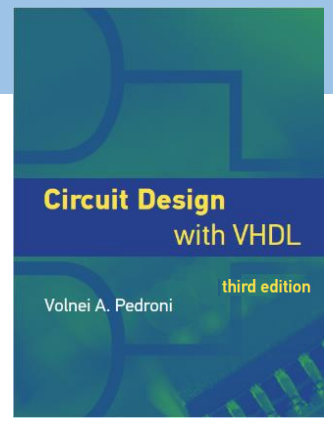
(15) R'length-1 downto 0 (output length = length of input on the right)



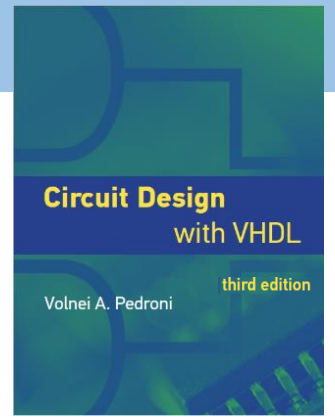
Chapter 14

Packages and Subprograms

1. Package
2. Package with generics
3. Subprograms
4. Function
5. Procedure
6. Subprogram with generics and generic subprograms
7. The *assert* statement
8. The *report* statement



6. Subprogram with generics and generic subprograms

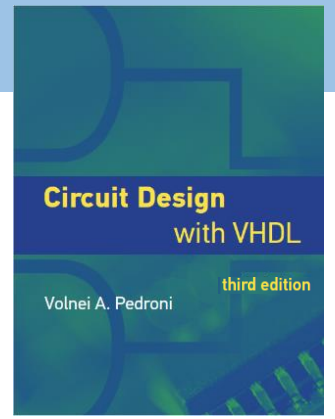


6. Subprogram with generics and generic subprograms

- VHDL-2008 allows the inclusion of a generic list is a subprogram, resulting the (complete) syntaxes below:

```
[pure|impure] function function_name
   [generic (generic_list)]
   [[parameter] (input_list)] return return_type is
   [function_declarative_part]
begin
   sequential_statement_part
end [function] [function_name];
```

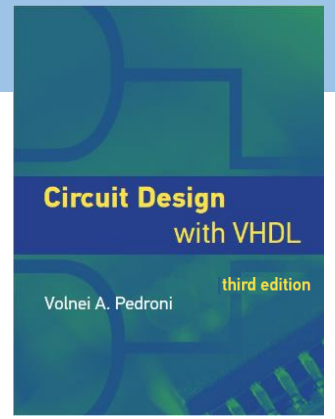
```
procedure procedure_name
   [generic (generic_list)]
   [[parameter] (input_output_list)] is
   [procedure_declarative_part]
begin
   sequential_statement_part
end [procedure] [procedure_name];
```



6. Subprogram with generics and generic subprograms

- It allows also the inclusion of subprograms in generic lists, as in the example below:

```
-------------------------------------------------------------------
entity generic_adder is
   generic (
      type data_type;
      function add (signal in1, in2: data_type) return data_type);
   port (
      ...);
end entity generic_adder;

architecture behavioral of generic_adder is
   ...
end architecture behavioral;
-------------------------------------------------------------------
```

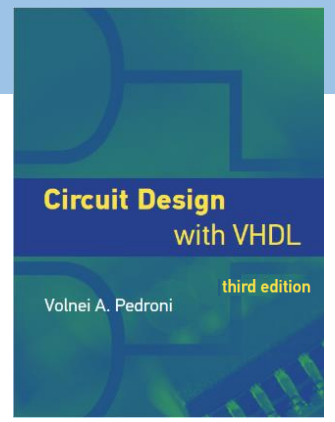


6. Subprogram with generics and generic subprograms

- It allows also the inclusion of subprograms in generic lists, as in the example below:

```
-----------------------------------------------------------------
entity generic_adder is
   generic (
      type data_type;
      function add (signal in1, in2: data_type) return data_type);
   port (
      ...);
end entity generic_adder;

architecture behavioral of generic_adder is
   ...
end architecture behavioral;
-----------------------------------------------------------------
```

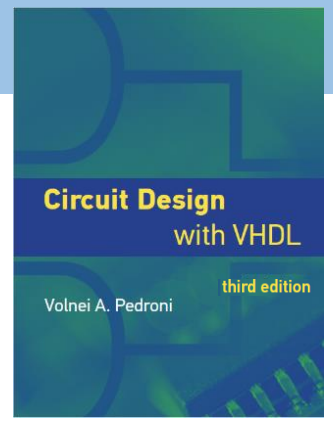
- The usage rules, with examples, are detailed in section 14.6



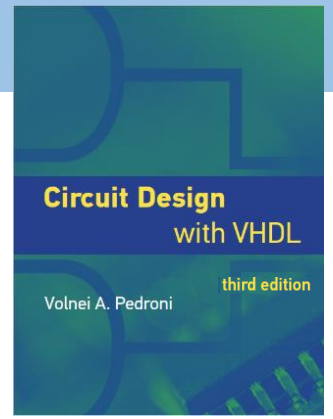
Chapter 14

Packages and Subprograms

1. Package
2. Package with generics
3. Subprograms
4. Function
5. Procedure
6. Subprogram with generics and generic subprograms
7. The *assert* statement
8. The *report* statement



7. The *assert* statement

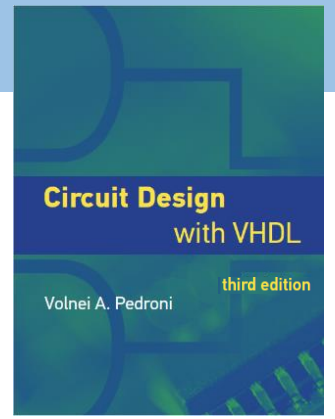


7. The *assert* statement

Syntax:

```
[label:] assert condition [report message] [severity level];
```

- It does not infer hardware
- Instead, it is used for checking the design

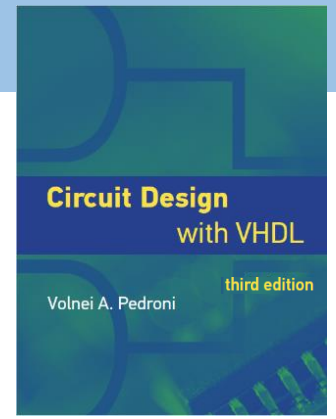


7. The *assert* statement

Syntax:

```
[label:] assert condition [report message] [severity level];
```

- It does not infer hardware
- Instead, it is used for checking the design
- Particularly helpful for:
  - Checking subprogram parameters (input and output sizes, for example)
  - Simulation results

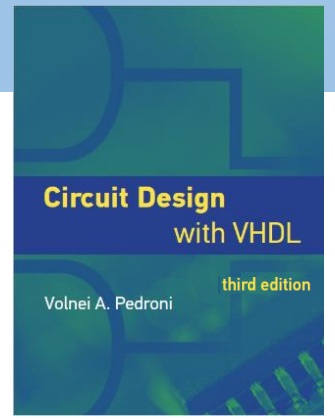


7. The *assert* statement

Syntax:

```
[label:] assert condition [report message] [severity level];
```

- It does not infer hardware
- Instead, it is used for checking the design
- Particularly helpful for:
  - Checking subprogram parameters (input and output sizes, for example)
  - Simulation results
- Reports message when the condition is false

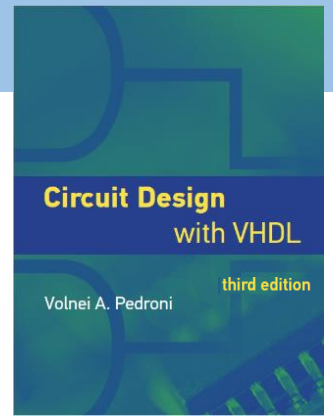


7. The *assert* statement

Syntax:

```
[label:] assert condition [report message] [severity level];
```

- It does not infer hardware
- Instead, it is used for checking the design
- Particularly helpful for:
  - Checking subprogram parameters (input and output sizes, for example)
  - Simulation results
- Reports message when the condition is false
- Severity levels: note, warning, error, failure (last two usually halt synthesis/simulation)

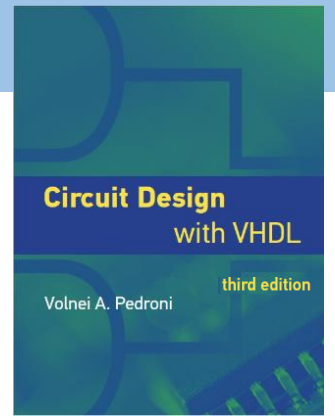


7. The *assert* statement

Syntax:

```
[label:] assert condition [report message] [severity level];
```

- The reported message can be static or dynamic
- For dynamic messages (involving time), the *to_string* function is used (details in sec. 14.8)



7. The *assert* statement

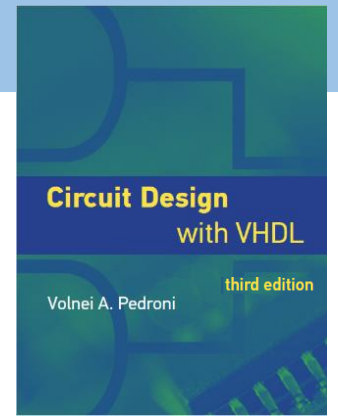
Syntax:

```
[label:] assert condition [report message] [severity level];
```

- The reported message can be static or dynamic
- For dynamic messages (involving time), the *to_string* function is used (details in sec. 14.8)

Example of static message:

```
assert address < 2**ADDRESS_WIDTH
  report "Unexpected address value."
  severity failure;
```



7. The *assert* statement

Syntax:

```
[label:] assert condition [report message] [severity level];
```

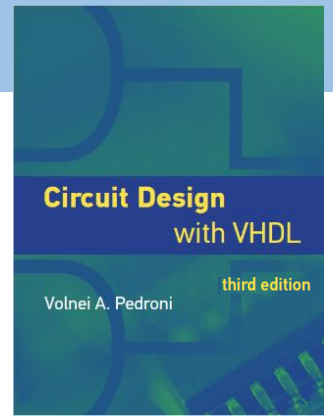
- The reported message can be static or dynamic
- For dynamic messages (involving time), the *to_string* function is used (details in sec. 14.8)

Example of static message:

```
assert address < 2**ADDRESS_WIDTH
  report "Unexpected address value."
  severity failure;
```

Example of dynamic message (see details in sec. 14.8):

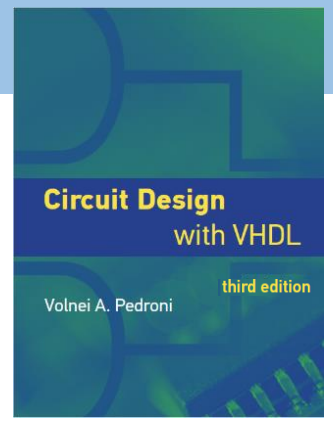
```
assert actual_val = expected_val
  report "Actual value does not match expected value at t=" & to_string(now) & "."
  severity error;
```



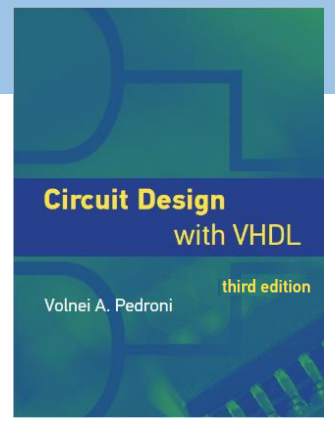
Chapter 14

Packages and Subprograms

1. Package
2. Package with generics
3. Subprograms
4. Function
5. Procedure
6. Subprogram with generics and generic subprograms
7. The *assert* statement
8. The *report* statement



8. The *report* statement



8. The *report* statement

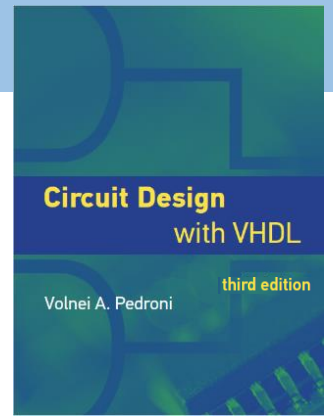
Syntax:

```
report message;
```

- Reports unconditional messages

Example:

```
report "Test was successful.";
```



End of Chapter 14