Turun yliopisto
University of Turku

HDL Based Design
ME620138

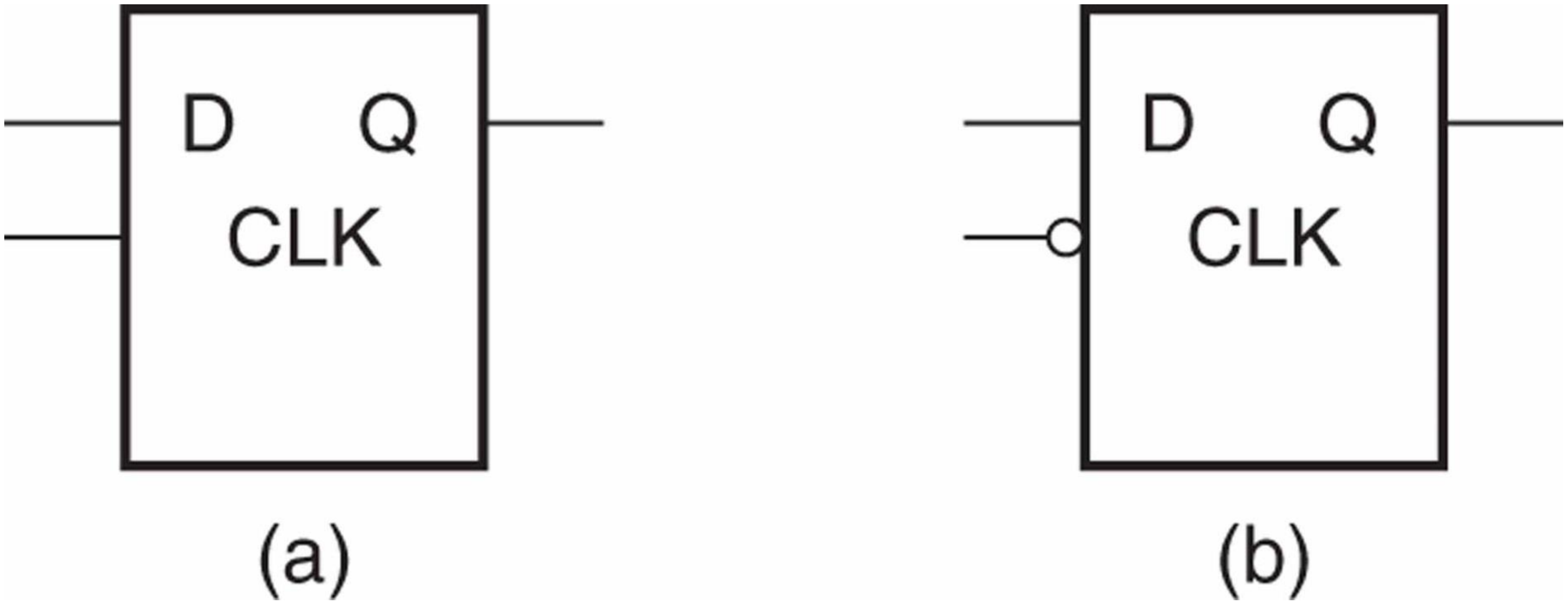
Delta Delay

- The signal value does not change as soon as the signal assignment statement is executed
- The process does not see the effect of the assignment until the next time it resumes, even if this is at the same simulation time

```
s <= '1';
...
if s then ...
```

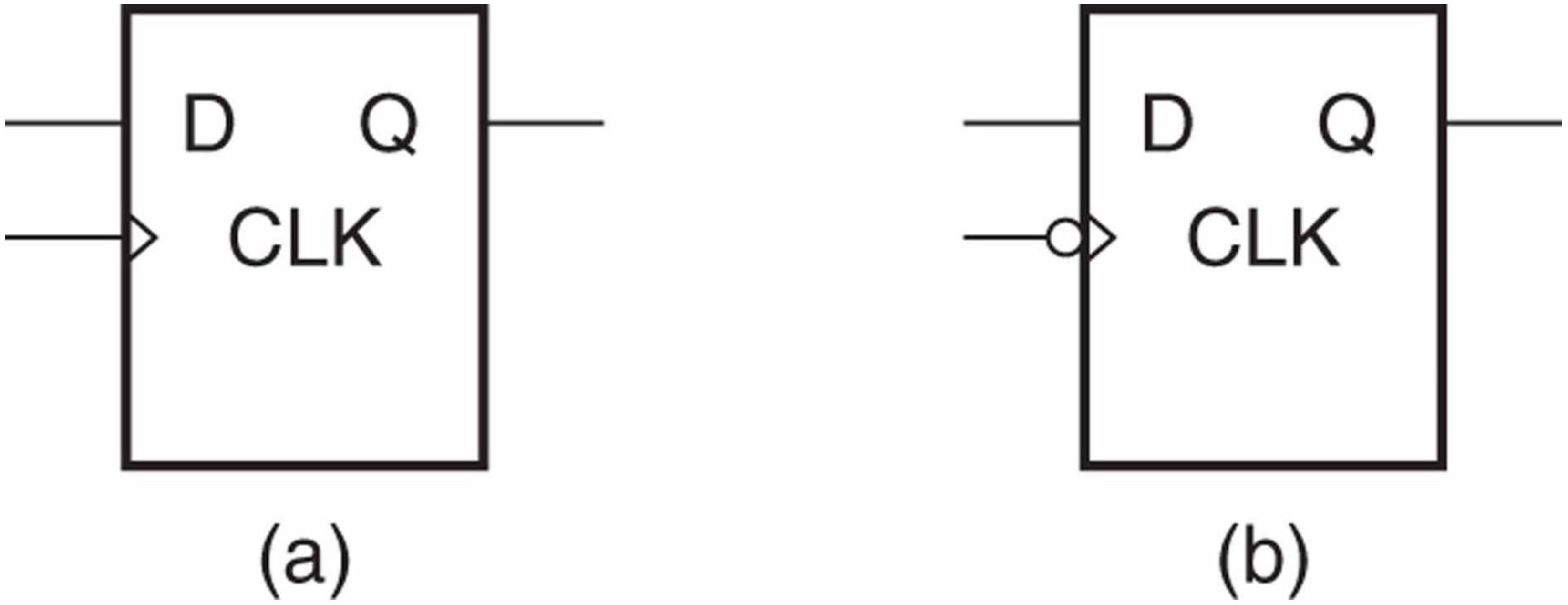
Turun yliopisto
University of Turku

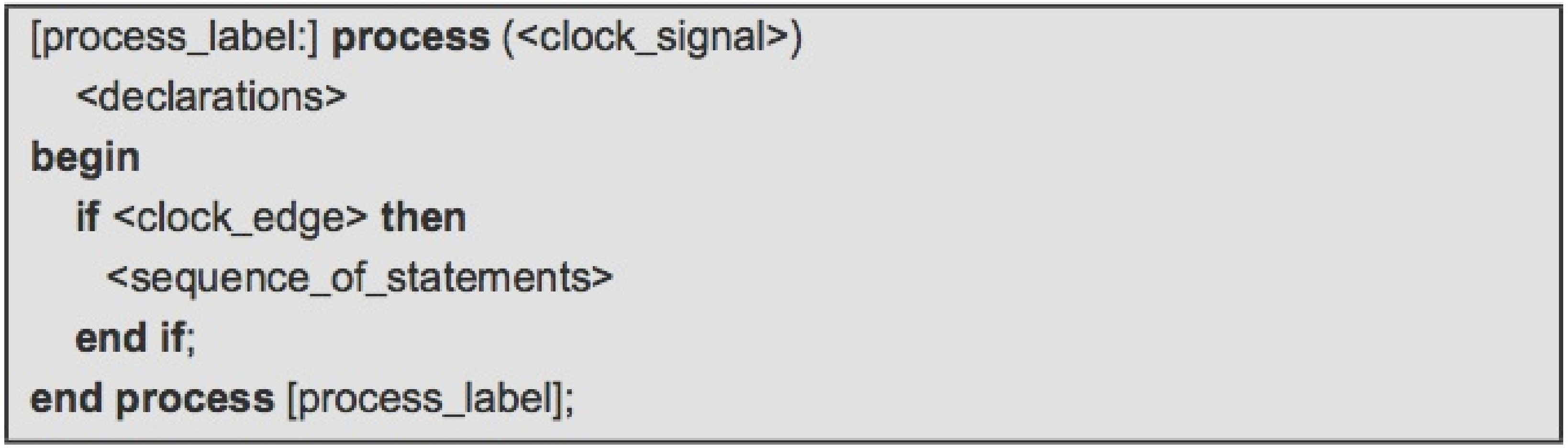
D Latch



```
[process_label:] **process** (<clock_signal>, <input_signals>)
    <declarations>
begin
    if <clock_level> then
      <sequence_of_statements>
    end if;
end process [process_label];
```

D Flip-flop





```
[process_label:] process (<clock_signal>)
   <declarations>
begin
   if <clock_edge> then
     <sequence_of_statements>
   end if;
end process [process_label];
```

Turun yliopisto
University of Turku

ARRAYS AND VECTORS

Arrays

- Arrays group elements of the same type together as a singel object
- Vectors are one-dimensional arrays

type *identifier* **is array** (type_mark **range** <>) **of** type_of_elements

For example:

type std_logic_vector **is array** (natural **range** <>) of std_logic;
signal databyte: std_logic_vector(7 downto 0);

type dbus is array (31 downto 0) of bit;
signal databus: dbus;

Multidimensional Arrays

```
type four_bits is array ( 3 downto 0 ) of std_logic;

type fourbyfour is array ( 3 downto 0 ) of four_bits;

type fourbyeight is array ( 3 downto 0, 7 downto 0 ) of std_logic;

signal nelja : four_bits;
signal neljaxnelja: fourbyfour;

nelja <= neljaxnelja(0);
```

Turun yliopisto
University of Turku

Multidimensional Arrays

```
type four_bits is array ( 3 downto 0 ) of std_logic;

signal nelja : four_bits;
signal testi : std_logic_vector (3 downto 0);

nelja <= testi;
```

```
# ** Error: Signal "testi" is type
ieee.std_logic_1164.STD_LOGIC_VECTOR;
expecting type four_bits.
```

Turun yliopisto
University of Turku

Subtype: Multidimensional Arrays

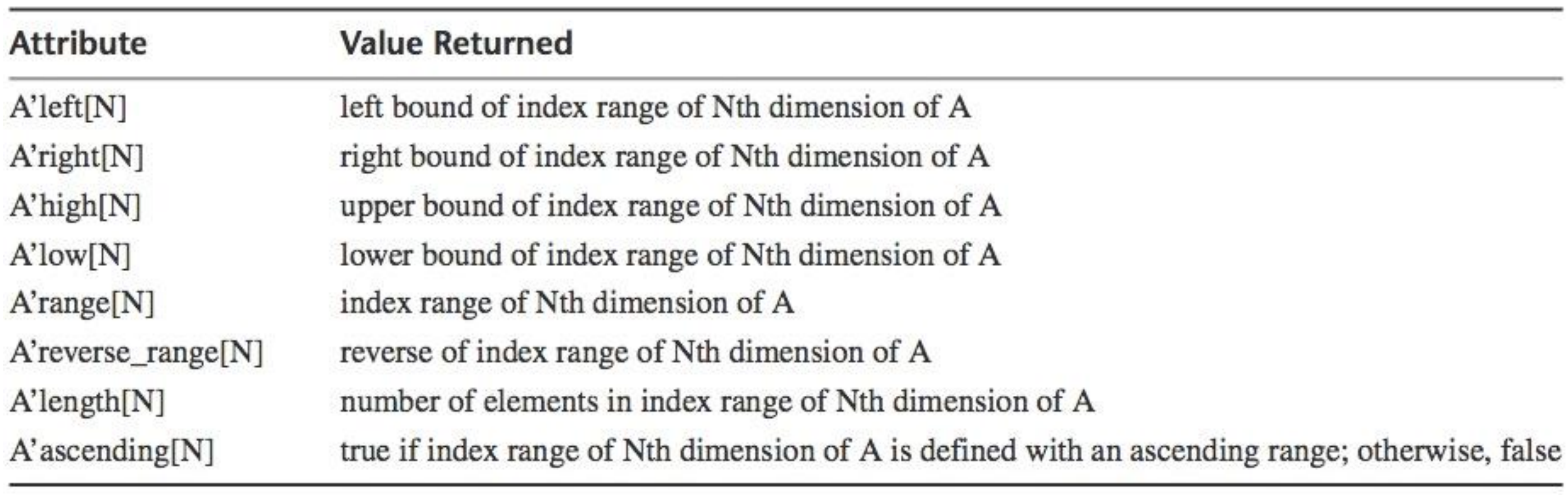
```
subtype four_bits is  std_logic_vector( 3 downto 0 );

signal nelja : four_bits;
signal testi : std_logic_vector (3 downto 0);

nelja <= testi;
```

```
# Errors: 0, Warnings: 0
```

Array Attributes



Attribute	Value Returned
A'left[N]	left bound of index range of Nth dimension of A
A'right[N]	right bound of index range of Nth dimension of A
A'high[N]	upper bound of index range of Nth dimension of A
A'low[N]	lower bound of index range of Nth dimension of A
A'range[N]	index range of Nth dimension of A
A'reverse_range[N]	reverse of index range of Nth dimension of A
A'length[N]	number of elements in index range of Nth dimension of A
A'ascending[N]	true if index range of Nth dimension of A is defined with an ascending range; otherwise, false

type A is array(1 to 10, 5 downto 1) of *some_type*;

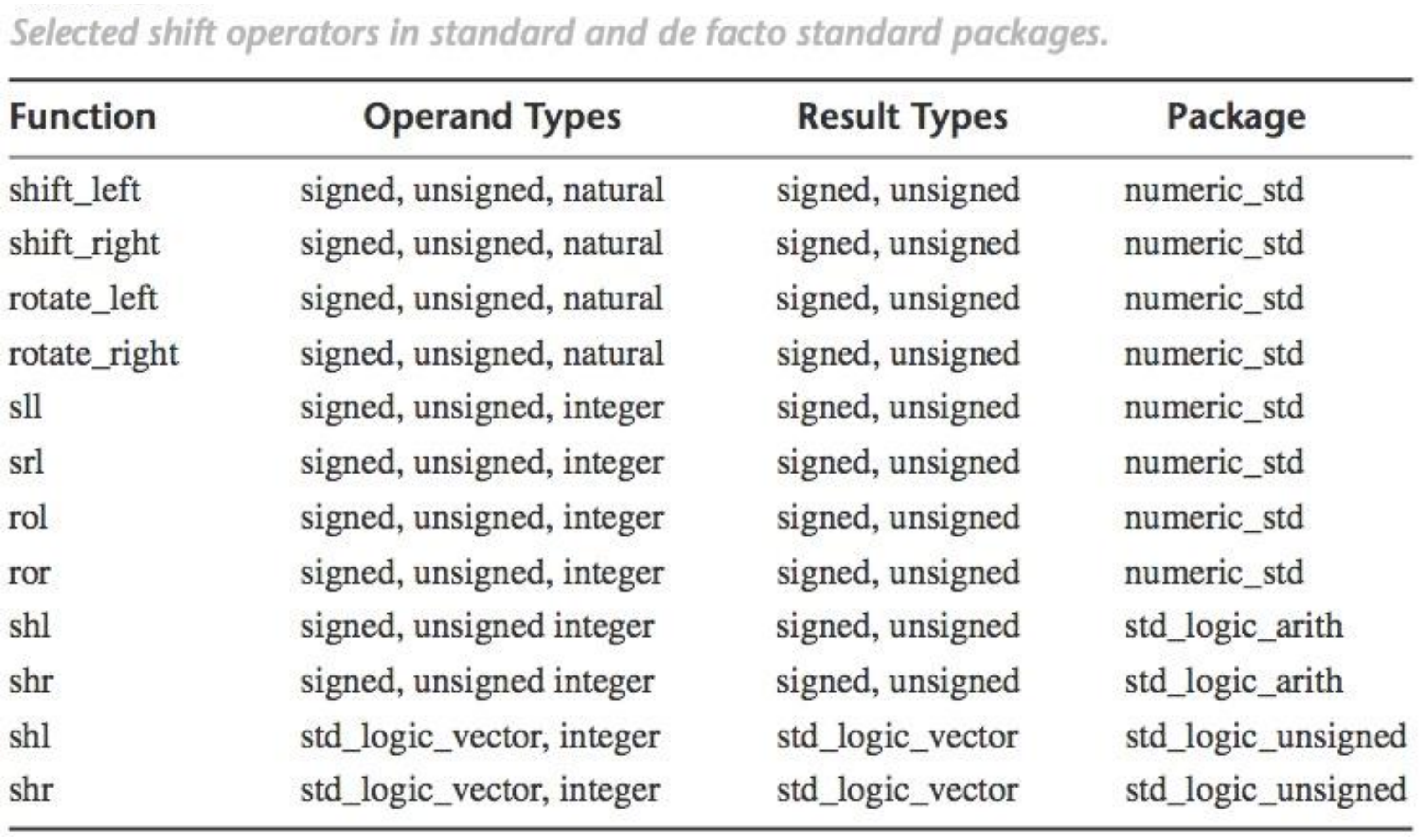
A'length(1) = 10
A'length(2) = 5
A'left(1) = 1
A'left(2) = 5

Normally you should use simple array definitions such as
array(7 downto 0, 3 downto 0)

The above array definition was for demonstration purposes

Turun yliopisto
University of Turku

Libraries and Packages



Selected shift operators in standard and de facto standard packages.

Function	Operand Types	Result Types	Package
shift_left	signed, unsigned, natural	signed, unsigned	numeric_std
shift_right	signed, unsigned, natural	signed, unsigned	numeric_std
rotate_left	signed, unsigned, natural	signed, unsigned	numeric_std
rotate_right	signed, unsigned, natural	signed, unsigned	numeric_std
sll	signed, unsigned, integer	signed, unsigned	numeric_std
srl	signed, unsigned, integer	signed, unsigned	numeric_std
rol	signed, unsigned, integer	signed, unsigned	numeric_std
ror	signed, unsigned, integer	signed, unsigned	numeric_std
shl	signed, unsigned integer	signed, unsigned	std_logic_arith
shr	signed, unsigned integer	signed, unsigned	std_logic_arith
shl	std_logic_vector, integer	std_logic_vector	std_logic_unsigned
shr	std_logic_vector, integer	std_logic_vector	std_logic_unsigned

MULTIDIMENSIONAL ARRAYS

One dimensional Array/Vector

```
signal x1: std_logic;
signal x4: std_logic_vector(3 downto 0);
signal x16: std_logic_vector(15 downto 0);
signal x32: std_logic_vector(31 downto 0);

x4(0) <= ‘1’;
-- assigns value to the rightmost element of x4
x4 <= x16(3 downto 0);
-- assigns the 4 rightmost values of x16 to x4
x4 <= ( 3 => ‘1’, 0 => ‘1’, 2 => ‘1’, 1 => ‘0’);
x4 <= ( 3|0 |2 => ‘1’, 1 => ‘0’);
x4 <= (others => ‘0’);
```

Turun yliopisto
University of Turku

Array: Examples

- Fill a vector with a constant:

```
x4 <= ( 3|0 |2 => ‘1’, 1 => ‘0’);
x16 <= (‘0’,’1’,’1’, others => ‘0’);
x16 <= (others => ‘0’);   -- set to all 0

x32(15 downto 0) <= x16(15 downto 0);
x32(31 downto 16) <= (others => x16(15)); -- sign extension!
```

- Concatenate bits and bit_vectors

```
x16 <= x10 & x32(0 to 3) & “00”;
```

Turun yliopisto
University of Turku

Multidimensional Arrays

- Example declarations:

```
type array4x1 is array(3 downto 0) of std_logic;
type array8x4 is array(7 downto 0) of array4x1;
type array6x4 is array(5 downto 0, 3 downto 0) of
std_logic;

signal mem8x4 : array8x4;
signal mem6x4 : array6x4;
signal array0, array1, array2, array3, array4, array5,
array6, array7 : array4x1;
```

Multidimensional Array

Type definitions:

```
type array8x4 is array(7 downto 0) of array4x1;
type array6x4 is array(5 downto 0, 3 downto 0) of
std_logic;
```

Single values :

```
mem8x4(0)(0) <= ’1’;
mem6x4(0,0) <= ’1’;
```

Multidimensional Array

Type definitions:

```
type array8x4 is array(7 downto 0) of array4x1;
type array6x4 is array(5 downto 0, 3 downto 0) of
std_logic;
```

Aggregates:

```
mem8x4 <= (array0, array1, … , array7);
mem6x4 <= (array0, array1, … , array5);

mem8x4 <= (others => array0);
mem8x4 <= (others => (others => ’0’));
mem6x4 <= (others => (others => ’1’));
```

Multidimensional Arrays

- In principle, it is possible to define any number of dimensions for a data type
  - Normally no more than two- or three-dimensional data types are used
- When defining multidimensional data types, array is used in the type declaration
- Example:

```
type array4x8 is array (0 to 3)
                              of std_logic_vector(7 downto 0);
type array3x4x8 is array (0 to 2) of array4x8;
```

  - Data type array4x8 is a two-dimensional data type (4 x 8 bits), while array3x4x8 is a three-dimensional data type (3 x 4 x 8 bits).

- type byte is array (7 downto 0) of std_logic;
  type data4x8 is array (integer range 0 to 3) of byte;

Array dimensions!

```
signal z_bus : std_logic_vector(3 downto 0);
signal c_bus : std_logic_vector(0 to 3);

z_bus <= c_bus;
```

- This may be correct(?), but the result is:

z_bus(3) gets the c_bus(0) value

z_bus(2) gets the c_bus(1) value

z_bus(1) gets the c_bus(2) value

z_bus(0) gets the c_bus(3) value

Turun yliopisto
University of Turku

Miscellaneous – for register transfer design

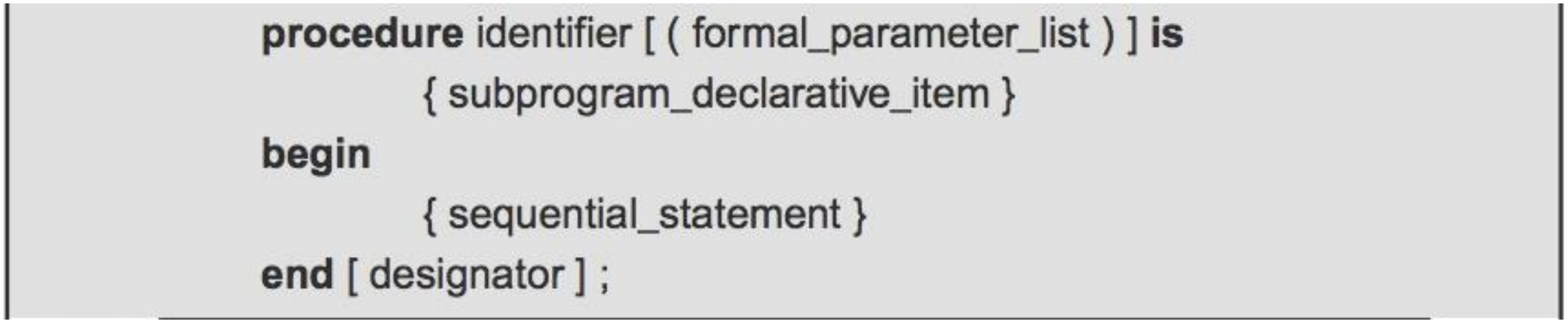
- “Alias” for existing elements

```
signal instruction: bit_vector(0 to 31);
alias opcode: bit_vector(0 to 5) is instruction(0 to 5);
alias rd: bit_vector(0 to 4) is instruction(6 to 10);
alias rs: bit_vector(0 to 4) is instruction(11 to 15);
```

SUBPROGRAMS

Procedures

- Encapsulated sequences of sequential statements



```
procedure identifier [ ( formal_parameter_list ) ] is
          { subprogram_declarative_item }
begin
          { sequential_statement }
end [ designator ] ;
```

- Procedures do not have *return* value
  - Their execution can, however, be ended by executing a **return** statement without a value
  - The *result* of computation *is passed through interface signals or variables* similarly with entity's interface list

Procedures

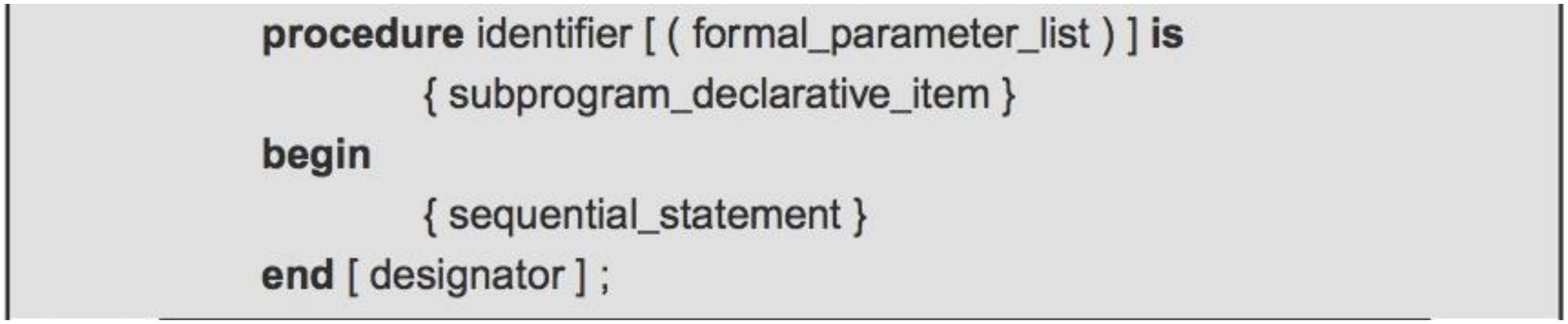
- Encapsulated sequences of sequential statements

```
procedure identifier [ ( formal_parameter_list ) ] is
          { subprogram_declarative_item }
begin
          { sequential_statement }
end [ designator ] ;
```

- Formal parameter list is similar to the entity’s interface list
  - Constants, variables and signals of mode *in, out* and *inout*
  - Mode *in* is assumed to be a *constant*, and, furthermore, *in* is the *default* mode

Procedures

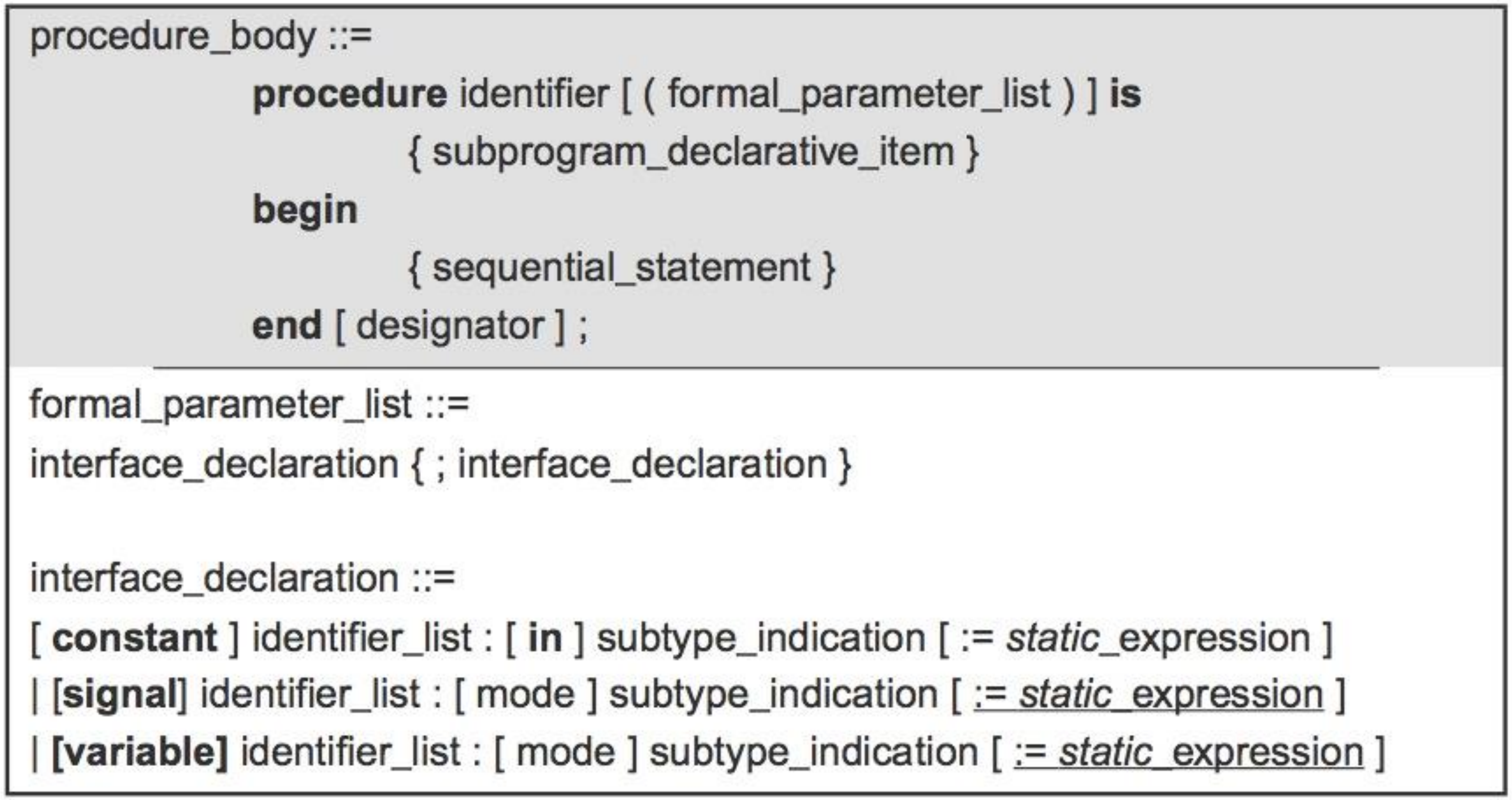
- Encapsulated sequences of sequential statements



```
procedure identifier [ ( formal_parameter_list ) ] is
        { subprogram_declarative_item }
begin
        { sequential_statement }
end [ designator ] ;
```

- Variables in procedures are created anew and initialised every time it is called
  - Note, processes’ variables are created only once
- Sequential statements are equivalent with processess’ sequential statements

Procedures



procedure_body ::=

procedure identifier [(formal_parameter_list)] **is**

{ subprogram_declarative_item }

begin

{ sequential_statement }

end [designator] ;

formal_parameter_list ::=

interface_declaration { ; interface_declaration }

interface_declaration ::=

[**constant**] identifier_list : [**in**] subtype_indication [:= *static*_expression]

| [**signal**] identifier_list : [mode] subtype_indication [:= *static*_expression]

| **[variable]** identifier_list : [mode] subtype_indication [:= *static*_expression]

Procedure

- Procedures can be defined within an architecture’s or process’s declarative part:

```
procedure Parity
   (signal X : in std_ulogic_vector;
    signal Y : out std_ulogic)
is
  variable temp : std_ulogic := '0';
begin
  for J in X'range loop
    temp := temp XOR X(J);
  end loop;
  Y <= TMP;
end Parity;
```

Turun yliopisto
University of Turku

Procedure

```
procedure Parity
   (signal X : in std_ulogic_vector;
    signal Y : out std_ulogic)
is
  variable temp : std_ulogic := '0';
begin
  for J in X'range loop
    temp := temp XOR X(J);
  end loop;
  Y <= TMP;
end Parity;
```

- Procedure is called as follows:

```
Parity(VectorIn, P);
```

- If you have a procedure without an interface list, just call it like this:

```
Procedure_name;
```

Procedure

- Procedures can be defined within an architecture’s or process’s declarative part:

```
architecture behavioural of some_entity is
        procedure Parity
           (signal X : in std_ulogic_vector;
             s
        is
          va
        be
          fo
            t
          e
          Y
        en
begin

end beh
```

```
process calculation is
        procedure Parity
           (X : in std_ulogic_vector;
            Y : out std_ulogic)
        is
          variable temp : std_ulogic := '0';
        begin
          for J in X'range loop
            temp := temp XOR X(J);
          end loop;
          Y <= TMP;
        end Parity;
begin
        . . .
end behavioural;
```

Procedure Parameters

```
procedure Parity
   (signal X : in std_ulogic_vector;
    signal Y : out std_ulogic)
is
  variable temp : std_ulogic := '0';
begin
  for J in X'range loop
    temp := temp XOR X(J);
    wait for 5 ns;
  end loop;
  Y <= TMP;
end Parity;
```

```
procedure Parity
   (  X : in std_ulogic_vector;
      Y : out std_ulogic)
is
  variable temp : std_ulogic := '0';
begin
  for J in X'range loop
    temp := temp XOR X(J);
  end loop;
  Y <= TMP;
end Parity;
```

- Signal parameters work somewhat differently from constant and variable parameters
  - If we have a wait statement within a procedure, the value of a signal might change during the waiting period

Procedures Usage

- Procedures can be used both as a sequential statement and as a concurrent statement
  - Concurrent procedure cannot have a variable in its interface list
  - Synthesisable concurrent procedures cannot have *wait* statements
  - A procedure *cannot have wait* statements if the *calling process* have an interface list
- Concurrent procedure’s process equivalent does not have an interface list and it waits the procedure’s input signals (*wait on)*

Functions

- When called, function calculates and returns a value

```
function designator [ ( formal_parameter_list ) ] return type_mark is
        { subprogram_declarative_item }
begin
        { sequential_statement }
end [ designator ] ;
```

- Function can have more than one *return* statement in its sequential statements

```
return [ expression ] ;
```

- Only *in* mode is allowed in the interface list
  - Its use is optional

Functions

- When called, function calculates and returns a value

```
function designator [ ( formal_parameter_list ) ] return type_mark is
        { subprogram_declarative_item }
begin
        { sequential_statement }
end [ designator ] ;
```

- Function can have more than one *return* statement in its sequential statements

```
return [ expression ] ;
```

- If you need to have a vector as a return type, do not define its width
    - You can also leave the width of the input away

Procedure

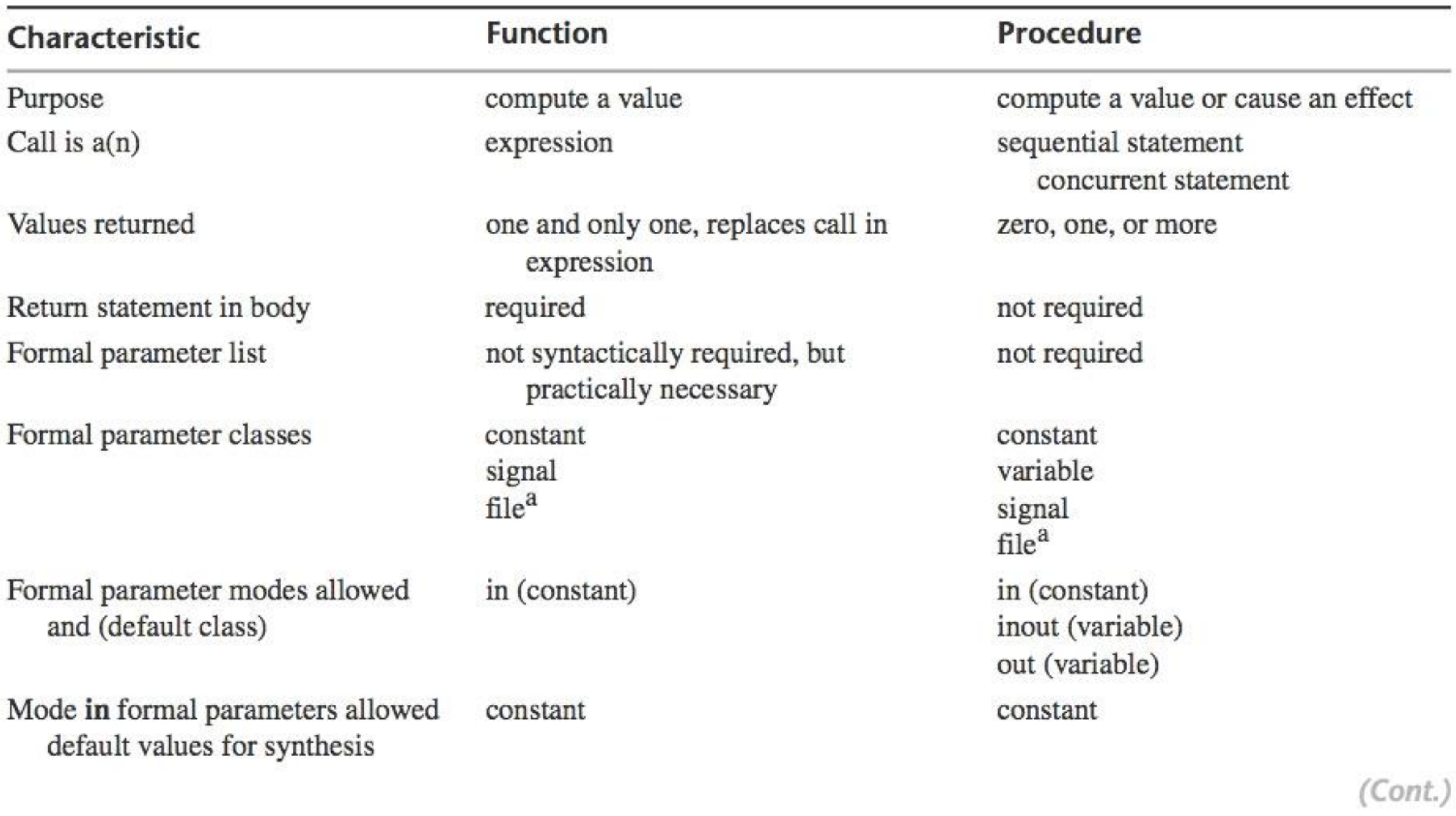
- Procedures can be defined within an architecture’s or process’s declarative part:

```
function Parity
   (signal X : std_ulogic_vector) return std_ulogic
is
  variable temp : std_ulogic := '0';
begin
  for J in X'range loop
    temp := temp XOR X(J);
  end loop;
  return temp;
end Parity;
```

- Procedure is called as follows:

```
Parity_value <= Parity(VectorIn);
```

Function versus Procedure

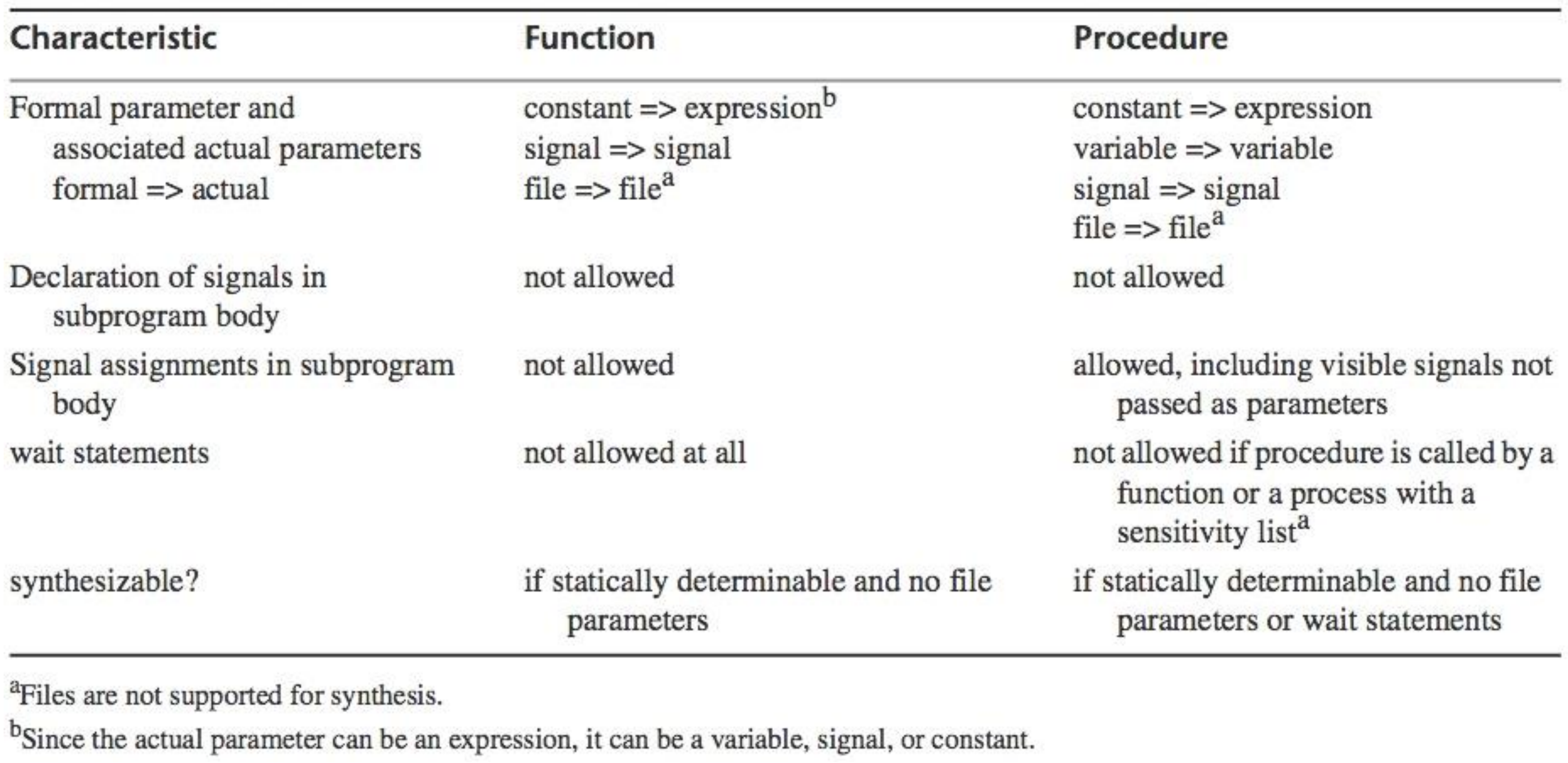


Characteristic	Function	Procedure
Purpose	compute a value	compute a value or cause an effect
Call is a(n)	expression	sequential statement concurrent statement
Values returned	one and only one, replaces call in expression	zero, one, or more
Return statement in body	required	not required
Formal parameter list	not syntactically required, but practically necessary	not required
Formal parameter classes	constant signal file[a]	constant variable signal file[a]
Formal parameter modes allowed and (default class)	in (constant)	in (constant) inout (variable) out (variable)
Mode **in** formal parameters allowed default values for synthesis	constant	constant

(Cont.)

Turun yliopisto
University of Turku

Function versus Procedure



Characteristic	Function	Procedure

Formal parameter and associated actual parameters formal => actual	constant => expression[b]
	signal => signal
	file => file[a]
		constant => expression
		variable => variable
		signal => signal
		file => file[a]

Declaration of signals in subprogram body	not allowed	not allowed
Signal assignments in subprogram body	not allowed	allowed, including visible signals not passed as parameters
wait statements	not allowed at all	not allowed if procedure is called by a function or a process with a sensitivity list[a]
synthesizable?	if statically determinable and no file parameters	if statically determinable and no file parameters or wait statements

[a]Files are not supported for synthesis.

[b]Since the actual parameter can be an expression, it can be a variable, signal, or constant.

Subprogram Overloading

- The naming of subprograms should indicate their purpose and make our models more readable for other designers
- What to do if we might want to use the same name for some other subprogram?

We need to implement the same function with different input variables

Subprogram Overloading

- Excerpt of the NUMERIC_STD package

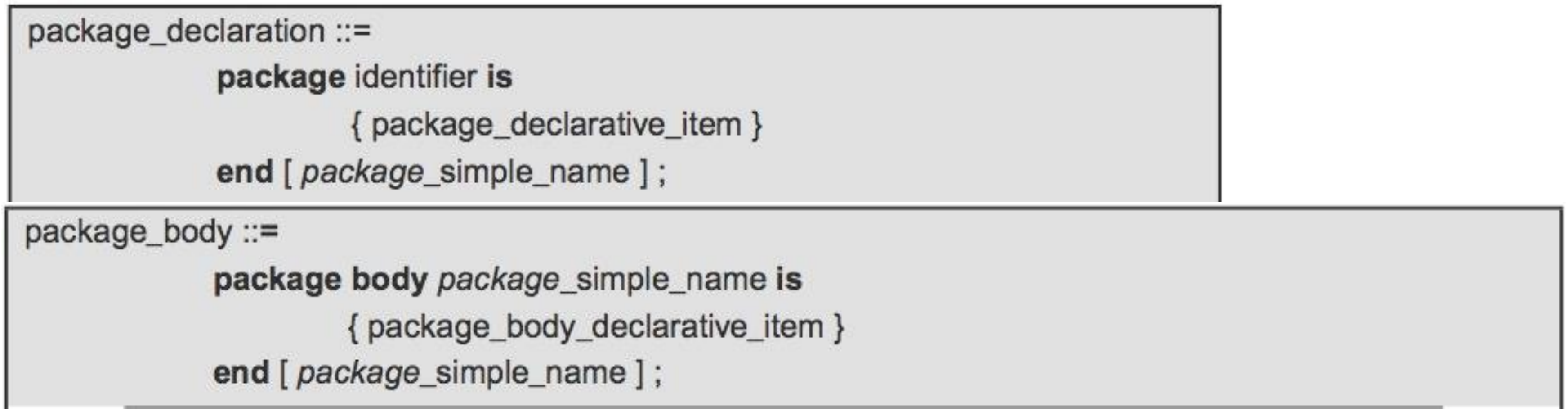
```
type UNRESOLVED_UNSIGNED is array (NATURAL range <>) of STD_ULOGIC;
type UNRESOLVED_SIGNED is array (NATURAL range <>) of STD_ULOGIC;
alias U_UNSIGNED is UNRESOLVED_UNSIGNED;
alias U_SIGNED is UNRESOLVED_SIGNED;

function "*" (L : UNRESOLVED_UNSIGNED; R : NATURAL)
                return UNRESOLVED_UNSIGNED is
begin
        return L * TO_UNSIGNED(R, L'length);
end function "*";

function "*" (L : UNRESOLVED_SIGNED; R : INTEGER)
        return UNRESOLVED_SIGNED is
begin
    return L * TO_SIGNED(R, L'length);
end function "*";
```

VHDL Package

- Package = file of type definitions, functions, procedures to be shared across VHDL models
  - User created
  - Standard lib's/3rd party – usually distributed in "libraries"

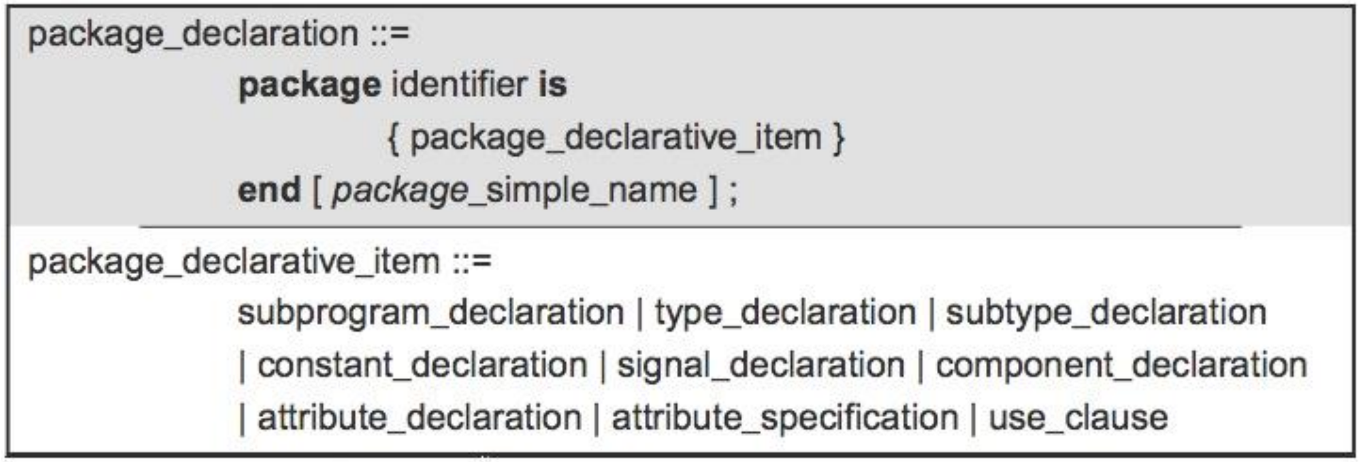


package_declaration ::=
    package identifier **is**
        { package_declarative_item }
    end [*package*_simple_name] ;

package_body ::=
    package body *package*_simple_name **is**
        { package_body_declarative_item }
    end [*package*_simple_name] ;

Turun yliopisto
University of Turku

VHDL Package



package_declaration ::=

package identifier **is**

{ package_declarative_item }

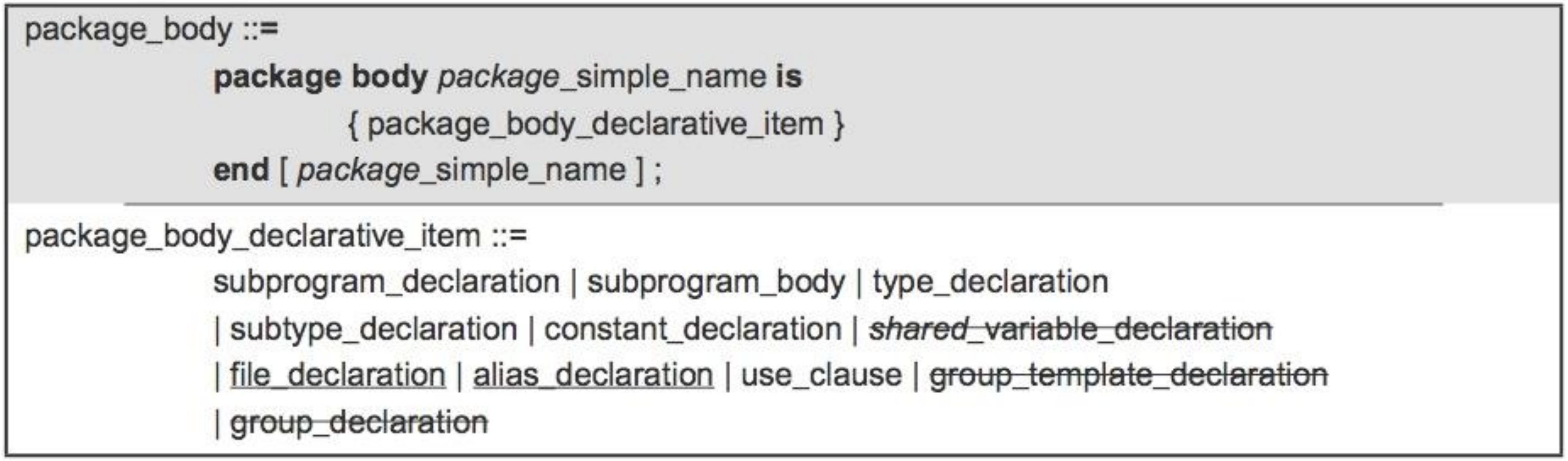
end [*package*_simple_name] ;

package_declarative_item ::=

subprogram_declaration | type_declaration | subtype_declaration

| constant_declaration | signal_declaration | component_declaration

| attribute_declaration | attribute_specification | use_clause



package_body ::=

package body *package*_simple_name **is**

{ package_body_declarative_item }

end [*package*_simple_name] ;

package_body_declarative_item ::=

subprogram_declaration | subprogram_body | type_declaration

| subtype_declaration | constant_declaration | ~~*shared*_variable_declaration~~

| file_declaration | alias_declaration | use_clause | ~~group_template_declaration~~

| ~~group_declaration~~

Turun yliopisto
University of Turku

VHDL Package: Example

```
PACKAGE my_stuff IS
    TYPE binary IS ( ON, OFF );
    CONSTANT PI : REAL := 3.14;
    CONSTANT My_ID : INTEGER;
    PROCEDURE add_bits3(SIGNAL a, b, en : IN BIT;
                 SIGNAL temp_result, temp_carry : OUT BIT);
END my_stuff;
```

```
PACKAGE BODY my_stuff IS
   CONSTANT My_ID : INTEGER := 2;

   PROCEDURE add_bits3(SIGNAL a, b, en : IN BIT;
            SIGNAL temp_result, temp_carry : OUT BIT) IS
   BEGIN    -- this function can return a carry
      temp_result <= (a XOR b) AND en;
      temp_carry <= a AND b AND en;
   END add_bits3;
END my_stuff;
```

Turun yliopisto
University of Turku

Library

- LIBRARY is a collection of packages

- The standard VHDL libraries are *std* and *ieee*
  - The working library *work*, where we store our currently analysed design entities

- To make a package visible to the design, two declarations are needed.
  LIBRARY library_name;
  USE library_name.package_name.all;

Turun yliopisto
University of Turku

WE'RE GOING TO TRY SOMETHING CALLED AGILE PROGRAMMING.
scottadams@aol.com
www.dilbert.com
THAT MEANS NO MORE PLANNING AND NO MORE DOCUMENTATION. JUST START WRITING CODE AND COMPLAINING.
11-26-07 ©2007 Scott Adams, Inc./Dist. by UFS, Inc.
I'M GLAD IT HAS A NAME.
THAT WAS YOUR TRAINING.