UNIVERSITÀ
DEGLI STUDI DI TRIESTE

dipartimento
di ingegneria
e architettura

06 - Basic system modeling concepts

A.Carini – Progettazione di sistemi elettronici

Entity declarations

```
entity_declaration ⇐
    entity identifier is
        [ port ( port_interface_list ) ; ]
        { entity_declarative_item }
    end [ entity ] [ identifier ] ;
interface_list ⇐
    ( identifier { , ... } : [ mode ] subtype_indication [ := expression ] ) { ; ... }
mode ⇐ in | out | inout
```

From VHDL 2008:

```
mode ⇐ in | out | buffer | inout
```

Entity declaration examples

```
entity adder is
    port ( a : in word;
           b : in word;
           sum : out word );
end entity adder;
```

```
entity adder is
    port ( a, b : in word;
           sum : out word );
end entity adder;
```

```
entity and_or_inv is
    port ( a1, a2, b1, b2 : in bit := '1';
           y : out bit );
end entity and_or_inv;
```

```
entity top_level is
end entity top_level;
```

Entity declaration examples

```
entity program_ROM is
    port ( address : in std_ulogic_vector(14 downto 0);
           data : out std_ulogic_vector(7 downto 0);
           enable : in std_ulogic );

    subtype instruction_byte is bit_vector(7 downto 0);
    type program_array is array (0 to 2**14 – 1) of instruction_byte;
    constant program : program_array
        := ( X"32", X"3F", X"03",          -- LDA  $3F03
             X"71", X"23",                 -- BLT   $23
             ...
             );
end entity program_ROM;
```

Architecture bodies

```
architecture_body ⇐
    architecture identifier of entity_name is
        { block_declarative_item }
    begin
        { concurrent_statement }
    end [ architecture ] [ identifier ] ;
```

Signal declaration

signal_declaration ⇐
signal identifier { , ... } : subtype_indication [:= expression] ;

Example:

```
architecture primitive of and_or_inv is
    signal and_a, and_b : bit;
    signal or_a_b : bit;
begin
    and_gate_a : process (a1, a2) is
    begin
        and_a <= a1 and a2;
    end process and_gate_a;
    and_gate_b : process (b1, b2) is
    begin
        and_b <= b1 and b2;
    end process and_gate_b;
    or_gate : process (and_a, and_b) is
    begin
        or_a_b <= and_a or and_b;
    end process or_gate;
    inv : process (or_a_b) is
    begin
        y <= not or_a_b;
    end process inv;
end architecture primitive;
```

Signal Assignment

signal_assignment_statement ⇐
    ⟦ label : ⟧ name <= ⟦ delay_mechanism ⟧ waveform ;
waveform ⇐ (*value*_expression ⟦ **after** *time*_expression ⟧) { , ... }

- Example:

```
y <= not or_a_b after 5 ns;
```

- Two interpretations:
  - For a Description.
  - For a simulation.

Simulation time

- It is the time at which the modeled system is assumed to work.
- It is different from the real time of the simulation on the host computer.
- It starts from 0 at the beginning of the simulation and it is updated with discrete steps.
- This is called a discrete event simulation.

Transactions and Events

- A signal assignment schedules a *transaction* on the signal, i.e.,
- it schedules a novel value and a simulation time when the novel value shall be applied to the signal.
- When the time reaches the moment of a transaction, the signal is said to be *active.*
- If the novel value is different from the previous one, we say we have an *event* on the signal.
- Processes respond to the events, not to the transactions.

Assignments with multiple transactions

```
clk <= '1' after T_pw, '0' after 2*T_pw;
```

- If executed at time T0 = 50ns, T_pw = 10ns:
  - First transaction for T1 = 60ns.
  - Second transaction for T2 = 70ns.

```
clock_gen : process (clk) is
begin
    if clk = '0' then
        clk <= '1' after T_pw, '0' after 2*T_pw;
    end if;
end process clock_gen;
```

Multiple Assignments

```
mux : process (a, b, sel) is
begin
      case sel is
            when '0' =>
                  z <= a after prop_delay;
            when '1' =>
                  z <= b after prop_delay;
      end case;
end process mux;
```

Drivers

- A process defines a *driver* for a signal if and only if it contains at least an assignment on that signal.
- A process can have several drivers.
- A driver is a source for the signal, it provides values that will be applied to the signal.
- *Normal signals* can have only one source.
- Different processes can not assign the same “normal” signal.
- Busses and wired-ORs are modeled with special signals: the *resolved signals* (to be studied later on).

Signal attributes

S'delayed(T)	A signal that takes on the same values as S but is delayed by time T.
S'stable(T)	A Boolean signal that is true if there has been no event on S in the time interval T up to the current time, otherwise false.
S'quiet(T)	A Boolean signal that is true if there has been no transaction on S in the time interval T up to the current time, otherwise false.
S'transaction	A signal of type bit that changes value from '0' to '1' or vice versa each time there is a transaction on S.
S'event	True if there is an event on S in the current simulation cycle, false otherwise.
S'active	True if there is a transaction on S in the current simulation cycle, false otherwise.
S'last_event	The time interval since the last event on S.
S'last_active	The time interval since the last transaction on S.
S'last_value	The value of S just before the last event on S.

Signal attributes: examples

- Set-up time verification:

```
if clk'event and (clk = '1' or clk = 'H')
        and (clk'last_value = '0' or clk'last_value = 'L') then
    assert d'last_event >= Tsu
        report "Timing error: d changed within setup time of clk";
end if;
```

- Clock duration verification:

```
assert (not clk'event) or clk'delayed'last_event >= Tpw_clk
    report "Clock frequency too high";
```

Edge triggered flip-flop

```
entity edge_triggered_Dff is
    port ( D : in bit;  clk : in bit;  clr : in bit;
           Q : out bit );
end entity edge_triggered_Dff;
-------------------------------------

architecture behavioral of edge_triggered_Dff is
begin
    state_change : process (clk, clr) is
    begin
        if clr = '1' then
            Q <= '0' after 2 ns;
        elsif clk'event and clk = '1' then
            Q <= D after 2 ns;
        end if;
    end process state_change;
end architecture behavioral;
```

Wait statement

```
wait_statement ⇐
    [ label : ] wait [ on signal_name { , ... } ]
                     [ until boolean_expression ]
                     [ for time_expression ] ;
```

Examples:

```
half_add : process is
begin
    sum <= a xor b after T_pd;
    carry <= a and b after T_pd;
    wait on a, b;
end process half_add;
```

~

```
half_add : process (a, b) is
begin
    sum <= a xor b after T_pd;
    carry <= a and b after T_pd;
end process half_add;
```

Example: a multiplexer

```
entity mux2 is
      port ( a, b, sel : in bit;
                z : out bit );
end entity mux2;
-------------------------------------
architecture behavioral of mux2 is
      constant prop_delay : time := 2 ns;
begin
      slick_mux : process is
      begin
            case sel is
                  when '0' =>
                        z <= a after prop_delay;
                        wait on sel, a;
                  when '1' =>
                        z <= b after prop_delay;
                        wait on sel, b;
            end case;
      end process slick_mux;
end architecture behavioral;
```

Wait until

- **Until** specifies a condition that must be verified before the process restarts.
- Example:

```
wait until clk = '1';
```

- The process is suspended even if clk=‘1’.
- It will be resumed only when clk becomes ‘1’ again.
- If not specified, the *sensitivity list* is given by the signals used in the **until** expression.
- If specified, the expression of **until** is evaluated only on the events on the signals of the sensitivity list.

Wait until: examples

```
clock_gen : process is
begin
      clk <= '1' after T_pw, '0' after 2*T_pw;
      wait until clk = '0';
end process clock_gen;
```

```
wait on clk until reset = '0';
```

Wait for

- **Wait for** specifies a *time-out* time after which the process is anyway resumed.

```
wait until trigger = '1' for 1 ms;
```

```
clock_gen : process is
begin
    clk <= '1' after T_pw, '0' after 2*T_pw;
    wait for 2*T_pw;
end process clock_gen;
```

Wait;

- It causes the indefinite suspension of the process.
- Useful in test-benches for ending the stimulus process.
- Example:

```
test_gen : process is
begin
    test0 <= '0' after 10 ns, '1' after 20 ns, '0' after 30 ns, '1' after 40 ns;
    test1 <= '0' after 10 ns, '1' after 30 ns;
    wait;
end process test_gen;
```

Delta delays

- In signal assignments we often omit the delay value.
- It is equivalent to specify a delay of 0 fs.
- The new value shall be applied to the signal at the current simulation time.
- The signal does not change during the assignment execution.
- On the contrary, a signal transaction is scheduled and it will become active when all processes are suspended.
- The process does not feel the signal assignment effect till the next reactivation!
- A delay 0 fs delay is called *delta delay.*

Simulation

- The simulation cycle has two phases:
- Signal update phase:
  - The simulation time is set at the time of the first scheduled transaction.
  - The values of all scheduled transactions planned for this moment are applied to the signals.
  - There could be events on the signals.
- Execution:
  - The processes that respond to these events are reactivated and executed till they suspend on a wait statement.

Delta delay example

- At time t during the execution phase of a process we execute:

```
data <= X"00";
```

- A transaction on *data* is planned at time t with value X”00”.
- The transaction is not applied immediately, because we are still in the execution phase;
  - *data* does not change!
- All the processes stop on a **wait** statement.
- A new simulation cycle begin.
- The simulation time is updated and it does not change because at t we have a new transaction.
- *data* becomes X”00” and the processes that respond to this new value are reactivated.

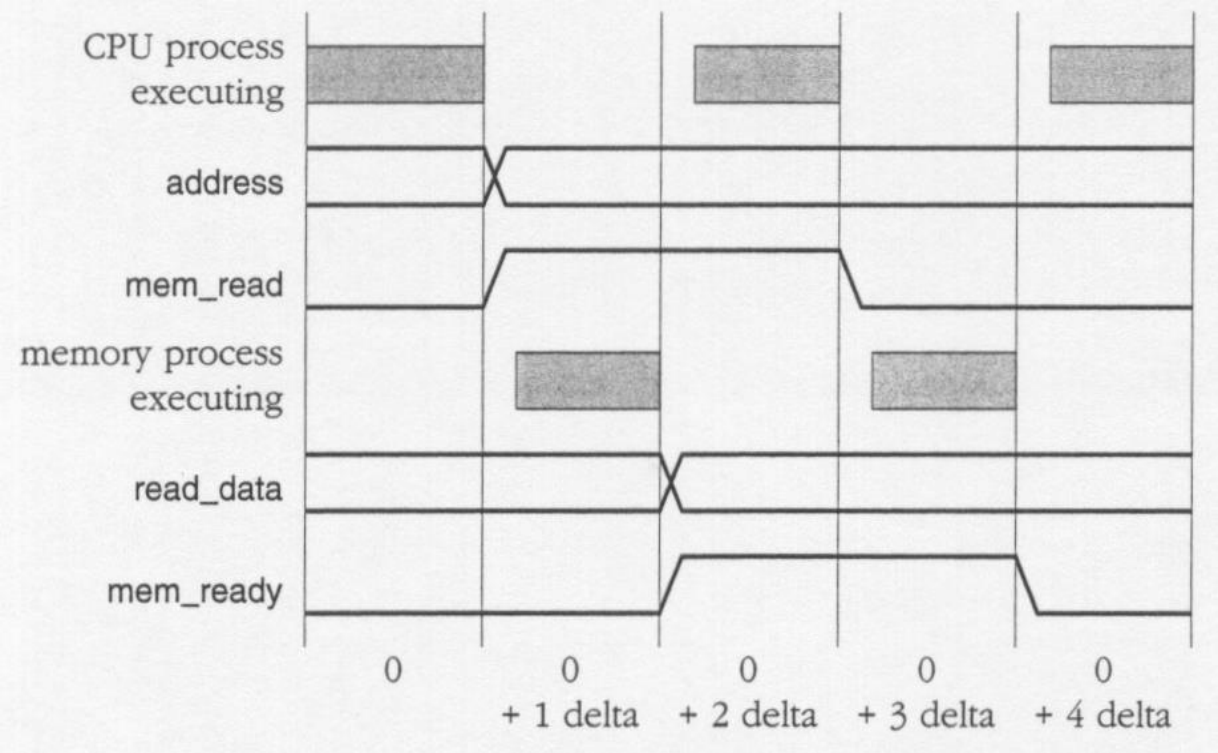
Example of a CPU

```
architecture abstract of computer_system is
    subtype word is bit_vector(31 downto 0);
    signal address : natural;
    signal read_data, write_data : word;
    signal mem_read, mem_write : bit := '0';
    signal mem_ready : bit := '0';
begin
    cpu : process is
        variable instr_reg : word;
        variable PC : natural;
        ...        -- other declarations
    begin
        loop
            address <= PC;
            mem_read <= '1';
            wait until mem_ready = '1';
            instr_reg := read_data;
            mem_read <= '0';
            wait until mem_ready = '0';
            PC := PC + 4;
            ...        -- execute the instruction
        end loop;
    end process cpu;
```

Example of a CPU

```
    memory : process is
        type memory_array is array (0 to 2**14 – 1) of word;
        variable store : memory_array := (
                ...
                );
    begin
        wait until mem_read = '1' or mem_write = '1';
        if mem_read = '1' then
            read_data <= store( address / 4 );
            mem_ready <= '1';
            wait until mem_read = '0';
            mem_ready <= '0';
        else
            ...     -- perform write access
        end if;
    end process memory;
end architecture abstract;
```

Example of a CPU



Delay mechanisms

signal_assignment_statement ⇐
    [label :] name <= [delay_mechanism] waveform ;
waveform ⇐ (*value*_expression [**after** *time*_expression]) { , ... }

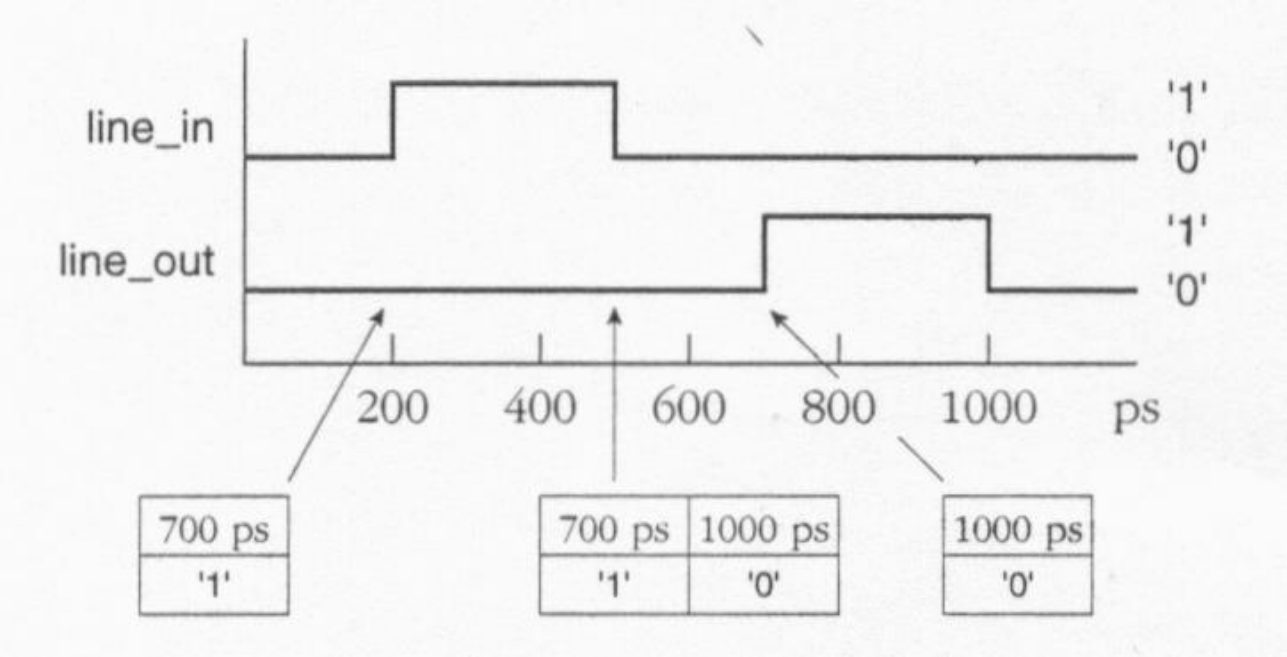
- If there is a pending transaction on a signal and we assign again that signal, the way the two assignments are combined depends from the delay mechanism, which can be **inertial** or **transport.**
- The delay mechanism specification is optional, if not specified the default delay mechanism is **inertial.**

Transport delay mechanism

- Used to model systems where every input pulse, no matter how short, produces an output pulse.
- Example: an ideal transmission line

```
transmission_line : process (line_in) is
begin
    line_out <= transport line_in after 500 ps;
end process transmission_line;
```

Transport delay mechanism

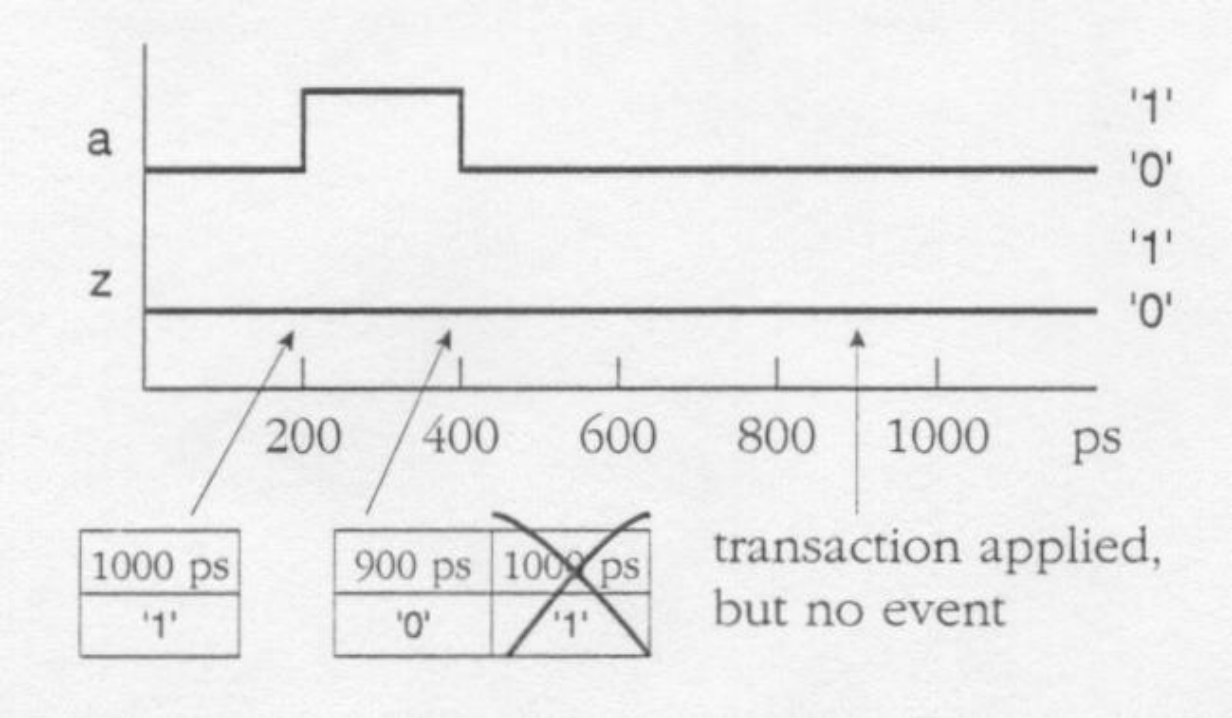


Transport delay mechanism

- If we schedule a transaction for a time sooner than that of other scheduled transactions, these later transactions are erased from the driver queue.
- Example: asymmetric delay with a short pulse

```
asym_delay : process (a) is
    constant Tpd_01 : time := 800 ps;
    constant Tpd_10 : time := 500 ps;
begin
    if a = '1' then
        z <= transport a after Tpd_01;
    else  -- a = '0'
        z <= transport a after Tpd_10;
    end if;
end process asym_delay;
```

Transport delay mechanism

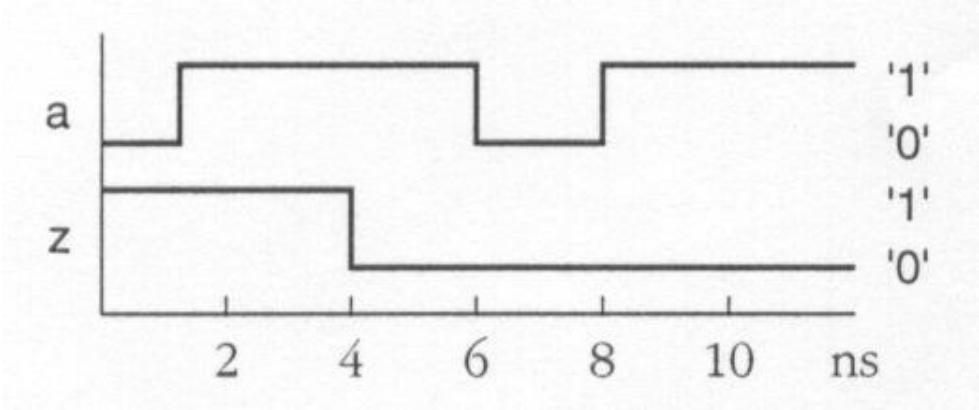


Inertial delay mechanism

- Real systems always have capacitive or inductive effects.
- These effects introduce some inertia in the systems.
- The output changes only it we apply a certain input for a sufficiently long time.
- In VHDL, this behavior is modeled with the **inertial** delay mechanism.
- *Inertial delay:* every input pulse shorter than the propagation delay is rejected (i.e., absorbed) by the system and does not produce any output.

Inertial delay mechanism

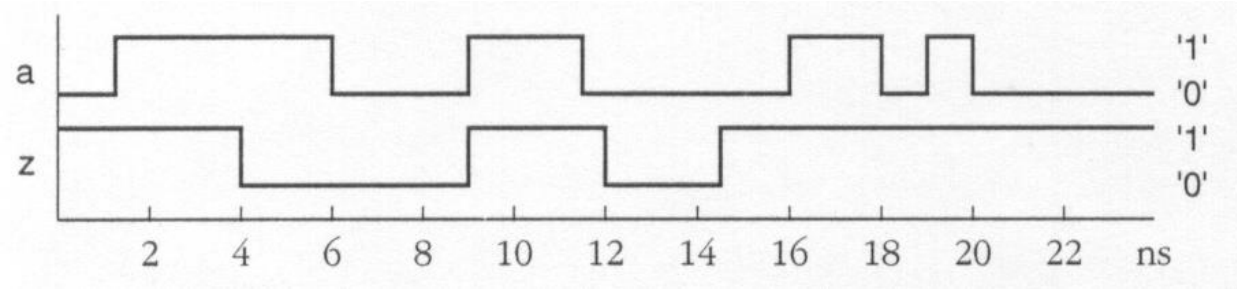
```
inv : process (a) is
begin
      y <= inertial not a after 3 ns;
end process inv;
```



Inertial delay mechanism

- We can also specify the maximum duration of a rejected pulse:

```
inv : process (a) is
begin
      y <= reject 2 ns inertial not a after 3 ns;
end process inv;
```



- If the input originates a pulse shorter than 2 ns, the pulse will be absorbed by the system.

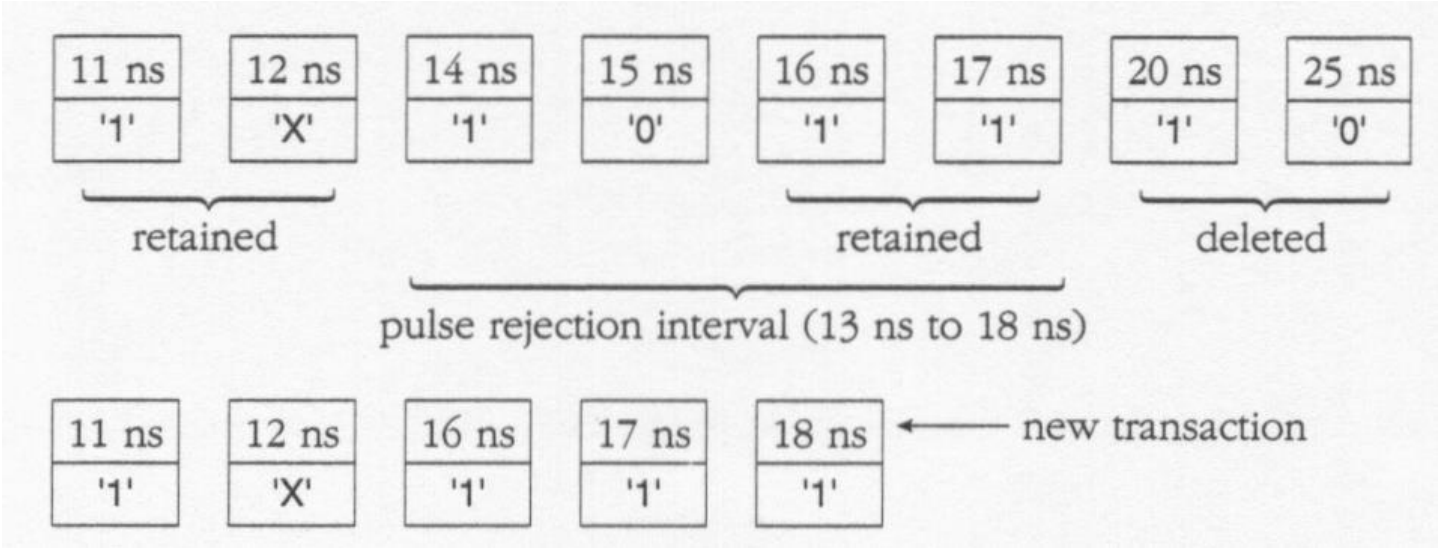
Inertial delay mechanism

- Let us assume to have variable delays.
- We schedule a new transaction for time t_new with *pulse rejection* t_r.
- All transaction scheduled for t > t_new will be erased from the driver.
- We then check one after the other all the transactions in the interval t_new > t > t_new-t_r, starting from t_new and moving towards t_new-tr.
- If the transaction has the same value of the transaction scheduled for t_new, the transaction is kept.
- If the transaction has a different value, that transactions is erased together with all the other transactions still to be checked.

Inertial delay mechanism example

- Assignment done at t=10 ns:

```
s <= reject 5 ns inertial '1' after 8 ns;
```



Example: AND gate

```
library ieee;  use ieee.std_logic_1164.all;
entity and2 is
      port ( a, b : in std_ulogic;  y : out std_ulogic );
end entity and2;
-----------------------------------------
architecture detailed_delay of and2 is
      signal result : std_ulogic;
begin
      gate : process (a, b) is
      begin
            result <= a and b;
      end process gate;
       delay : process (result) is
       begin
            if result = '1' then
                  y <= reject 400 ps inertial '1' after 1.5 ns;
            elsif result = '0' then
                  y <= reject 300 ps inertial '0' after 1.2 ns;
            else
                  y <= reject 300 ps inertial 'X' after 500 ps;
            end if;
       end process delay;
 end architecture detailed_delay;
```

Process statement

```
process_statement ⇐
    [ process_label : ]
    process [ ( signal_name { , ... } ) ] [ is ]
        { process_declarative_item }
    begin
        { sequential_statement }
    end process [ process_label ] ;
```

Concurrent signal assignment statement

```
concurrent_signal_assignment_statement ⇐
    [ label : ] conditional_signal_assignment
  | [ label : ] selected_signal_assignment
```

Conditional signal assignment statement

```
conditional_signal_assignment ⇐
    name <= [ delay_mechanism ]
                { waveform when boolean_expression else }
                waveform [ when boolean_expression ] ;
```

Example

```
zmux : z <= d0 when sel1 = '0' and sel0 = '0' else
            d1 when sel1 = '0' and sel0 = '1' else
            d2 when sel1 = '1' and sel0 = '0' else
            d3 when sel1 = '1' and sel0 = '1';
-------------------------------------------------
zmux : process is
begin
    if sel1 = '0' and sel0 = '0' then
        z <= d0;
    elsif sel1 = '0' and sel0 = '1' then
        z <= d1;
    elsif sel1 = '1' and sel0 = '0' then
        z <= d2;
    elsif sel1 = '1' and sel0 = '1' then
        z <= d3;
    end if;
    wait on d0, d1, d2, d3, sel0, sel1;
end process zmux;
```

Example modified

```
zmux : z <= d0 when sel1 = '0' and sel0 = '0' else
            d1 when sel1 = '0' and sel0 = '1' else
            d2 when sel1 = '1' and sel0 = '0' else
            d3;
-------------------------------------------------
zmux : process is
begin
    if sel1 = '0' and sel0 = '0' then
        z <= d0;
    elsif sel1 = '0' and sel0 = '1' then
        z <= d1;
    elsif sel1 = '1' and sel0 = '0' then
        z <= d2;
    else
        z <= d3;
    end if;
    wait on d0, d1, d2, d3, sel0, sel1;
end process zmux;
```

Examples without conditions

```
PC_incr : next_PC <= PC + 4 after 5 ns;
```

```
PC_incr : process is
begin
    next_PC <= PC + 4 after 5 ns;
    wait on PC;
end process PC_incr;
```

Example: an initialization

```
reset_gen : reset <= '1', '0' after 200 ns when extended_reset else
                     '1', '0' after 50 ns;
```

```
reset_gen : process is
begin
    if extended_reset then
        reset <= '1', '0' after 200 ns;
    else
        reset <= '1', '0' after 50 ns;
    end if;
    wait;
end process reset_gen;
```

Example: with delay mechanism

```
asym_delay : z <= transport a after Tpd_01 when a = '1' else
                  a after Tpd_10;
```

Unaffected

```
scheduler :
    request <= first_priority_request after scheduling_delay
                    when priority_waiting and server_status = ready else
               first_normal_request after scheduling_delay
                    when not priority_waiting and server_status = ready else
               unaffected
                    when server_status = busy else
               reset_request after scheduling_delay;
```

```
scheduler : process is
begin
    if priority_waiting and server_status = ready then
        request <= first_priority_request after scheduling_delay;
    elsif not priority_waiting and server_status = ready then
        request <= first_normal_request after scheduling_delay;
    elsif server_status = busy then
        null;
    else
        request <= reset_request after scheduling_delay;
    end if;
    wait on first_priority_request, priority_waiting, server_status,
            first_normal_request, reset_request;
end process scheduler;
```

Selected signal assignment

```
selected_signal_assignment ⇐
    with expression select
        name <= ⟦ delay_mechanism ⟧
                    { waveform when choices , }
                    waveform when choices ;
```

- The corresponding process implements a case statement.

Example

```
alu : with alu_function select
          result <=  a + b after Tpd       when alu_add | alu_add_unsigned,
                     a – b after Tpd       when alu_sub | alu_sub_unsigned,
                     a and b after Tpd     when alu_and,
                     a or b after Tpd      when alu_or,
                     a after Tpd           when alu_pass_a;
-------------------------------------------------------------------
alu : process is
begin
     case alu_function is
          when alu_add | alu_add_unsigned =>  result <= a + b after Tpd;
          when alu_sub | alu_sub_unsigned =>  result <= a – b after Tpd;
          when alu_and                    =>  result <= a and b after Tpd;
          when alu_or                     =>  result <= a or b after Tpd;
          when alu_pass_a                 =>  result <= a after Tpd;
     end case;
     wait on alu_function, a, b;
end process alu;
```

Example: full adder

```
entity full_adder is
      port ( a, b, c_in : bit;  s, c_out : out bit );
end entity full_adder;

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

architecture truth_table of full_adder is
begin
      with bit_vector'(a, b, c_in) select
            (c_out, s) <= bit_vector'("00") when "000",
                          bit_vector'("01") when "001",
                          bit_vector'("01") when "010",
                          bit_vector'("10") when "011",
                          bit_vector'("01") when "100",
                          bit_vector'("10") when "101",
                          bit_vector'("10") when "110",
                          bit_vector'("11") when "111";
end architecture truth_table;
```

Concurrent assertion statement

```
concurrent_assertion_statement ⇐
    [ label : ]
    assert boolean_expression
        [ report expression ] [ severity expression ] ;
```

Example

```
entity S_R_flipflop is
      port ( s, r : in bit;  q, q_n : out bit );
end entity S_R_flipflop;
```

```
architecture functional of S_R_flipflop is
begin
      q <= '1' when s = '1' else
            '0' when r = '1';
      q_n <= '0' when s = '1' else
               '1' when r = '1';
      check : assert not (s = '1' and r = '1')
                     report "Incorrect use of S_R_flip_flop: s and r both '1'";
end architecture functional;
```

Equivalent process

```
check : process is
begin
      assert not (s = '1' and r = '1')
            report "Incorrect use of S_R_flip_flop: s and r both '1'";
      wait on s, r;
end process check;
```

Entity and passive processes

```
entity_declaration ⇐
    entity identifier is
        [ port ( port_interface_list ) ; ]
        { entity_declarative_item }
    [ begin
        { concurrent_assertion_statement
        | passive_concurrent_procedure_call_statement
        | passive_process_statement } ]
    end [ entity ] [ identifier ] ;
```

Entity and passive processes

```
entity S_R_flipflop is
  port ( s, r : in bit;  q, q_n : out bit );

begin

  check : assert not (s and r)
            report "Incorrect use of S_R_flip_flop: "
                   "s and r both '1'";

end entity S_R_flipflop;
```

Entity and passive processes

```
entity ROM is
  port ( address : in natural;
         data : out bit_vector(0 to 7);
         enable : in bit );

begin

  trace_reads : process (enable) is
  begin
    if enable then
      report "ROM read at time " & to_string(now)
             & " from address " & to_string(address);
    end if;
  end process trace_reads;

end entity ROM;
```

Component Instantiation

component_instantiation_statement ⇐
    *instantiation*_label :
        entity *entity*_name [(*architecture*_identifier)]
            [**port map** (port_association_list)] ;

port_association_list ⇐
    ([*port*_name =>] (*signal*_name | expression | **open**)) { , ... }

Example

```
entity DRAM_controller is
      port ( rd, wr, mem : in bit;
                  ras, cas, we, ready : out bit  );
end entity DRAM_controller;
```

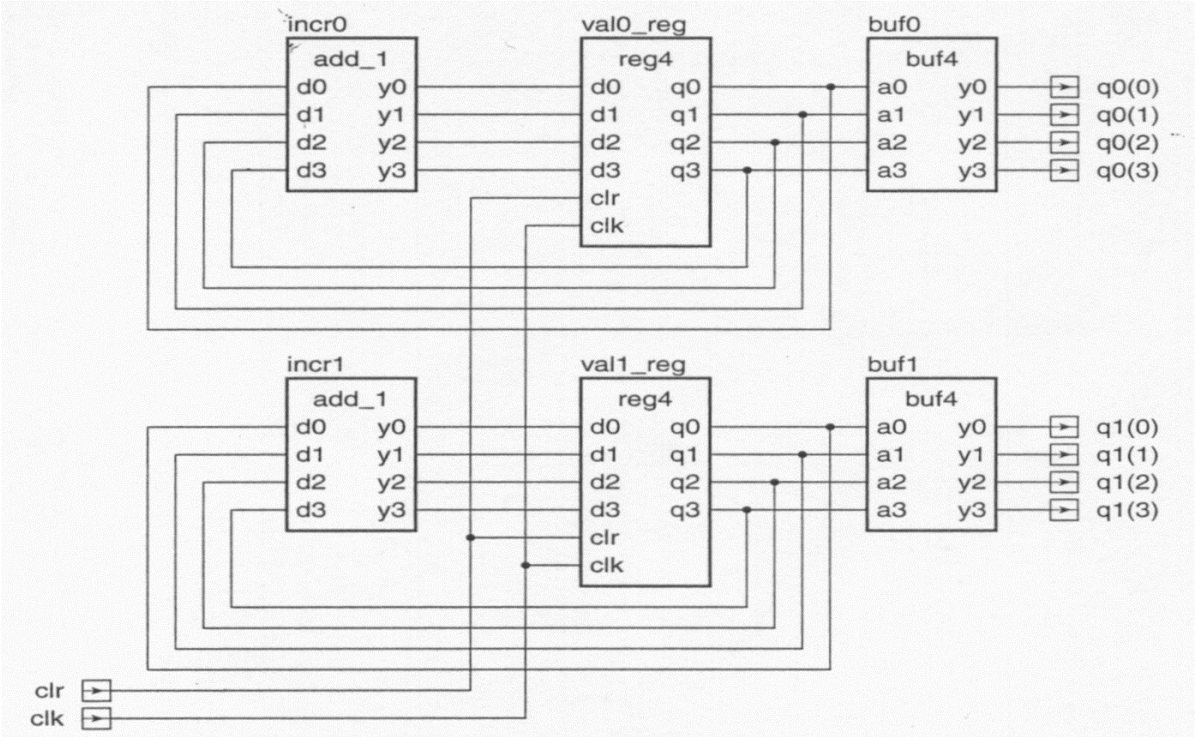
```
main_mem_controller : entity work.DRAM_controller(fpld)
      port map ( cpu_rd, cpu_wr, cpu_mem,
                        mem_ras, mem_cas, mem_we, cpu_rdy );
```

```
main_mem_controller : entity work.DRAM_controller(fpld)
      port map ( rd => cpu_rd, wr => cpu_wr,
                        mem => cpu_mem, ready => cpu_rdy,
                        ras => mem_ras, cas => mem_cas, we => mem_we );
```

A 4 bit register

```
entity reg4 is
      port ( clk, clr, d0, d1, d2, d3 : in bit;  q0, q1, q2, q3 : out bit );
end entity reg4;

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

architecture struct of reg4 is
begin
      bit0 : entity work.edge_triggered_Dff(behavioral)
            port map (d0, clk, clr, q0);
      bit1 : entity work.edge_triggered_Dff(behavioral)
            port map (d1, clk, clr, q1);
      bit2 : entity work.edge_triggered_Dff(behavioral)
            port map (d2, clk, clr, q2);
      bit3 : entity work.edge_triggered_Dff(behavioral)
            port map (d3, clk, clr, q3);
end architecture struct;
```

A two digit counter



A two digit counter

```
subtype digit is bit_vector(3 downto 0);
```

```
entity counter is
    port ( clk, clr : in bit;
           q0, q1 : out digit );
end entity counter;
-------------------------------------------------------------------
architecture registered of counter is
    signal current_val0, current_val1, next_val0, next_val1 : digit;
begin
    val0_reg : entity work.reg4(struct)
        port map ( d0 => next_val0(0), d1 => next_val0(1),
                   d2 => next_val0(2), d3 => next_val0(3),
                   q0 => current_val0(0), q1 => current_val0(1),
                   q2 => current_val0(2), q3 => current_val0(3),
                   clk => clk, clr => clr );
    val1_reg : entity work.reg4(struct)
        port map ( d0 => next_val1(0), d1 => next_val1(1),
                   d2 => next_val1(2), d3 => next_val1(3),
                   q0 => current_val1(0), q1 => current_val1(1),
                   q2 => current_val1(2), q3 => current_val1(3),
                   clk => clk, clr => clr );
    incr0 : entity work.add_1(boolean_eqn) ...;
    incr1 : entity work.add_1(boolean_eqn) ...;
    buf0 : entity work.buf4(basic) ...;
    buf1 : entity work.buf4(basic) ...;
end architecture registered;
```

Sub-element Association

```
type FIFO_status is record
        nearly_full, nearly_empty, full, empty : bit;
    end record FIFO_status;
```

```
DMA_buffer : entity work.FIFO
    port map ( ..., status.nearly_full => start_flush,
               status.nearly_empty => end_flush,
               status.full => DMA_buffer_full,
               status.empty => DMA_buffer_empty, ... );
```

Example: program status register

```
entity reg is
      port ( d : in bit_vector(7 downto 0);
             q : out bit_vector(7 downto 0);
             clk : in bit );
end entity reg;
-------------------------------------------------------------------
architecture RTL of microprocessor is
      signal interrupt_req : bit;
      signal interrupt_level : bit_vector(2 downto 0);
      signal carry_flag, negative_flag, overflow_flag, zero_flag : bit;
      signal program_status : bit_vector(7 downto 0);
      signal clk_PSR : bit;
      ...
begin
      PSR : entity work.reg
            port map ( d(7) => interrupt_req,
                       d(6 downto 4) => interrupt_level,
                       d(3) => carry_flag,  d(2) => negative_flag,
```

Sub-element association with unconstrained types

```
entity and_gate is
      port ( i : in bit_vector;  y : out bit );
end entity and_gate;
```

```
signal serial_select, write_en, bus_clk, serial_wr : bit;
```

```
serial_write_gate : entity work.and_gate
      port map ( i(1) => serial_select,
                 i(2) => write_en,
                 i(3) => bus_clk,
                 y => serial_wr );
```

Associations and expressions

```
entity mux4 is
      port ( i0, i1, i2, i3, sel0, sel1 : in bit;
             z : out bit );
end entity mux4;
```

```
a_mux : entity work.mux4
      port map ( sel0 => select_line, i0 => line0, i1 => line1,
                 z => result_line,
                 sel1 => '0', i2 => '1', i3 => '1' );
```

Non associated ports

```
entity and_or_inv is
      port ( a1, a2, b1, b2 : in bit := '1';
                  y : out bit );
end entity and_or_inv;
```

```
f_cell : entity work.and_or_inv
      port map ( a1 => A, a2 => B, b1 => C, b2 => open, y => F );
```

Non associated ports

- With mode **out**, the output value is simply ignored.
- With mode **inout**, the value used inside the model is the value sent to the port.
- If suffices to omit the port name from the association list to say that a port must not be associated to any signal or expression.

Non associated ports: examples

```
entity and3 is
    port ( a, b, c : in bit := '1';
           z, not_z : out bit);
end entity and3;
```

```
g1 : entity work.and3 port map ( a => s1, b => s2, not_z => ctrl1 );
```

```
g1 : entity work.and3 port map ( a => s1, b => s2, not_z => ctrl1,
                                 c => open, z => open );
```

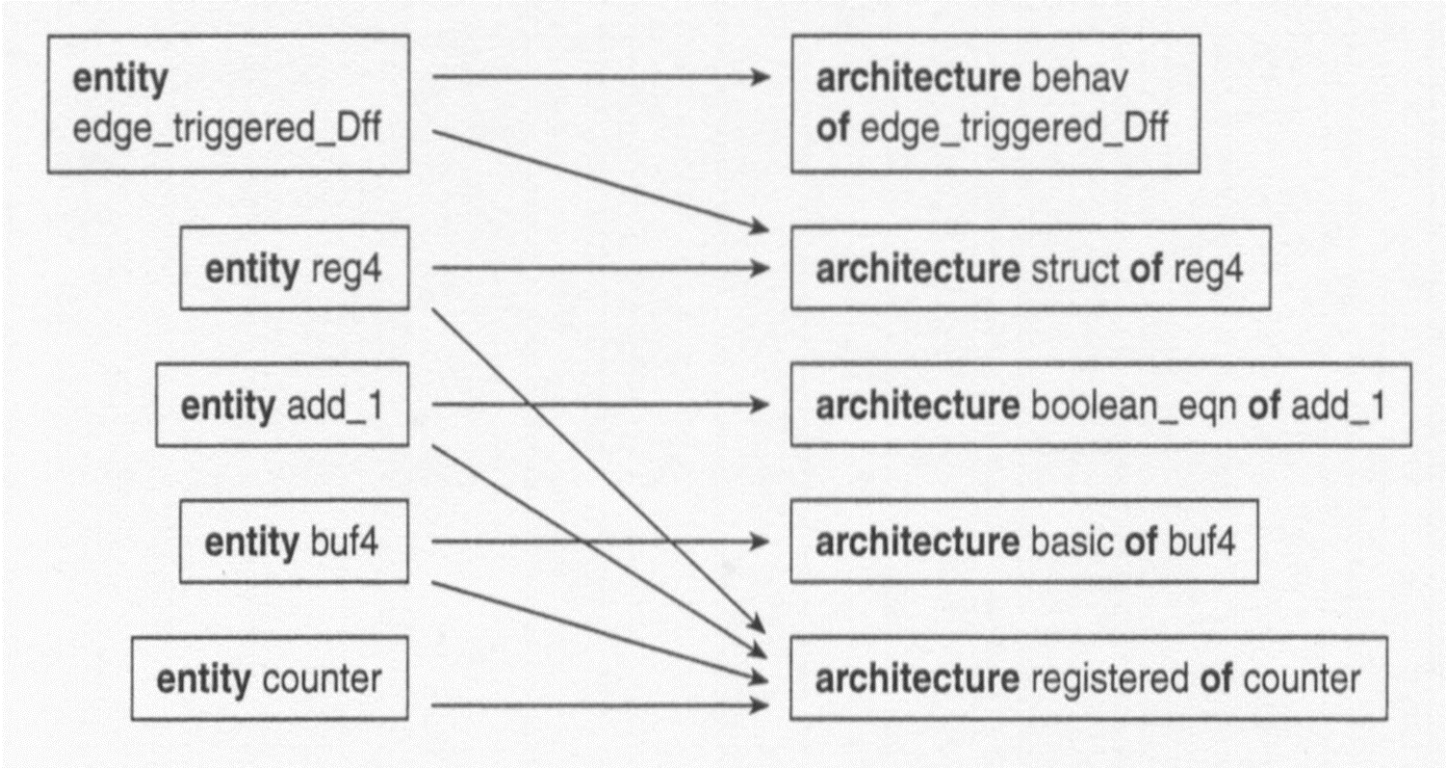
Design processing

- For simulation or synthesis.
- We talk about:
  - Analysis.
  - Elaboration.
  - Execution.

Analysis

- Analyzer
  - Checks syntax and semantics of the VHDL module.
  - Translates a VHDL description in some internal description.
  - Every entity/architecture pair (called, *“design unit”*) is processed and transformed into a “*library unit*”, placed in a design library.
- Design library
  - It is the place the library units are stored.
  - Example: the current work library *work.*
- The dependences between the design units force a certain analysis order.
- We divide the *design units* between:
  - *Primary units* (e.g., the entity declaration).
  - *Secondary units* (e.g., the architecture).

Example


entity
edge_triggered_Dff
architecture behav
of edge_triggered_Dff
entity reg4
architecture struct of reg4
entity add_1
architecture boolean_eqn of add_1
entity buf4
architecture basic of buf4
entity counter
architecture registered of counter

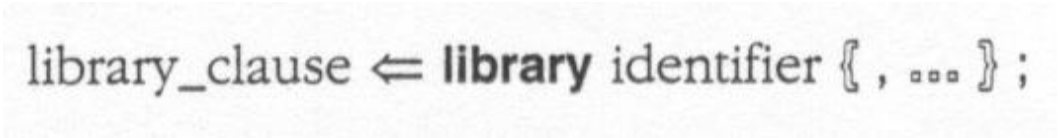
Example

- A possible analysis order:

```
entity edge_triggered_Dff
architecture behav of edge_triggered_Dff

entity reg4
architecture struct of reg4

entity add_1
architecture boolean_eqn of add_1

entity buf4
architecture basic of buf

entity counter
architecture registered of counter
```

Design Libraries

- It is the place where the design units are saved.
- Current work library: **work**
- All other libraries are called: *“resource libraries”.*
- Library clause:



Example

```
library widget_cells, wasp_lib;
architecture cell_based of filter is
    -- declaration of signals, etc
    ...
begin
    clk_pad : entity wasp_lib.in_pad
        port map ( i => clk, z => filter_clk );
    accum : entity widget_cells.reg32
        port map ( en => accum_en, clk => filter_clk, d => sum,
                   q => result );
    alu : entity work.adder
        port map ( a => alu_op1, b => alu_op2, y => sum, c => carry );
    -- other component instantiations
    ...
end architecture cell_based;
```

Use clause

use_clause ⇐ **use** selected_name { , ... } ;

selected_name ⇐ name . (identifier | **all**)

```
library widget_cells, wasp_lib;
use widget_cells.reg32;
```

```
accum : entity reg32
    port map ( en => accum_en, clk => filter_clk, d => sum,
               q => result );
```

```
use wasp_lib.all;
```

VHDL 2008: context declaration

```
context_declaration ⇐
    context identifier is
        { library_clause ‖ use_clause ‖ context_reference }
    end [ context ] [ identifier ] ;

context_reference ⇐
    context selected_name { , ... } ;
```

Example:

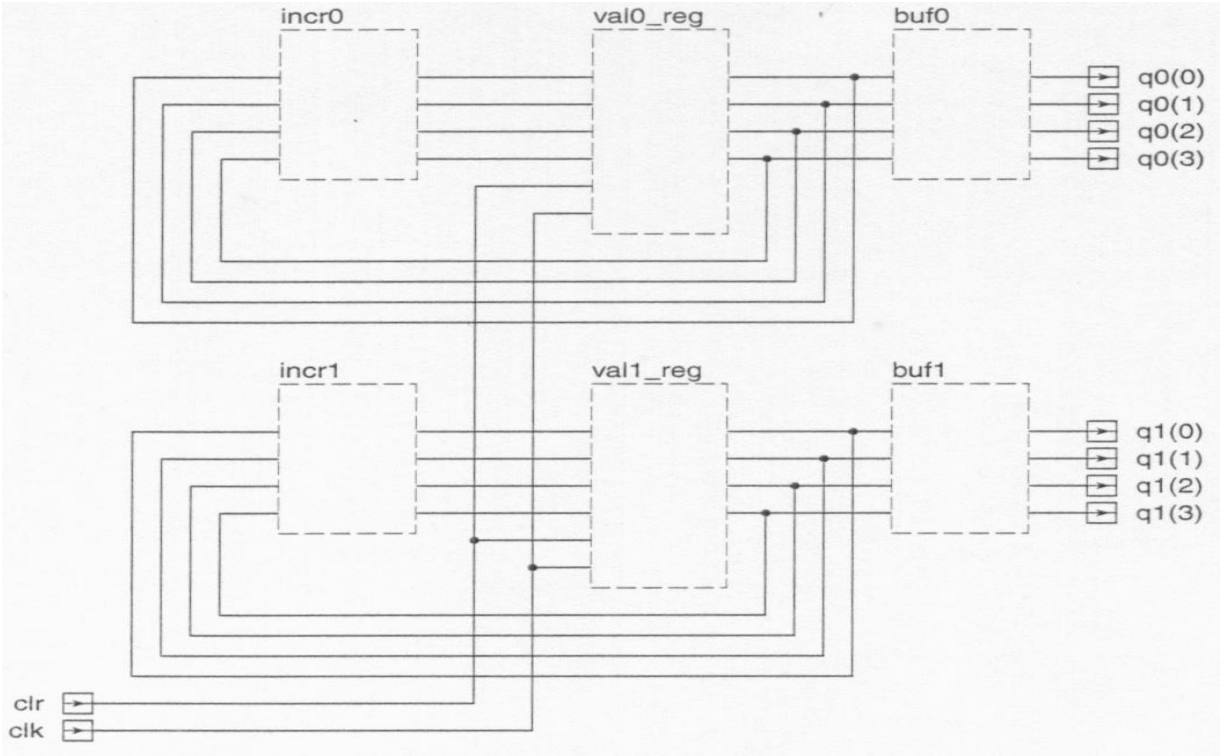
```
context widget_context is
  library ieee;
  use ieee.std_logic_1164.all;
  use widget_lib.all;
end context widget_context;
```

```
library widget_lib;
context widget_lib.widget_context;
entity sample is
  ...
end entity sample;
```

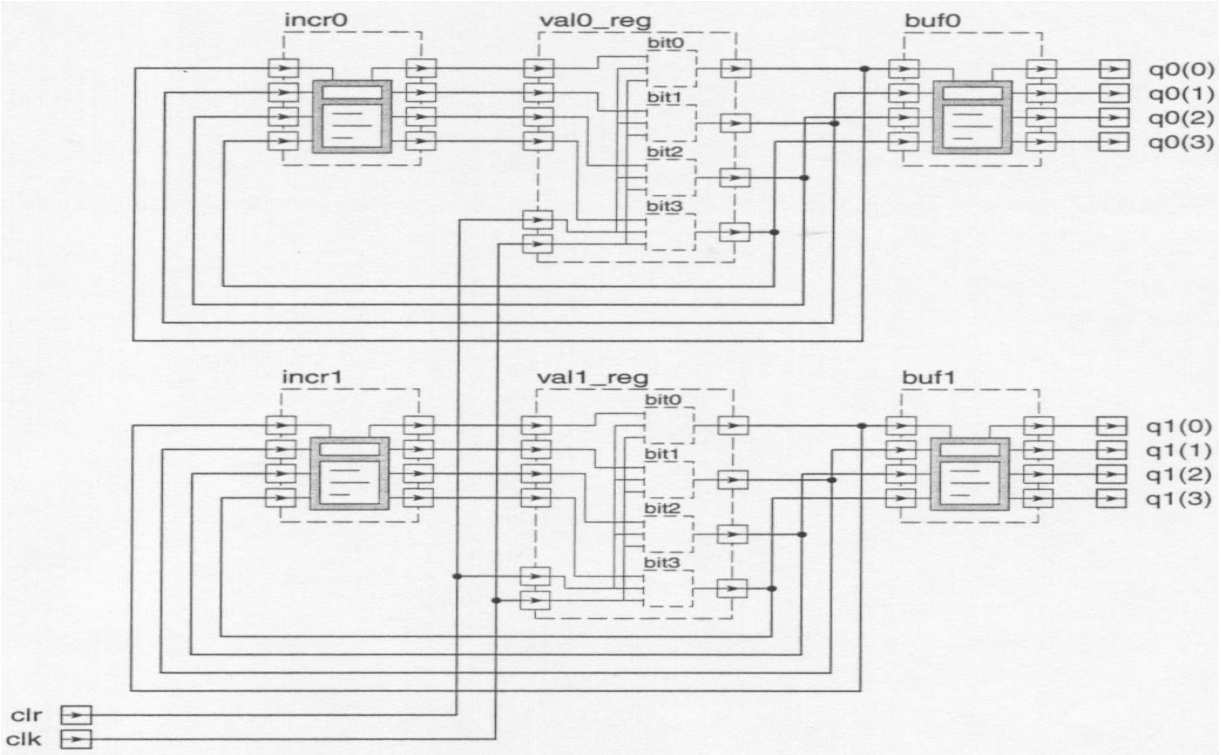
Elaboration

- If gives flesh to the design, producing a collection of processes interconnected by nets.
- The elaboration is a recursive process.
- It starts from the top level of our design and it replaces all instantiated entities with the corresponding architecture.
- It repeats this operation till the system is described only in term of processes and signals.

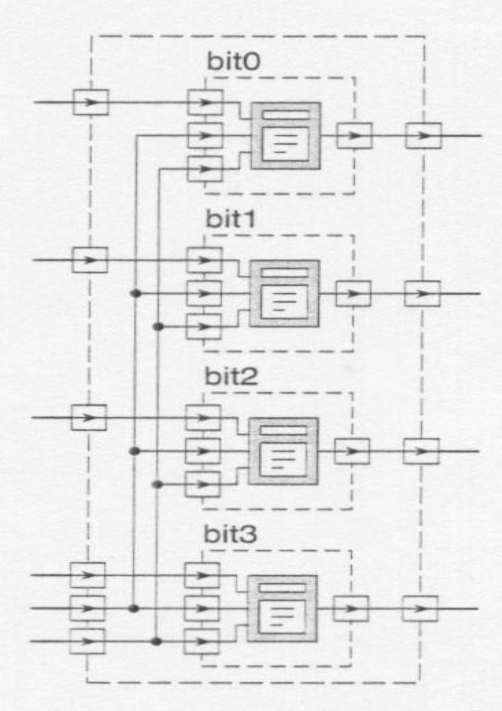
Elaboration



Elaboration



Elaboration



Elaboration

- The ports of the processed entity are created.
- The corresponding architecture is elaborated:
  - The signal declared in the architecture are created,
  - For each instantiated components the ports are created and connected to signals and ports of the architecture.
- The architectures of the instantiated components are recursively elaborated till we reach a description composed only by processes that interacts by means of signals.
- Processes are also elaborated:
  - A memory element is created for every variable declaration,
  - A driver is created for each signal assignment.

Execution

- Initialization phase.
  - Simulation time is set to 0.
  - Initialization of all signal drivers and memory elements.
  - All processes are executed one time till they stop on a **wait** statement.
- Simulation cycle.
  - The internal clock is advanced at the time of the first transaction or time-out.
  - The processes where a time-out has occurred and the processes sensitive to signals where an event has occurred are resumed and executed till they stop on a **wait** statement.

See:

- Peter Ashenden, «The designers' guide to VHDL» Morgan Kaufmann,
  - Chapter 5