UNIVERSITÀ DEGLI STUDI DI TRIESTE

dipartimento di ingegneria e architettura

09 – Resolved signals

A.Carini – Progettazione di sistemi elettronici

Resolved signals

- The VHDL language requires the designer to specify precisely which values are obtained from the interconnection of multiple outputs (i.e., drivers) on the same signal.
- This is done with the *resolved signals*, which are an extension of the normal signals studied in the previous lessons.
- The resolved signals include in their definition a function, called *resolution function*, that is used for computing the final value of the signal from the contributions of all drivers.

Example: tri-state logic

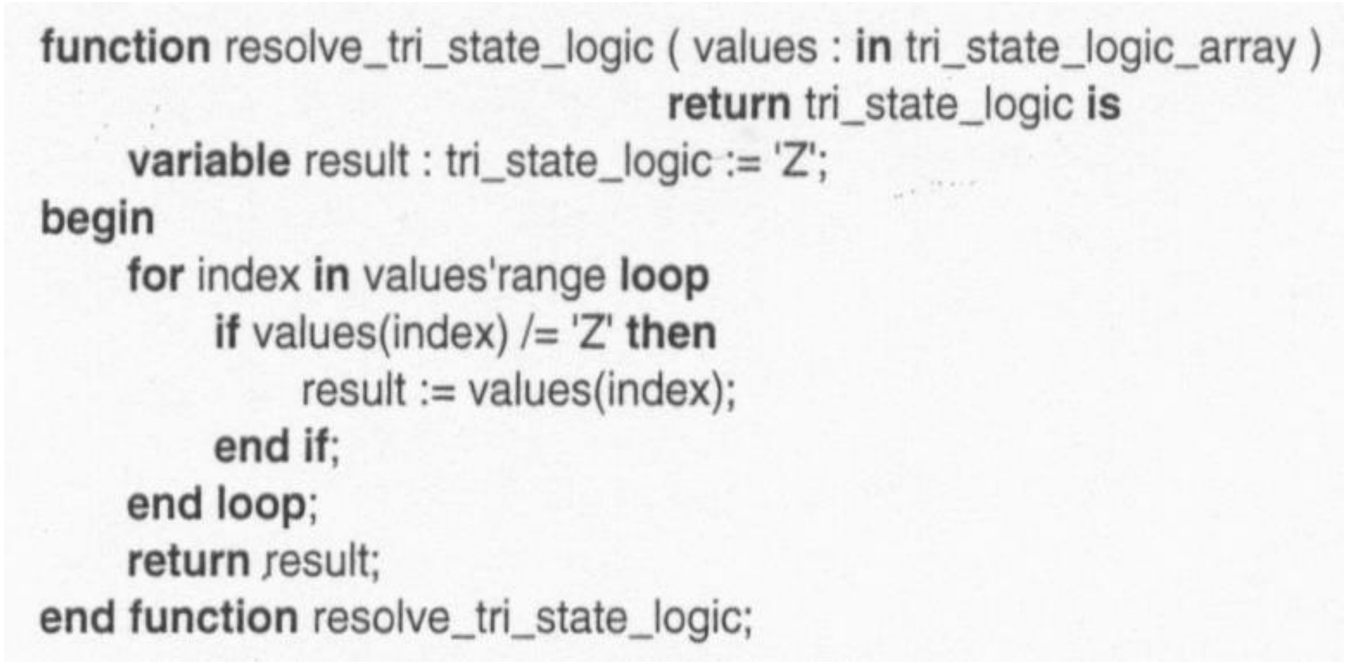
- We want to write a resolved signal of the type:

```
type tri_state_logic is ('0', '1', 'Z');
```

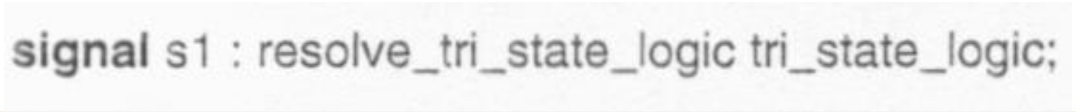
- The resolution function must be capable to accept any number of contributions of type *tri_state_logic* and must return a *tri_state_logic*.
- We write a resolution function that has a single parameter, which is an unconstrained array of elements *tri_state_logic.*

```
type tri_state_logic_array is array (integer range <>) of tri_state_logic;
```

Example: tri-state logic

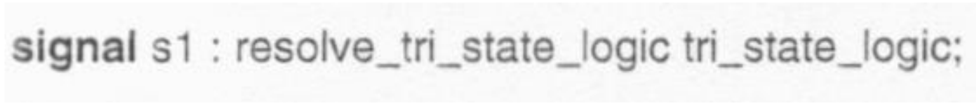


```
function resolve_tri_state_logic ( values : in tri_state_logic_array )
                                        return tri_state_logic is
    variable result : tri_state_logic := 'Z';
begin
    for index in values'range loop
        if values(index) /= 'Z' then
            result := values(index);
        end if;
    end loop;
    return result;
end function resolve_tri_state_logic;
```



```
signal s1 : resolve_tri_state_logic tri_state_logic;
```

Resolved signal



```
signal s1 : resolve_tri_state_logic tri_state_logic;
```

- The resolution function indicates that the signal is a *resolved signal*, with the resolution function there given.
- Since the signal is a resolved signal, it can have different drivers.
- When a transaction for the signal becomes active, the novel value is not applied directly to the signal.
- On the contrary, the contributions of all sources connected to the signal are grouped together in an array, which is passed to the resolution function.
- The result of the resolution function is the new value applied to the signal.

Resolved signal

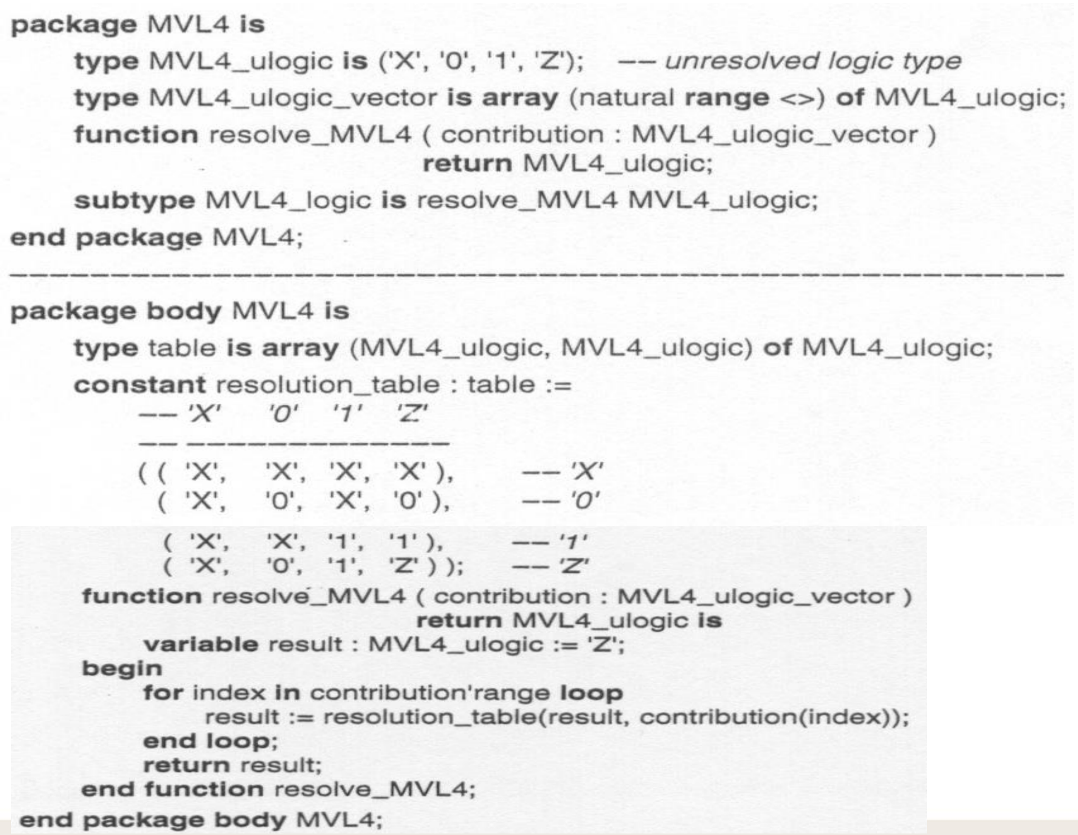
- The syntax behind the *resolve signals:*

```
subtype_indication ⇐
    ⟦ resolution_function_name ⟧
    type_mark  ⟦ range ⦅ range_attribute_name

   | simple_expression ⦅ to | downto ⦆ simple_expression ⦆
| ( discrete_range { , … } ) ⟧
```

- Also *resolved subtypes* and types*:*

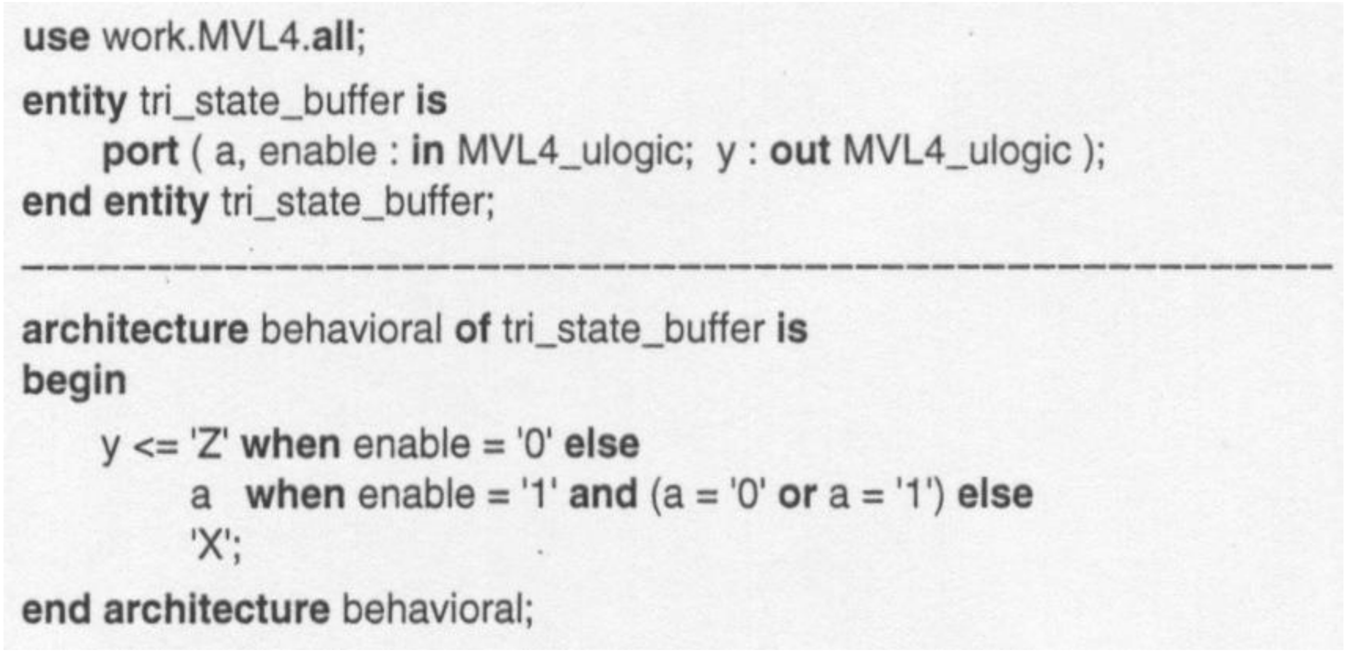
```
subtype resolved_logic is
  resolve_tri_state_logic tri_state_logic;

signal s2, s3 : resolved_logic;
```

Example: a four value logic



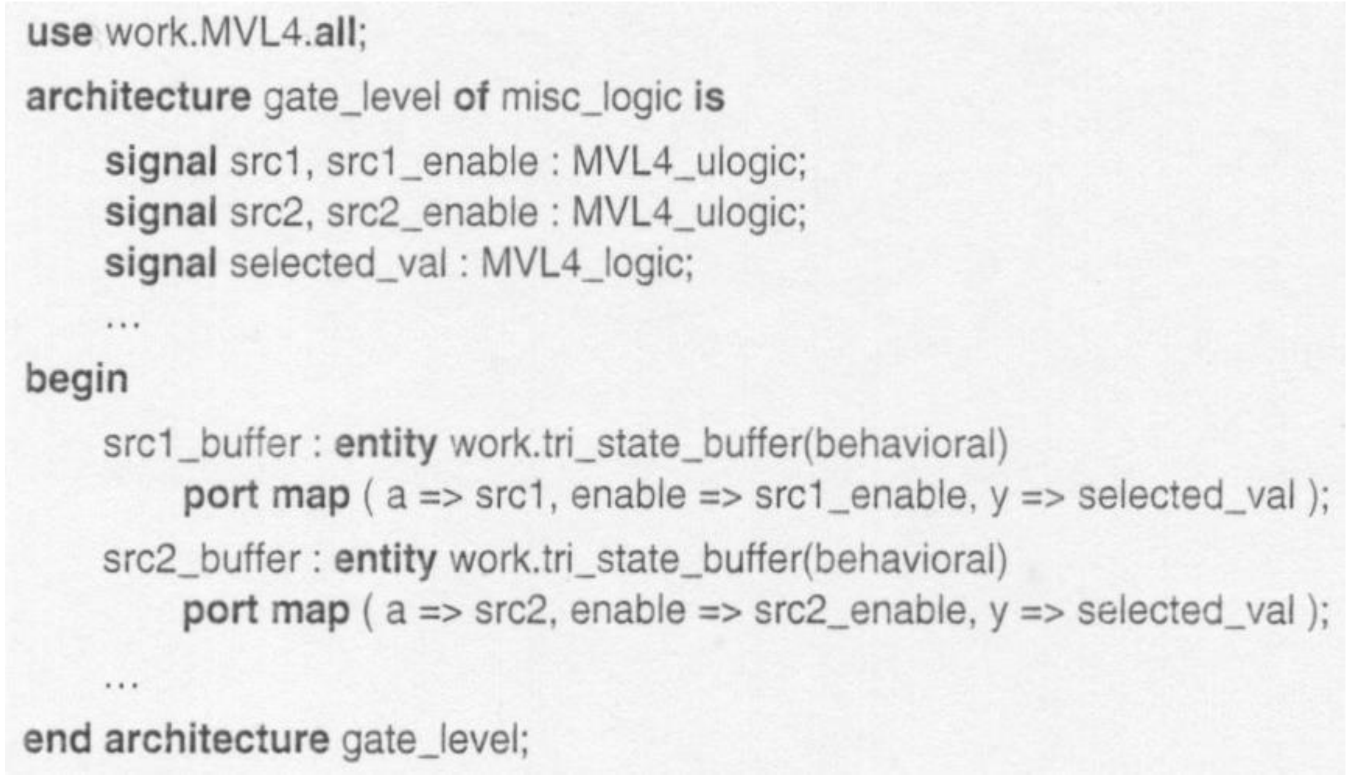
```
package MVL4 is
    type MVL4_ulogic is ('X', '0', '1', 'Z');    -- unresolved logic type
    type MVL4_ulogic_vector is array (natural range <>) of MVL4_ulogic;
    function resolve_MVL4 ( contribution : MVL4_ulogic_vector )
                                return MVL4_ulogic;
    subtype MVL4_logic is resolve_MVL4 MVL4_ulogic;
end package MVL4;
--------------------------------------------------------------------
package body MVL4 is
    type table is array (MVL4_ulogic, MVL4_ulogic) of MVL4_ulogic;
    constant resolution_table : table :=
        -- 'X'    '0'   '1'   'Z'
        -- ---------------
        ( ( 'X',   'X',  'X',  'X' ),      -- 'X'
          ( 'X',   '0',  'X',  '0' ),      -- '0'
          ( 'X',   'X',  '1',  '1' ),      -- '1'
          ( 'X',   '0',  '1',  'Z' ) );    -- 'Z'
    function resolve_MVL4 ( contribution : MVL4_ulogic_vector )
                                return MVL4_ulogic is
        variable result : MVL4_ulogic := 'Z';
    begin
        for index in contribution'range loop
            result := resolution_table(result, contribution(index));
        end loop;
        return result;
    end function resolve_MVL4;
end package body MVL4;
```

Example: a tri-state buffer



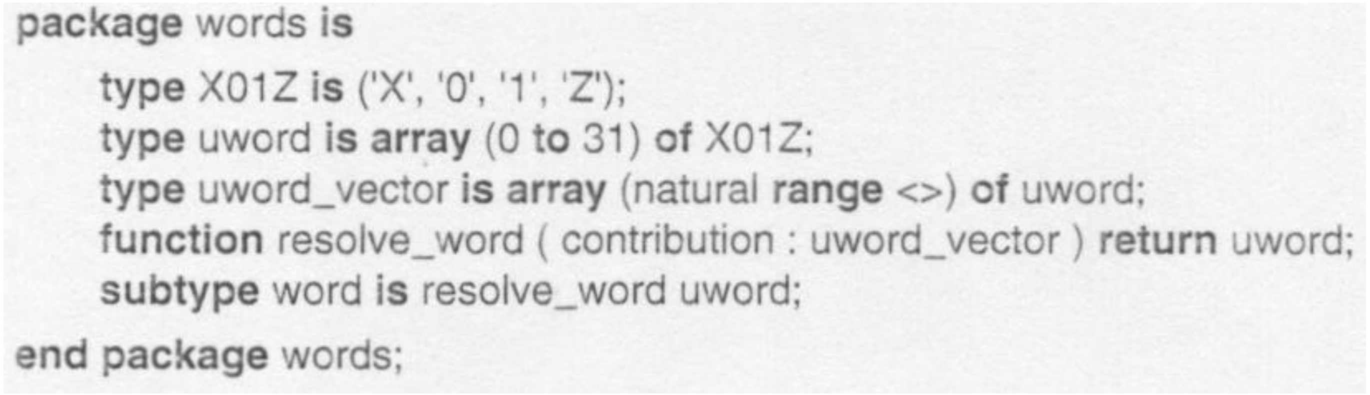
```
use work.MVL4.all;

entity tri_state_buffer is
      port ( a, enable : in MVL4_ulogic;  y : out MVL4_ulogic );
end entity tri_state_buffer;
------------------------------------------------------------------
architecture behavioral of tri_state_buffer is
begin
      y <= 'Z' when enable = '0' else
           a   when enable = '1' and (a = '0' or a = '1') else
           'X';
end architecture behavioral;
```

Example: a tri-state buffer



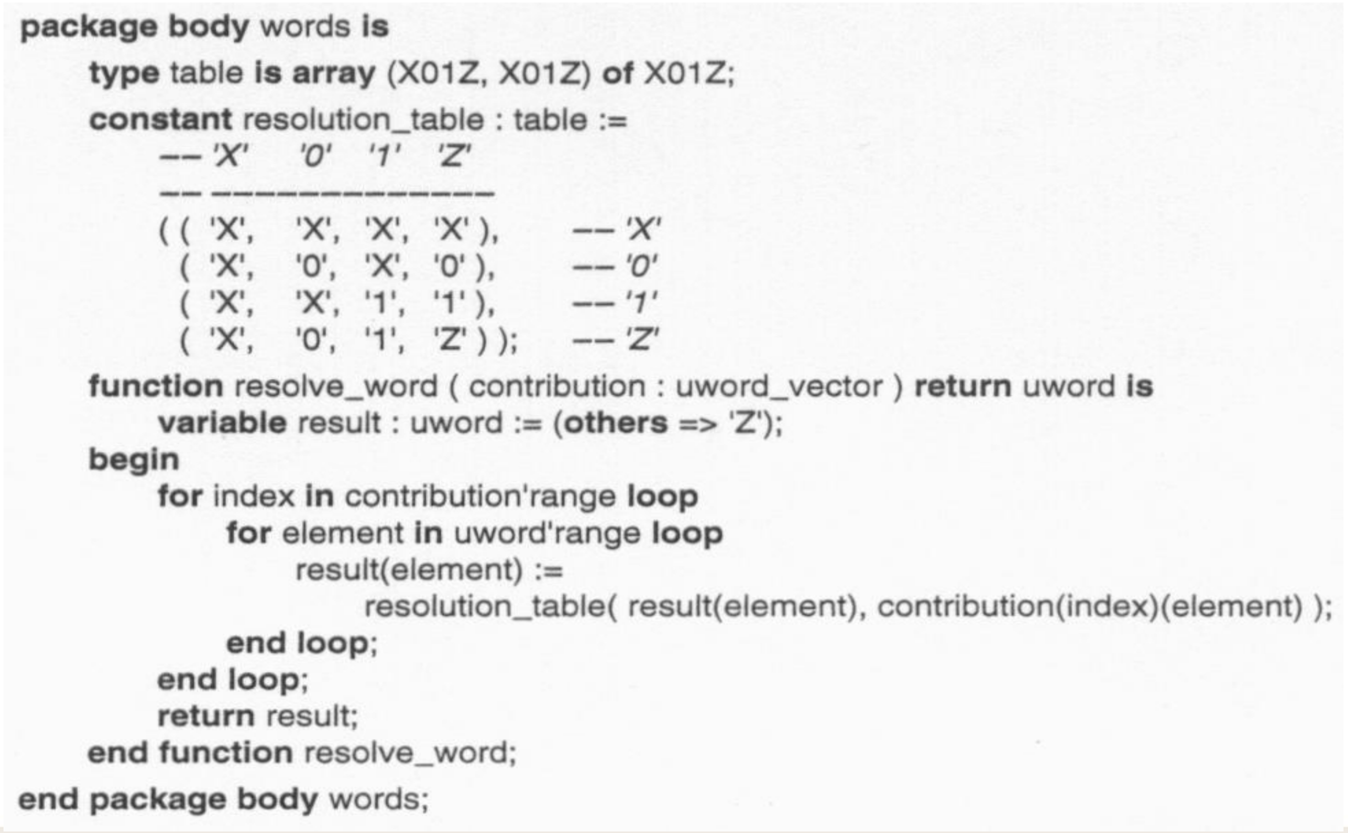
```
use work.MVL4.all;
architecture gate_level of misc_logic is
    signal src1, src1_enable : MVL4_ulogic;
    signal src2, src2_enable : MVL4_ulogic;
    signal selected_val : MVL4_logic;
    ...
begin
    src1_buffer : entity work.tri_state_buffer(behavioral)
        port map ( a => src1, enable => src1_enable, y => selected_val );
    src2_buffer : entity work.tri_state_buffer(behavioral)
        port map ( a => src2, enable => src2_enable, y => selected_val );
    ...
end architecture gate_level;
```

Composite resolved subtype



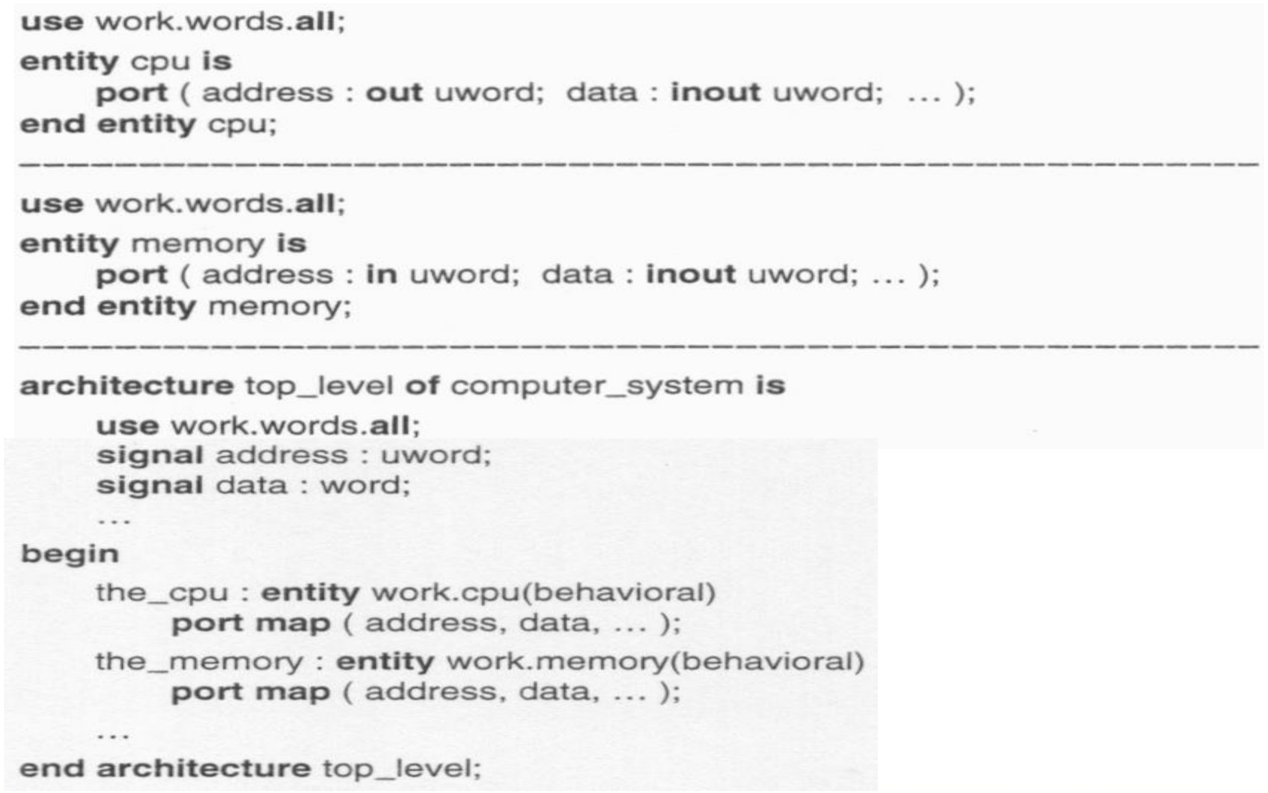
```
package words is
        type X01Z is ('X', '0', '1', 'Z');
        type uword is array (0 to 31) of X01Z;
        type uword_vector is array (natural range <>) of uword;
        function resolve_word ( contribution : uword_vector ) return uword;
        subtype word is resolve_word uword;
end package words;
```

Composite resolved subtype



```
package body words is
    type table is array (X01Z, X01Z) of X01Z;
    constant resolution_table : table :=
        -- 'X'    '0'   '1'   'Z'
        -- ----------------
        ( ( 'X',   'X',  'X',  'X' ),      -- 'X'
          ( 'X',   '0',  'X',  '0' ),      -- '0'
          ( 'X',   'X',  '1',  '1' ),      -- '1'
          ( 'X',   '0',  '1',  'Z' ) );    -- 'Z'
    function resolve_word ( contribution : uword_vector ) return uword is
        variable result : uword := (others => 'Z');
    begin
        for index in contribution'range loop
            for element in uword'range loop
                result(element) :=
                    resolution_table( result(element), contribution(index)(element) );
            end loop;
        end loop;
        return result;
    end function resolve_word;
end package body words;
```

Example: data port inout



```
use work.words.all;
entity cpu is
        port ( address : out uword;  data : inout uword;  ... );
end entity cpu;
-----------------------------------------------------------------
use work.words.all;
entity memory is
        port ( address : in uword;  data : inout uword; ... );
end entity memory;
-----------------------------------------------------------------
architecture top_level of computer_system is
        use work.words.all;
        signal address : uword;
        signal data : word;
        ...
begin
        the_cpu : entity work.cpu(behavioral)
                port map ( address, data, ... );
        the_memory : entity work.memory(behavioral)
                port map ( address, data, ... );
        ...
end architecture top_level;
```

Problem

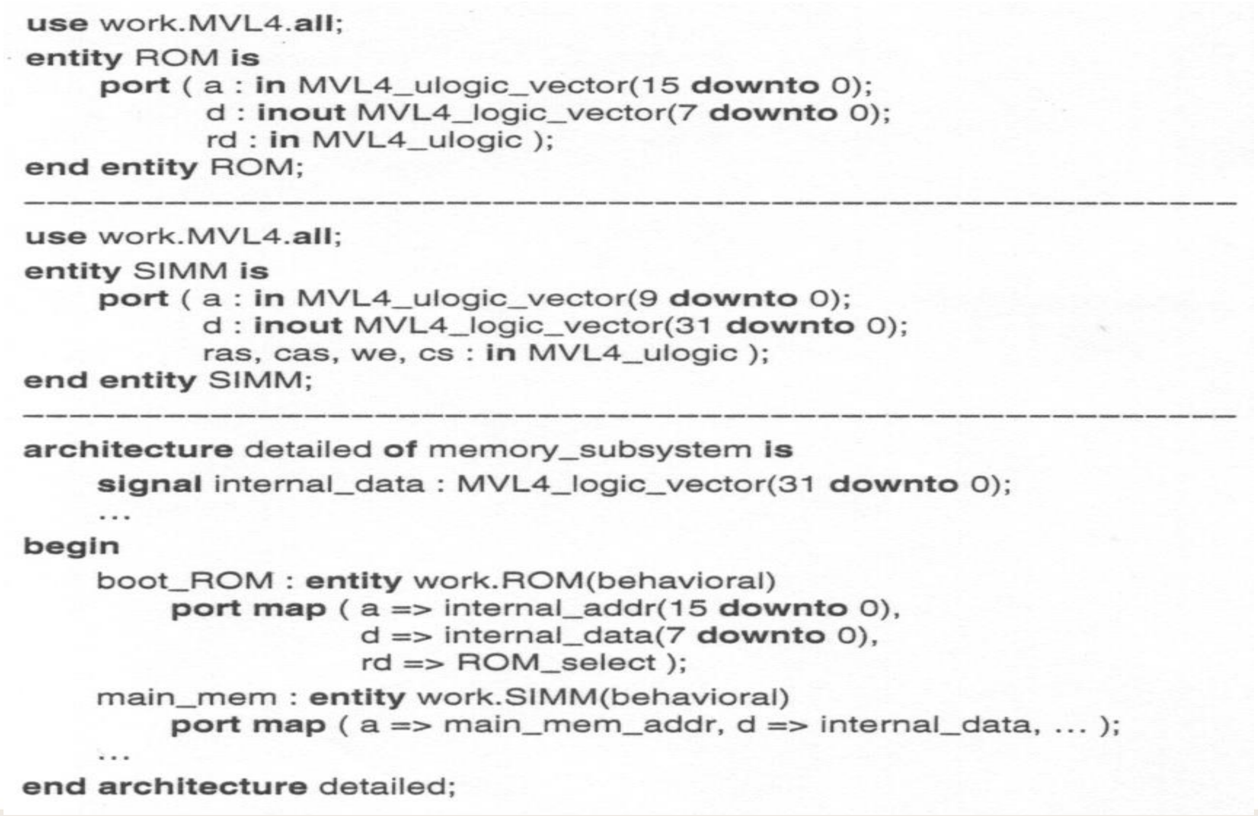
- It is illegal to do this kind of partial assignment:

```
boot_rom : entity work.ROM(behavioral)
      port map ( a => address, d => data(24 to 31), ... );    -- illegal
```

- In the data signal we would have 2 sources for bits 0 to 23, 3 sources for bits 24 to 31.
- Solution: declare a composite type composed by elements of a resolved type:

```
type MVL4_logic_vector is array (natural range <>) of MVL4_logic;
```

Example



```
use work.MVL4.all;
entity ROM is
        port ( a : in MVL4_ulogic_vector(15 downto 0);
                  d : inout MVL4_logic_vector(7 downto 0);
                  rd : in MVL4_ulogic );
end entity ROM;
----------------------------------------------------------------
use work.MVL4.all;
entity SIMM is
        port ( a : in MVL4_ulogic_vector(9 downto 0);
                  d : inout MVL4_logic_vector(31 downto 0);
                  ras, cas, we, cs : in MVL4_ulogic );
end entity SIMM;
----------------------------------------------------------------
architecture detailed of memory_subsystem is
        signal internal_data : MVL4_logic_vector(31 downto 0);
        . . .
begin
        boot_ROM : entity work.ROM(behavioral)
                port map ( a => internal_addr(15 downto 0),
                                 d => internal_data(7 downto 0),
                                 rd => ROM_select );
        main_mem : entity work.SIMM(behavioral)
                port map ( a => main_mem_addr, d => internal_data, ... );
        . . .
end architecture detailed;
```

VHDL 2008: resolved composite subtypes

- In the previous examples, MLV4_ulogic_vector and MLV4_logic_vector are different types, and their values are incompatible.
- To solve this problem, since VHDL 2008 it is possible to define a subtype of a composite type, in which we introduce a resolution function for the elements:

```
subtype MVL4_logic_vector is (resolve_MVL4)MVL4_ulogic_vector;
```

- The syntax rule that is applied is:

```
resolution_indication ⇐
    resolution_function_name
    | (   resolution_indication
        | ⟦ record_element_identifier resolution_indication ⟧ ⦃ , ... ⦄ )
```

VHDL 2008: resolved composite subtypes

```
type unresolved_RAM_content_type is
  array (natural range <>) of MVL4_ulogic_vector;

subtype RAM_content_type is
  ((resolve_MVL4)) unresolved_RAM_content_type;

type unresolved_status_type is record
  valid : MVL4_ulogic;
  dirty : MVL4_ulogic;
  tag : MVL4_ulogic_vector;
end record unresolved_status_type;

subtype status_resolved_valid is
  (valid wired_and) unresolved_status_type;
```

Summarizing

- The *resolved signal* and the *resolved types* are the only way we can connect together multiple drivers for a signal.
- We need a *resolution function* for determining the final value of the signal.
- The *resolution function* must have a single parameter, which is an *unconstrained array* with value of the type of the signal, and must return a value of the type of the signal.
- The type of the array’s index does not matter, provided that it is capable to enumerate the largest collection of sources for the *resolved signal*.
- The *resolution function* must be a **pure** function, i.e., must not have side effects.
- Since the order of the contributions passed to the function is unknown, it must be a *commutative* function.

Summarizing

- During simulation, the *resolution function* is called every time one of the *resolved signal* sources is active.
- During synthesis, the *resolution function* specifies how the synthesized hardware shall combine the values of the different sources of the *resolved signal*.

IEEE Std_Logic_1164 resolved subtypes

```
type std_ulogic is ('U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '–');
type std_ulogic_vector is array ( natural range <> ) of std_ulogic;
```

- ulogic means unresolved logic
- The resolved types are:

```
function resolved ( s : std_ulogic_vector ) return std_ulogic;
subtype std_logic is resolved std_ulogic;
type std_logic_vector is array ( natural range <>) of std_logic;
```

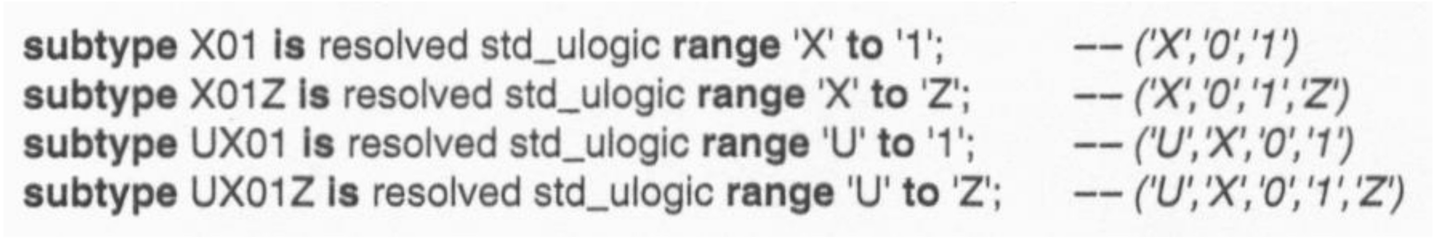
- VHDL 2008:

```
subtype std_logic_vector is (resolved) std_ulogic_vector;
```

The resolution function

```
type stdlogic_table is array (std_ulogic, std_ulogic) of std_ulogic;
constant resolution_table : stdlogic_table :=
  -- ---------------------------------------------
  --  'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'
  -- ---------------------------------------------
  ( ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ),  --  'U'
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ),  --  'X'
    ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X' ),  --  '0'
    ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X' ),  --  '1'
    ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X' ),  --  'Z'
    ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X' ),  --  'W'
    ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X' ),  --  'L'
    ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X' ),  --  'H'
    ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' )   --  '-'
  );

function resolved ( s : std_ulogic_vector ) return std_ulogic is
  variable result : std_ulogic := 'Z';  -- weakest state default
begin
  if s'length = 1 then
    return s(s'low);
  else
    for i in s'range loop
      result := resolution_table(result, s(i));
    end loop;
  end if;
  return result;
end function resolved;
```

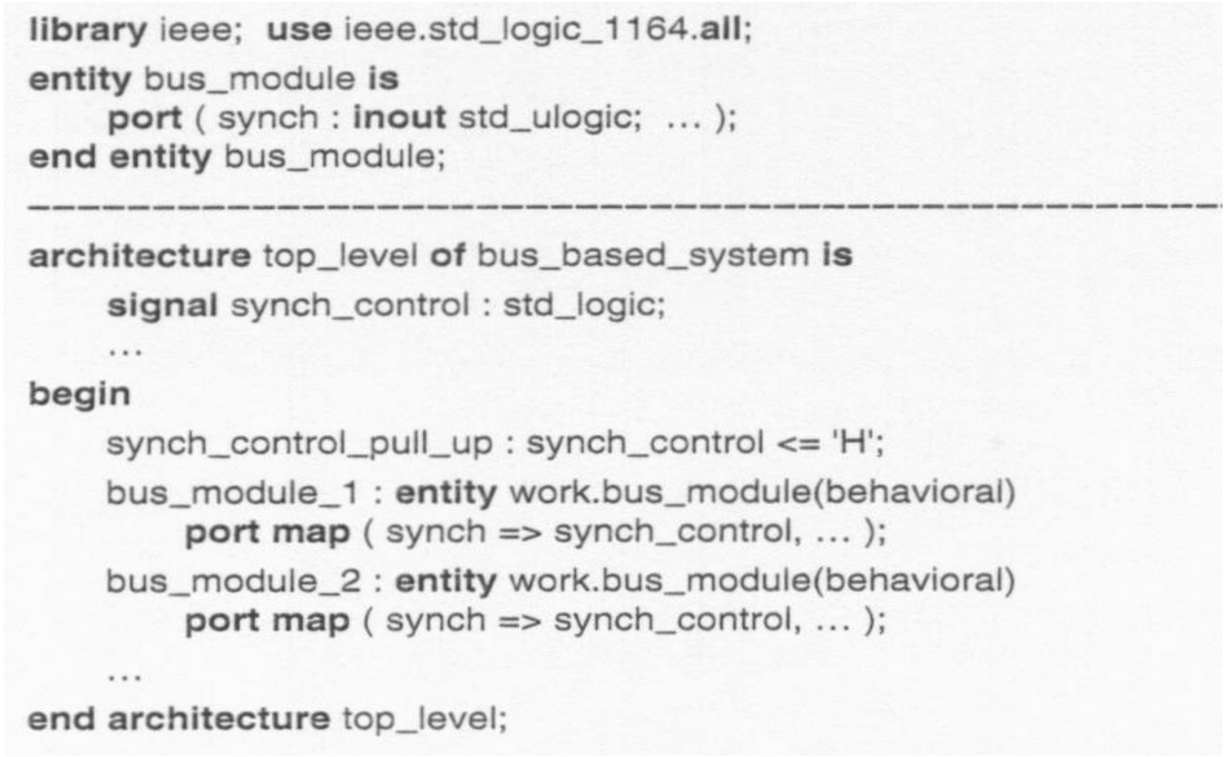
Resolved subtypes


```
subtype X01 is resolved std_ulogic range 'X' to '1';       -- ('X','0','1')
subtype X01Z is resolved std_ulogic range 'X' to 'Z';      -- ('X','0','1','Z')
subtype UX01 is resolved std_ulogic range 'U' to '1';      -- ('U','X','0','1')
subtype UX01Z is resolved std_ulogic range 'U' to 'Z';     -- ('U','X','0','1','Z')
```

Port *inout* with a resolved signal

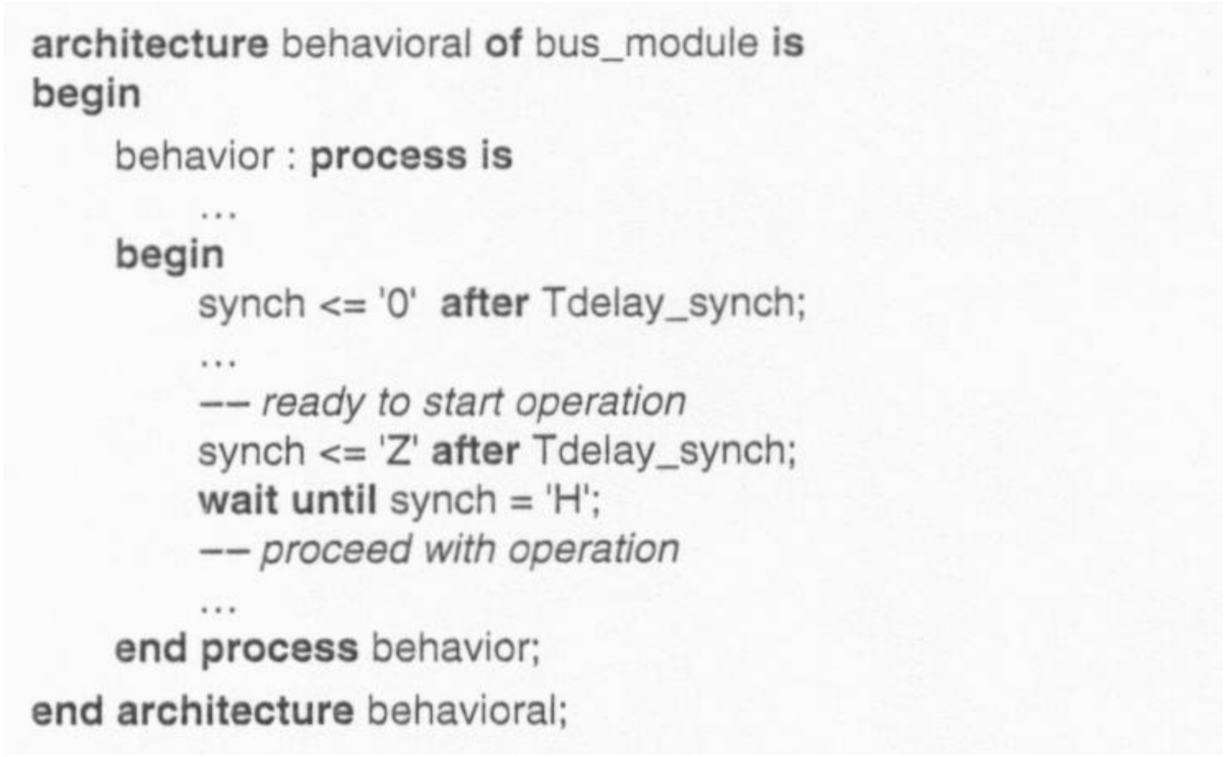
- Let us consider a port with mode ***inout*** connected to a resolved signal.
- Which is the value at the port’s input: the value driven by the port or the value of the resolved signal?
- The value of the resolved signal.
- A port with mode ***inout*** is modeled at the output as a driver that contributes to the signal value and at the input as a sensor that monitors the actual value of the signal.

Example: wired-and synchronization



```
library ieee;  use ieee.std_logic_1164.all;
entity bus_module is
        port ( synch : inout std_ulogic;  ... );
end entity bus_module;
--------------------------------------------------------------
architecture top_level of bus_based_system is
        signal synch_control : std_logic;
        ...
begin
        synch_control_pull_up : synch_control <= 'H';
        bus_module_1 : entity work.bus_module(behavioral)
                port map ( synch => synch_control, ... );
        bus_module_2 : entity work.bus_module(behavioral)
                port map ( synch => synch_control, ... );
        ...
end architecture top_level;
```

Example: wired-and synchronization

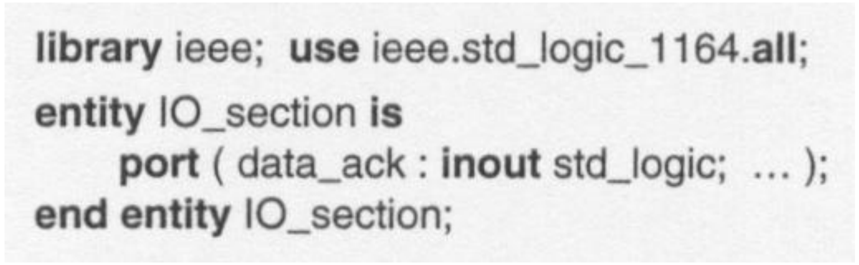


```
architecture behavioral of bus_module is
begin
    behavior : process is
        ...
    begin
        synch <= '0'  after Tdelay_synch;
        ...
        -- ready to start operation
        synch <= 'Z' after Tdelay_synch;
        wait until synch = 'H';
        -- proceed with operation
        ...
    end process behavior;
end architecture behavioral;
```

Resolved ports

- The ports of an entity can also be declared of a *resolved subtype.*
- Example: when an architecture comprises a certain number of processes that drive a port or a certain number of *component instances* whose outputs are connected to the same port.
- The final value of the *resolved port* will be determined by resolving all sources inside the architecture body.

Resolved port *inout*



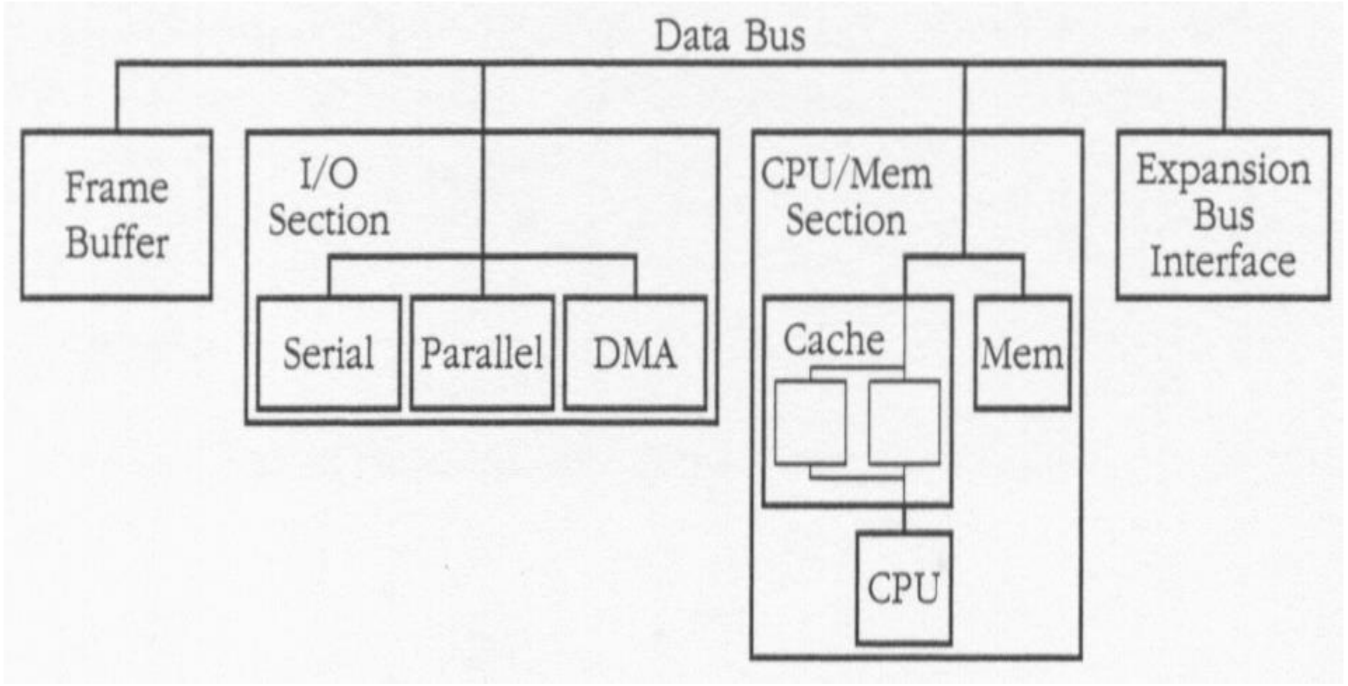
```
library ieee;  use ieee.std_logic_1164.all;
entity IO_section is
        port ( data_ack : inout std_logic;  ... );
end entity IO_section;
```

- We could have different I/O controllers connected to the same port of *data_ack*.
- At the top level this port could be connected to a resolved signal.
- The value of the signal will be resolved after the value driven by the *resolved port* is resolved.
- The *resolution function* of the signal could be different from that of the port.

Resolved port hierarchy

- We could consider different levels of port hierarchy with a process nested at the lowest level that drives a value to be passed through some *resolved ports* up to a signal at the top level.
- The value of the signal at the highest level is called *effective value* of the signal.
- The *effective value* is returned down the port hierarchy for determining the actual value of each port with mode ***in*** or ***inout***.

Example


Data Bus
Frame Buffer
I/O Section
Serial
Parallel
DMA
CPU/Mem Section
Cache
Mem
CPU
Expansion Bus Interface

‘driving_value attribute

- Allows a process to find the value it is presently contributing to a *resolved signal*.
- It cannot be used for determining the value contributed from other processes.

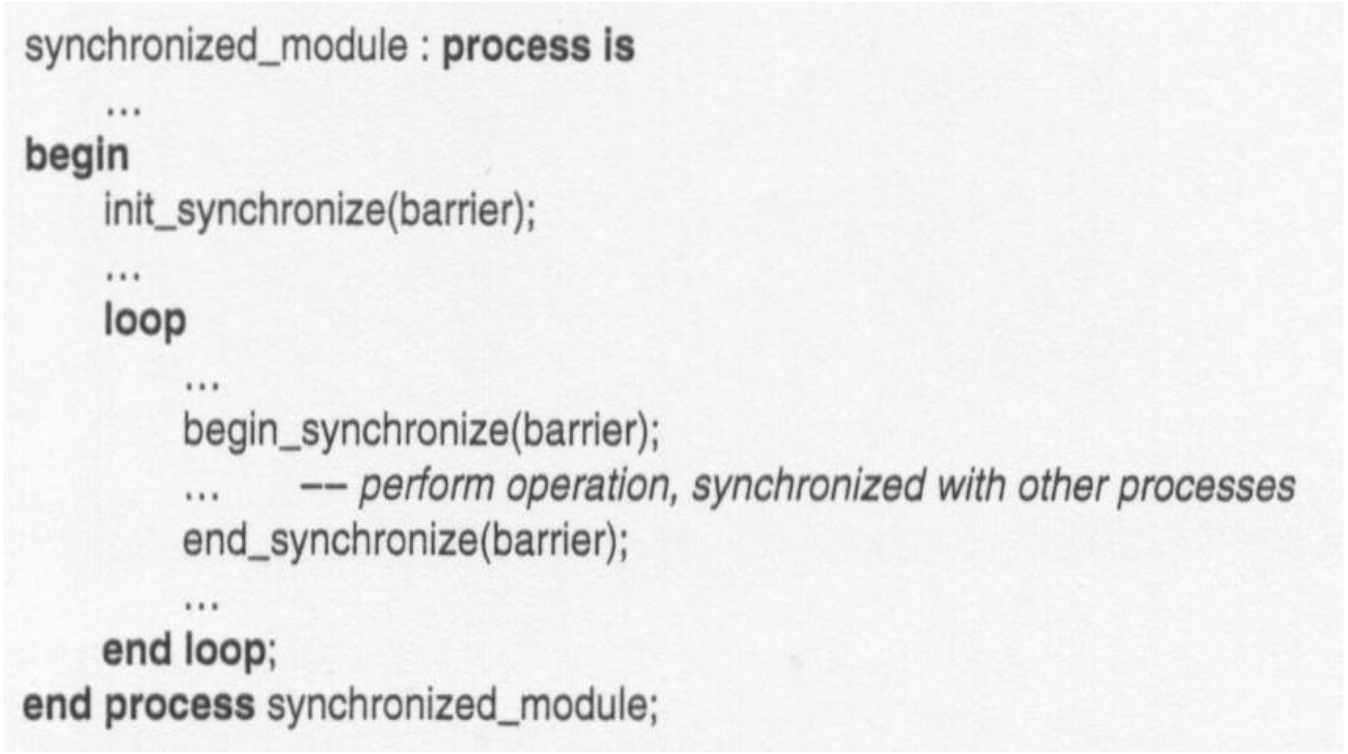
Resolved signal parameters

- In case of signal parameters in subprograms with mode ***out*** : no signal resolution is executed inside the subprogram.
- In case of signal parameters with mode ***in*** : the subprogram sees the effective value of the signal.
- In case of signal parameters with mode ***inout*** : the subprogram sees the effective value of the signal and no internal resolution is performed.

Example

```
procedure init_synchronize ( signal synch : out std_logic ) is
begin
        synch <= '0';
end procedure init_synchronize;

procedure begin_synchronize ( signal synch : inout std_logic;
                              Tdelay : in delay_length := 0 fs ) is
begin
        synch <= 'Z' after Tdelay;
        wait until synch = 'H';
end procedure begin_synchronize;

procedure end_synchronize ( signal synch : inout std_logic;
                            Tdelay : in delay_length := 0 fs ) is
begin
        synch <= '0' after Tdelay;
        wait until synch = '0';
end procedure end_synchronize;
```

Example



```
synchronized_module : process is
        ...
begin
        init_synchronize(barrier);
        ...
        loop
                ...
                begin_synchronize(barrier);
                ...        -- perform operation, synchronized with other processes
                end_synchronize(barrier);
                ...
        end loop;
end process synchronized_module;
```

See:

- Peter Ashenden, «The designers' guide to VHDL» Morgan Kaufmann,
  - Chapter 8