UNIVERSITÀ
DEGLI STUDI DI TRIESTE

dipartimento
di ingegneria
e architettura

11 – Aliases

A.Carini – Progettazione di sistemi elettronici

Aliases for data objects

- We can declare aliases for objects like variables, constants, signals:

alias_declaration ⇐ **alias** identifier **is** name ;

- The alias provides an alternative name for the named object.
- We treat the alias as an object of the type specified in the declaration of the original object.
- Every action performed on the alias is directly applied to the original object.

Example

- Assume you have two constants with the same name in two different packages:

```
library ieee;  use ieee.std_logic_1164.all;
use work.alu_types.all, work.io_types.all;
architecture structural of controller_system is
      alias alu_data_width is work.alu_types.data_width;
      alias io_data_width is work.io_types.data_width;
      signal alu_in1, alu_in2,
                  alu_result : std_logic_vector(0 to alu_data_width – 1);
      signal  io_data : std_logic_vector(0 to io_data_width – 1);
      ...
begin
      ...
end architecture structural;
```

Notes on aliases

- An alias can be associated with a single element of a composite object (a record or an array).
- We can consider also an alias of an alias.
- An alias can also be associated with a slice of a one-dimensional array.

Example

```
type register_array is array (0 to 15) of bit_vector(31 downto 0);
type register_set is record
        general_purpose_registers : register_array;
        program_counter : bit_vector(31 downto 0);
        program_status : bit_vector(31 downto 0);
    end record;
variable CPU_registers : register_set;
```

```
alias PSW is CPU_registers.program_status;
alias PC is CPU_registers.program_counter;
alias GPR is CPU_registers.general_purpose_registers;
```

Example

```
alias SP is CPU_registers.general_purpose_registers(15);
```

```
alias SP is GPR(15);
```

```
alias interrupt_level is PSW(30 downto 26);
```

Extended form of alias

- In the previous example, *interrupt_level* is an array 30 downto 26.
- It would be more useful to refer to *interrupt_level* as an array 4 downto 0.
- We can do that with the extended form of alias declaration:

alias_declaration ⇐ **alias** identifier [: subtype_indication] **is** name ;

- We can include the subtype indication is the alias of scalar objects, but the declared range must the same of the original object (useful only for documentation).
- We can include an array unconstrained subtype in the alias of an array. The index range, both for bounds and direction, is inherited from the original object (useful only for documenting the element type).

Extended form of alias

- When aliasing an array or an array slice, we can indicate different index bounds or directions, but the basic type and the number of elements shall not change.
- The association is done from the left to the right.

```
alias interrupt_level : bit_vector(4 downto 0) is PSW(30 downto 26);
```

- *Interrupt_level(4)* is *PSW(30)*
- *Interrupt_level(3)* is *PSW(29)*

Example

```
function "+" ( bv1, bv2 : bit_vector ) return bit_vector is
    alias norm1 : bit_vector(1 to bv1'length) is bv1;
    alias norm2 : bit_vector(1 to bv2'length) is bv2;
    variable result : bit_vector(1 to bv1'length);
    variable carry : bit := '0';
begin
    if bv1'length /= bv2'length then
        report "arguments of different length" severity failure;
    else
        for index in norm1'reverse_range loop
            result(index) := norm1(index) xor norm2(index) xor carry;
            carry := ( norm1(index) and norm2(index) )
                     or ( carry and ( norm1(index) or norm2(index) ) );
        end loop;
    end if;
    return result;
end function "+";
```

Aliases for non data items

- We can also define aliases for types, subprograms, packages, entities, etc.
- The only elements that cannot be aliased are labels and loop indexes.
- The syntax to be used in this case is:

```
alias_declaration ⇐
    alias ⟨ identifier ▯ character_literal ▯ operator_symbol ⟩
        is name ⟦ signature ⟧ ;
```

Aliases for types

- The declared alias may be used in all contexts where the original type was used.
- The predefined operators for the original type are also applicable to the alias (we do not need to redefine them).
- The alias simply defines another name for that type.

```
alias binary_string is bit_vector;
```

```
variable s1, s2 : binary_string(0 to 7);
...
s1 := s1 and not s2;
```

Aliases for enumeration types

- All the enumeration literals defined by the type are directly visible (we do not need to define aliases for the enumeration literals, nor we need to use the selected names).
- Assume to have in a package system_types

```
type system_status is (idle, active, overloaded);

alias status_type is work.system_types.system_status;
```

- You can refer to idle, active, overloaded instead of work.system_types.idle, etc.

Signature in aliases

- It is used only for the aliases of subprograms and enumeration literals.
- These could be overloaded and the name could be insufficient to univocally identify the element.
- The *signature* is used for this purpose.
- Syntax:

signature ⇐ [[type_mark { , ... }] [**return** type_mark]]

Signature in aliases

- In case of subprograms, the signature lists the type of the parameters in the same order of the subprogram declaration.
- If in a package *arithmetic_ops* we have:

```
procedure increment ( bv : inout bit_vector;  by : in integer := 1 );
procedure increment ( int : inout integer;  by : in integer := 1 );
```

we can declare aliases for the procedures as follows:

```
alias bv_increment is work.arithmetic_ops.increment [ bit_vector, integer ];
alias int_increment is work.arithmetic_ops.increment [ integer, integer ];
```

Signature in aliases

- In the alias of a function, the signature must include also the type of the returned value after the keyword ***return***.

```
alias "*" is "and" [ bit, bit return bit ];
alias "+" is "or" [ bit, bit return bit ];
alias "–" is "not" [ bit return bit ];
```

```
s <= a * b + (–a) * c;
```

Signature in aliases

- We must take into account the possibility of an overloading also when aliasing enumeration literals.
- In VHDL, an enumeration literal is equivalent to a function without parameters that returns a value of the enumeration type.

```
alias high is std.standard.'1' [ return bit ];
```

- Be careful: for character literals we must use the selected name!

A package of elements defined in other packages

```
package DMA_controller_types_and_utilities is
    alias word is work.cpu_types.word;
    alias status_value is work.cpu_types.status_value;
    alias "+" is work.bit_vector_unsigned_arithmetic."+"
                        [ bit_vector, bit_vector return bit_vector ];
    ...
end package DMA_controller_types_and_utilities;
```

See:

- Peter Ashenden, «The designers' guide to VHDL» Morgan Kaufmann,
  - Chapter 11