UNIVERSITÀ
DEGLI STUDI DI TRIESTE

dipartimento
di ingegneria
e architettura

08 – Packages and use clauses

A.Carini – Progettazione di sistemi elettronici

VHDL Packages

- They are simply a way to collect a group of declarations that are related to each other and are useful for a common task.
- They are separate design units.
- They divide the external vision, the package declaration, from the corresponding implementation, the package body.

Package declaration

```
package_declaration ⇐
    package identifier is
        { package_declarative_item }
    end [ package ] [ identifier ] ;
```

- We can declare types, subtypes, subprograms, etc.

Example

```
package cpu_types is
    constant word_size : positive := 16;
    constant address_size : positive := 24;
    subtype word is bit_vector(word_size – 1 downto 0);
    subtype address is bit_vector(address_size – 1 downto 0);
    type status_value is ( halted, idle, fetch, mem_read, mem_write,
                           io_read, io_write, int_ack );
end package cpu_types;
```

Package: a design unit

- The package is a separate design unit.
- The analyzer separately analyzes it, and places it in the work library as a library unit.
- Its elements can be called with the *selected name*: composed by the library name, dot, the package name, dot, the element name.

```
work.cpu_types.status_value
```

Example

```
entity address_decoder is
      port ( addr : in work.cpu_types.address;
             status : in work.cpu_types.status_value;
             mem_sel, int_sel, io_sel : out bit );
end entity address_decoder;
-------------------------------------------------------------------
architecture functional of address_decoder is
      constant mem_low : work.cpu_types.address := X"000000";
      constant mem_high : work.cpu_types.address := X"EFFFFF";
      constant io_low : work.cpu_types.address := X"F00000";
      constant io_high : work.cpu_types.address := X"FFFFFF";
begin
      mem_decoder :
            mem_sel <= '1' when ( work.cpu_types."="(status, work.cpu_types.fetch)
                                  or work.cpu_types."="(status, work.cpu_types.mem_read)
                                  or work.cpu_types."="(status, work.cpu_types.mem_write) )
                                  and addr >= mem_low and addr <= mem_high else
                       '0';
      int_decoder :
            int_sel <= '1' when work.cpu_types."="(status, work.cpu_types.int_ack) else
                       '0';
      io_decoder :
            io_sel <= '1' when ( work.cpu_types."="(status, work.cpu_types.io_read)
                                  or work.cpu_types."="(status, work.cpu_types.io_write) )
                                 and addr >= io_low and addr <= io_high else
                      '0';
end architecture functional;
```

Packages and Libraries

- We could place the package in a resource library and then call its elements using the name of that library:

```
variable stored_state : ieee.std_logic_1164.std_ulogic;
```

Packages and signals

```
library ieee;  use ieee.std_logic_1164.all;
package clock_pkg is
        constant Tpw : delay_length := 4 ns;
        signal clock_phase1, clock_phase2 : std_ulogic;
end package clock_pkg;
```

Packages and signals

```
library ieee;  use ieee.std_logic_1164.all;
entity io_controller is
      port ( ref_clock : in std_ulogic;  ... );
end entity io_controller;
-------------------------------------------------------------
architecture top_level of io_controller is
      ...
begin
      internal_clock_gen : entity work.phase_locked_clock_gen(std_cell)
            port map ( reference => ref_clock,
                       phi1 => work.clock_pkg.clock_phase1,
                       phi2 => work.clock_pkg.clock_phase2 );
      the_bus_sequencer : entity work.bus_sequencer(fsm)
            port map ( rd, wr, sel, width, burst, addr(1 downto 0), ready,
                       control_reg_wr, status_reg_rd, data_fifo_wr, data_fifo_rd,
                       ... );
      ...
end architecture top_level;
```

Packages and signals

```
architecture fsm of bus_sequencer is

  -- This architecture implements the sequencer as
  -- a finite-state machine. NOTE: it uses the clock signals
  -- from clock_pkg to synchronize the fsm.

  signal next_state_vector : ...;

begin

  bus_sequencer_state_register :
    entity work.state_register(std_cell)
      port map ( phi1 => work.clock_pkg.clock_phase1,
                 phi2 => work.clock_pkg.clock_phase2,
                 next_state => next_state_vector,
                 ... );

  ...

end architecture fsm;
```

Packages and subprograms

- In the *package declaration* we must specify only the *header*, i.e., the name and the *interface list*.
- This is an example of *information hiding*.
- Example:

```
subtype word32 is bit_vector(31 downto 0);

procedure add ( a, b : in word32;
                result : out word32;  overflow : out boolean );

function "<" ( a, b : in word32 ) return boolean;
```

Packages and constants

- We can still apply the information hiding principle:

```
constant max_buffer_size : positive;
```

- And in the package body we have:

```
constant max_buffer_size : positive := 4096;
```

- Or we can fully declare our constants!

Example

```
package cpu_types is
    constant word_size : positive := 16;
    constant address_size : positive := 24;
    subtype word is bit_vector(word_size – 1 downto 0);
    subtype address is bit_vector(address_size – 1 downto 0);
    type status_value is ( halted, idle, fetch, mem_read, mem_write,
                           io_read, io_write, int_ack );
    subtype opcode is bit_vector(5 downto 0);
    function extract_opcode ( instr_word : word ) return opcode;
    constant op_nop : opcode := "000000";
    constant op_breq : opcode := "000001";
    constant op_brne : opcode := "000010";
    constant op_add : opcode := "000011";
    ...
end package cpu_types;
```

Example

```
architecture behavioral of cpu is
begin
    interpreter : process is
        variable instr_reg : work.cpu_types.word;
        variable instr_opcode : work.cpu_types.opcode;
    begin
        ...      -- initialize
        loop
            ...      -- fetch instruction
            instr_opcode := work.cpu_types.extract_opcode ( instr_reg );
            case instr_opcode is
                when work.cpu_types.op_nop => null;
                when work.cpu_types.op_breq => ...
                ...
            end case;
        end loop;
    end process interpreter;
end architecture behavioral;
```

Package body

- Every *package* with subprograms or deferred constants must have a *package body* that provides the missing information.

```
package_body ⇐
    package body identifier is
        { package_body_declarative_item }
    end [ package body ] [ identifier ] ;
```

Package body

- The subprograms of the *package body* must repeat exactly the same *header* of the package declaration and must have also a body.
- Idem for constants, that must have also an initialization expression.
- In the package body we can define also other types, subtypes, constants, subprograms.
- In the *package body* we cannot define signals.

Example

```
package bit_vector_signed_arithmetic is
    function "+" ( bv1, bv2 : bit_vector ) return bit_vector;
    function "–" ( bv : bit_vector ) return bit_vector;
    function "*" ( bv1, bv2 : bit_vector ) return bit_vector;
    ...
end package bit_vector_signed_arithmetic;
-------------------------------------------------------------------
package body bit_vector_signed_arithmetic is
    function "+" ( bv1, bv2 : bit_vector ) return bit_vector is ...
    function "–" ( bv : bit_vector ) return bit_vector is ...
    function mult_unsigned ( bv1, bv2 : bit_vector ) return bit_vector is
        ...
    begin
        ...
    end function mult_unsigned;
    function "*" ( bv1, bv2 : bit_vector ) return bit_vector is
    begin
        if bv1(bv1'left) = '0' and bv2(bv2'left) = '0' then
            return mult_unsigned(bv1, bv2);
        elsif bv1(bv1'left) = '0' and bv2(bv2'left) = '1' then
            return –mult_unsigned(bv1, –bv2);
        elsif bv1(bv1'left) = '1' and bv2(bv2'left) = '0' then
            return –mult_unsigned(–bv1, bv2);
        else
            return mult_unsigned(–bv1, –bv2);
        end if;
    end function "*";
    ...
end package body bit_vector_signed_arithmetic;
```

Use clause

- Applicable also to packages, because *library units:*

```
use work.cpu_types;
```

```
variable data_word : cpu_types.word;
variable next_address : cpu_types.address;
```

Use clause syntax

```
use_clause ⇐ use selected_name { , ... } ;
selected_name ⇐ name . ( identifier | character_literal | operator_symbol | all )
```

- The name can be an identifier or a *selected name* :

```
identifier . identifier . ( identifier | character_literal | operator_symbol | all )
```

- The first identifier can be the library, the second the package, and the third a package element :

```
use work.cpu_types.word, work.cpu_types.address;
variable data_word : word;
variable next_address : address;
```

Example

```
architecture behavioral of cpu is
begin
    interpreter : process is
        use work.cpu_types.all;
        variable instr_reg : word;
        variable instr_opcode : opcode;
    begin
        ...      -- initialize
        loop
            ...      -- fetch instruction
            instr_opcode := extract_opcode ( instr_reg );
            case instr_opcode is
                when op_nop => null;
                when op_breq => ...
                ...
            end case;
        end loop;
    end process interpreter;
end architecture behavioral;
```

Context clause

- The use clause can be written both at the beginning of the design unit or in any other declarative part.
- If written at the beginning of the design unit, it is visible everywhere in that unit.
- This area is called *context clause*.

```
library ieee;  use ieee.std_logic_1164.std_ulogic;
entity logic_block is
      port ( a, b : in std_ulogic;
                y, z : out std_ulogic );
end entity logic_block;
```

Predefined package standard

- All predefined types and operators are part of a package **standard** located in a special library **std**.
- Since we have always to refer to them, there is an implicit context clause:

```
library std, work;  use std.standard.all;
```

- If we need to refer to them when they are hided we can use the selected name:

```
result := std.standard."<" ( a, b );
```

Example

```
function "<" ( a, b : bit_vector ) return boolean is
  variable tmp1 : bit_vector(a'range) := a;
  variable tmp2 : bit_vector(b'range) := b;
begin
  tmp1(tmp1'left) := not tmp1(tmp1'left);
  tmp2(tmp2'left) := not tmp2(tmp2'left);
  return std.standard."<" ( tmp1, tmp2 );
end function "<";
```

See:

- Peter Ashenden, «The designers' guide to VHDL» Morgan Kaufmann,
  - Chapter 7