UNIVERSITÀ
DEGLI STUDI DI TRIESTE

dipartimento
di ingegneria
e architettura

07 – Subprograms

A.Carini – Progettazione di sistemi elettronici

Procedures

- First a procedure must be declared, then elsewhere it will be called.
- Procedure declaration:

```
subprogram_body ⇐
    procedure identifier [ ( parameter_interface_list ) ] is
        { subprogram_declarative_part }
    begin
        { sequential_statement }
    end [ procedure ] [ identifier ] ;
```

Example of declaration

```
procedure average_samples is
    variable total : real := 0.0;
begin
    assert samples'length > 0 severity failure;
    for index in samples'range loop
        total := total + samples(index);
    end loop;
    average := total / real(samples'length);
end procedure average_samples;
```

- The local variables are newly created and initialized at every procedure call.

Procedure call statement

- Without parameters:

procedure_call_statement ⇐ ⟦ label : ⟧ *procedure*_name ;

- Example:

```
average_samples;
```

Procedures

- Can be declared in many places:
  - In the declarative part of an architecture.
    - They can be called by all processes of the architecture body.
  - In the declarative part of a process.
    - They can be called only inside that process.
  - In the declarative part of a procedure.
    - They can be called only inside that procedure.
  - And in other places we will study later.

Example

```
architecture rtl of control_processor is
    type func_code is (add, subtract);
    signal op1, op2, dest : integer;
    signal Z_flag : boolean;
    signal func : func_code;
    ...
begin
    alu : process is
        procedure do_arith_op is
            variable result : integer;
        begin
            case func is
                when add =>
                    result := op1 + op2;
                when subtract =>
                    result := op1 – op2;
            end case;
            dest  <=  result after Tpd;
            Z_flag  <=  result = 0 after Tpd;
        end procedure do_arith_op;
    begin
        ...
        do_arith_op;
        ...
    end process alu;
    ...
end architecture rtl;
```

Example: instruction fetch

```
instruction_interpreter : process is
    variable mem_address_reg, mem_data_reg,
             prog_counter, instr_reg, accumulator, index_reg : word;
    ...
    procedure read_memory is
    begin
        address_bus <= mem_address_reg;
        mem_read <= '1';
        mem_request <= '1';
        wait until mem_ready = '1';
        mem_data_reg := data_bus_in;
        mem_request <= '0';
        wait until mem_ready = '0';
    end procedure read_memory;
begin
    ...     -- initialization
    loop
        -- fetch next instruction
        mem_address_reg := prog_counter;
        read_memory;                    -- call procedure
        instr_reg := mem_data_reg;
        ...
        case opcode is
            ...
            when load_mem =>
                mem_address_reg := index_reg + displacement;
                read_memory;     -- call procedure
                accumulator := mem_data_reg;
            ...
        end case;
    end loop;
end process instruction_interpreter;
```

Example: a procedure calling a procedure

```
control_sequencer : process is
    procedure control_write_back is
    begin
        wait until phase1 = '1';
        reg_file_write_en <= '1';
        wait until phase2 = '0';
        reg_file_write_en <= '0';
    end procedure control_write_back;
    procedure control_arith_op is
    begin
        wait until phase1 = '1';
        A_reg_out_en <= '1';
        B_reg_out_en <= '1';
        wait until phase1 = '0';
        A_reg_out_en <= '0';
        B_reg_out_en <= '0';
        wait until phase2 = '1';
        C_reg_load_en <= '1';
        wait until phase2 = '0';
        C_reg_load_en <= '0';
        control_write_back;      -- call procedure
    end procedure control_arith_op;
    ...
begin
    ...
    control_arith_op;      -- call procedure
    ...
end process control_sequencer;
```

Return statement in a procedure

return_statement ⇐ ⟦ label : ⟧ **return** ;

Example: instruction fetch

```
instruction_interpreter : process is
    ...
    procedure read_memory is
    begin
        address_bus <= mem_address_reg;
        mem_read <= '1';
        mem_request <= '1';
        wait until mem_ready = '1' or reset = '1';
        if reset = '1' then
            return;
        end if;
        mem_data_reg := data_bus_in;
        mem_request <= '0';
        wait until mem_ready = '0';
    end procedure read_memory;
begin
    ...        -- initialization
    loop
        ...
        read_memory;
        exit when reset = '1';
        ...
    end loop;
end process instruction_interpreter;
```

Parameterized procedures

- Can execute the algorithm by using different data objects or different values at every call.
- When declaring a parameterized procedure, we include after the procedure name a parameter interface list:

```
interface_list ⇐
    ( [ constant | variable | signal ]
        identifier { , ... } : [ mode ] subtype_indication
                                    [ := static_expression ] ) { ; ... }

mode ⇐ in | out | inout
```

- The parameters of the interface list are called formal parameters. They are the place holders for the actual parameters we pass with the procedure call.

Example

```
procedure do_arith_op ( op : in func_code ) is
    variable result : integer;
begin
    case op is
        when add =>
            result := op1 + op2;
        when subtract =>
            result := op1 – op2;
    end case;
    dest  <=  result after Tpd;
    Z_flag  <=  result = 0 after Tpd;
end procedure do_arith_op;
```

- The procedure can read the parameter but cannot modify it.
- The parameter is considered a **constant** (default parameter class for mode **in**).
- Default mode is **in**. Thus, it is equivalent to:

```
procedure do_arith_op ( op : func_code ) is …
```

Parameterized procedure calls

- Passing constants:

```
do_arith_op ( add );
```

- Or passing the value of signals or variables, or expressions:

```
do_arith_op ( func );
```

Out mode

```
procedure addu ( a, b : in word32;
                 result : out word32;  overflow : out boolean ) is
    variable sum : word32;
    variable carry : bit := '0';
begin
    for index in sum'reverse_range loop
        sum(index) := a(index) xor b(index) xor carry;
        carry := ( a(index) and b(index) ) or ( carry and ( a(index) xor b(index) ) );
    end loop;
    result := sum;
    overflow := carry = '1';
end procedure addu;
```

```
variable PC, next_PC : word32;
variable overflow_flag : boolean;
...
addu ( PC, X"0000_0004", next_PC, overflow_flag);
```

Default class: **variable**

Out mode

- The parameters with **out** mode are assumed to belong to the class **variable**.
- We can omit the keyword **variable** or we can include it.
- Mode **out** indicates that the only way the procedure can use the formal parameters is for updating them with a variable assignment. It cannot read them.
- The actual parameter passed to the procedure must be a variable.

- Things have changed with **VHDL 2008:**
- The parameters with mode **out** can now be read by the procedure.
- At each procedure call, the **out** parameters are reinitialized with their default value.

Inout mode

- It is a combination of **in** and **out**.
- It is used for objects that must be both read and written.
- The objects are assumed to belong to the class **variable** and the actual parameter passed to the procedure must be a variable.
- At each procedure call, the value of the variable will be used to initialize the formal parameter.
- The formal parameter will be assigned by the procedure and when the procedure return its value will be copied in the actual parameter.

Inout mode

```
procedure negate ( a : inout word32 ) is
    variable carry_in : bit := '1';
    variable carry_out : bit;
begin
    a := not a;
    for index in a'reverse_range loop
        carry_out :=  a(index) and carry_in;
        a(index) := a(index) xor carry_in;
        carry_in := carry_out;
    end loop;
end procedure negate;
```

```
variable op1 : word32;
...
negate ( op1 );
```

Signal parameters

- Class **signal** indicates a signal is passed to the procedure.
- Mode can be **in, out, inout.**
- With mode **in** the entire signal object is passed to the procedure.
- After a **wait** statement, the value of the formal parameter of the class *signal* can change, on the contrary it does not change in case of parameters of class **constant** or **variable**.

Example

```
architecture behavioral of receiver is
    ...      -- type declarations, etc
    signal recovered_data : bit;
    signal recovered_clock : bit;
    ...
    procedure receive_packet ( signal rx_data : in bit;
                               signal rx_clock : in bit;
                               data_buffer : out packet_array ) is
    begin
        for index in packet_index_range loop
            wait until rx_clock = '1';
            data_buffer(index) := rx_data;
        end loop;
    end procedure receive_packet;
begin
    packet_assembler : process is
        variable packet : packet_array;
    begin
        ...
        receive_packet ( recovered_data, recovered_clock, packet );
        ...
    end process packet_assembler;
    ...
end architecture behavioral;
```

Signal parameters

- With mode **out**, a reference to the driver of the actual parameter signal is passed to the procedure.
- When the formal parameter is assigned, a transaction is scheduled on the driver of the actual parameter.

Example: pulse train generator

```
library ieee;  use ieee.std_logic_1164.all;
architecture top_level of signal_generator is
    signal raw_signal : std_ulogic;
    ...
    procedure generate_pulse_train ( width, separation : in delay_length;
                                     number : in natural;
                                     signal s : out std_ulogic ) is
    begin
        for count in 1 to number loop
            s <= '1', '0' after width;
            wait for width + separation;
        end loop;
    end procedure generate_pulse_train;
begin
    raw_signal_generator : process is
    begin
        ...
        generate_pulse_train ( width => period / 2,
                               separation => period – period / 2,
                               number => pulse_count,
                               s => raw_signal );
        ...
    end process raw_signal_generator;
    ...
end architecture top_level;
```

Signal parameters

- With mode **inout**, the actual parameter signal and a reference to the driver of the actual parameter signal are both passed to the procedure.
- When the formal parameter is read, the read value is the current value of the actual parameter signal.
- When the formal parameter is assigned, a transaction is scheduled on the driver of the actual parameter.

Default values

```
procedure increment ( a : inout word32;  by : in word32 := X"0000_0001" ) is
    variable sum : word32;
    variable carry : bit := '0';
begin
    for index in a'reverse_range loop
        sum(index) := a(index) xor by(index) xor carry;
        carry := ( a(index) and by(index) ) or ( carry and ( a(index) xor by(index) ) );
    end loop;
    a := sum;
end procedure increment;
```

- Possible use:

```
increment(count, X"0000_0004");

increment(count);
```

Unconstrained array parameters

```
procedure find_first_set ( v : in bit_vector;
                           found : out boolean;
                           first_set_index : out natural ) is
begin
    for index in v'range loop
        if v(index) = '1' then
            found := true;
            first_set_index := index;
            return;
        end if;
    end loop;
    found := false;
end procedure find_first_set;
```

Unconstrained array parameters

```
variable int_req : bit_vector (7 downto 0);
variable top_priority : natural;
variable int_pending : boolean;
...
find_first_set ( int_req, int_pending, top_priority );
```

```
variable free_block_map : bit_vector(0 to block_count–1);
variable first_free_block : natural;
variable free_block_found : boolean;
...
find_first_set ( free_block_map, free_block_found, first_free_block );
```

Example

- Comparison between two numbers in two's complement form:

```
procedure bv_lt ( bv1, bv2 : in bit_vector;  result : out boolean ) is
      variable tmp1 : bit_vector(bv1'range) := bv1;
      variable tmp2 : bit_vector(bv2'range) := bv2;
begin
      tmp1(tmp1'left) := not tmp1(tmp1'left);
      tmp2(tmp2'left) := not tmp2(tmp2'left);
      result :=  tmp1 < tmp2;
end procedure bv_lt;
```

Summary of procedures

- Declaration
  - For each parameter we specify:
    - The class **constant**, **variable**, or **signal**.
    - The name of the formal parameter.
    - The mode **in, out, inout.**
    - The type or subtype of the parameter.
    - The default value

```
procedure_call_statement ⇐
    [[ label : ]] procedure_name [[ ( parameter_association_list ) ]] ;

parameter_association_list ⇐
    ( [[ parameter_name => ]]
        expression | signal_name | variable_name | open ) { , ... }
```

Example of notation

```
procedure p ( f1 : in t1;  f2 : in t2;  f3 : out t3;  f4 : in t4 := v4 ) is
begin
      ...
end procedure p;
```

```
p ( val1, val2, var3, val4 );
p ( f1 => val1, f2 => val2, f4 => val4, f3 => var3 );
p ( val1, val2, f4 => open, f3 => var3 );
p ( val1, val2, var3 );
```

Concurrent procedure call statements

concurrent_procedure_call_statement ⇐
    [[label :]] *procedure*_name [[(*parameter*_association_list)]] ;

- Example:

```
call_proc : p ( s1, s2, val1 );
```

```
call_proc : process is
begin
    p ( s1, s2, val1 );
    wait on s1, s2;
end process call_proc;
```

Example

```
procedure check_setup ( signal data, clock : in bit;
                        constant Tsu : in time ) is
begin
    if clock'event and clock = '1' then
        assert data'last_event >= Tsu
            report "setup time violation" severity error;
    end if;
end procedure check_setup;
```

Call example:

```
check_ready_setup : check_setup ( data => ready, clock => phi2,
                                  Tsu => Tsu_rdy_clk );
```

Example

```
procedure generate_clock ( signal clk : out std_ulogic;
                           constant Tperiod, Tpulse, Tphase : in time ) is
begin
    wait for Tphase;
    loop
        clk <= '1', '0' after Tpulse;
        wait for Tperiod;
    end loop;
end procedure generate_clock;
```

Call example:

```
signal phi1, phi2 : std_ulogic := '0';
...
gen_phi1 : generate_clock ( phi1, Tperiod => 50 ns, Tpulse => 20 ns,
                            Tphase => 0 ns );
gen_phi2 : generate_clock ( phi2, Tperiod => 50 ns, Tpulse => 20 ns,
                            Tphase => 25 ns );
```

Functions: return and function call

- Functions are a generalization of operators in expressions and define new operations that can be used in expressions.
- They are a collection of sequential statements that compute and return a result.

```
subprogram_body ⇐
    [ pure | impure ]
    function identifier [ ( parameter_interface_list ) ] return type_mark is
        { subprogram_declarative_item }
    begin
        { sequential_statement }
    end [ function ] [ identifier ] ;
```

Return and function call

- The result is returned with:

return_statement ⇐ ⟦ label : ⟧ **return** expression ;

- Syntax of a function call:

function_call ⇐ *function*_name ⟦ (*parameter*_association_list) ⟧

Example

```
function limit ( value, min, max : integer ) return integer is
begin
      if value > max then
            return max;
      elsif value < min then
            return min;
      else
            return value;
      end if;
end function limit;
```

- Call example:

```
new_temperature := limit ( current_temperature + increment, 10, 100 );
```

```
new_motor_speed := old_motor_speed + scale_factor * limit ( error, –10, +10 );
```

Example

```
function bv_to_natural ( bv : in bit_vector ) return natural is
    variable result : natural := 0;
begin
    for index in bv'range loop
        result := result * 2 + bit'pos(bv(index));
    end loop;
    return result;
end function bv_to_natural;
```

- It can be use for an address conversion:

```
type rom_array is array (natural range 0 to rom_size–1)
                                of bit_vector(0 to word_size–1);
variable rom_data : rom_array;
```

```
data <= rom_data ( bv_to_natural(address) ) after Taccess;
```

Functional modeling

- We can use the VHDL functions in concurrent signal assignments.
- For example, given the function:

```
function bv_add ( bv1, bv2 : in bit_vector ) return bit_vector is
begin
    ...
end function bv_add;
```

- And the signals:

```
signal source1, source2, sum : bit_vector(0 to 31);
```

```
adder : sum <= bv_add(source1, source2) after T_delay_adder;
```

Pure and *impure* functions

- Functions are called *pure* (and may be so declared using the keyword **pure**) if they do not refer to variables or signals declared by their parents (i.e., by any process, procedure or architecture where the function is nested).
- Functions are called *impure* (and must be so declared with the keyword **impure**) if they do refer to variables or signals of their parents.

impure function example

```
network_driver : process is

  constant seq_modulo : natural := 2**5;
  subtype seq_number is natural range 0 to seq_modulo-1;
  variable next_seq_number : seq_number := 0;
  ...

  impure function generate_seq_number return seq_number is
    variable number : seq_number;
  begin
    number := next_seq_number;
    next_seq_number := (next_seq_number + 1) mod seq_modulo;
    return number;
  end function generate_seq_number;

begin  -- network_driver
  ...
  new_header := pkt_header'( dest => target_host_id,
                             src => my_host_id,
                             pkt_type => control_pkt,
                             seq => generate_seq_number );
  ...
end process network_driver;
```

The function now

```
impure function now return delay_length;
```

- Example: for checking the *hold time*

```
hold_time_checker : process ( clk, d ) is
    variable last_clk_edge_time : time := 0 fs;
begin
    if clk'event and clk = '1' then
        last_clk_edge_time := now;
    end if;
    if d'event then
        assert now – last_clk_edge_time >= Thold_d_clk
            report "hold time violation";
    end if;
end process hold_time_checker;
```

Overloading

- We can define two subprograms with the same identifier, provided that they have different number or different type of formal parameters.
- When we call one of these subprograms, it is the number or the type of the parameters that determines which subprogram we have called.

Overloading example

```
procedure increment ( a : inout integer;   n : in integer := 1 ) is ...
procedure increment ( a : inout bit_vector;  n : in bit_vector := B"1" ) is ...
procedure increment ( a : inout bit_vector;  n : in integer := 1 ) is ...
```

```
variable count_int : integer := 2;
variable count_bv : bit_vector (15 downto 0) := X"0002";
```

```
increment ( count_int, 2 );
increment ( count_int );
increment ( count_bv, X"0002");
increment ( count_bv, 1 );
increment ( count_bv );
```

Overloading of operator symbols

- The operators are specialized functions with a convenient notation for calling.
- The VHDL language allows to define new functions using the symbols of the operators (“+”, “-”, “**and**”, “**or**”, etc.).

Overloading examples

```
function "+" ( left, right : in bit_vector ) return bit_vector is
begin
    ...
end function "+";
```

```
variable addr_reg : bit_vector(31 downto 0);
...
addr_reg := addr_reg + X"0000_0004";
```

```
function "abs" ( right : in bit_vector ) return bit_vector is
begin
    ...
end function "abs";
```

```
variable accumulator : bit_vector(31 downto 0);
...
accumulator := abs accumulator;
```

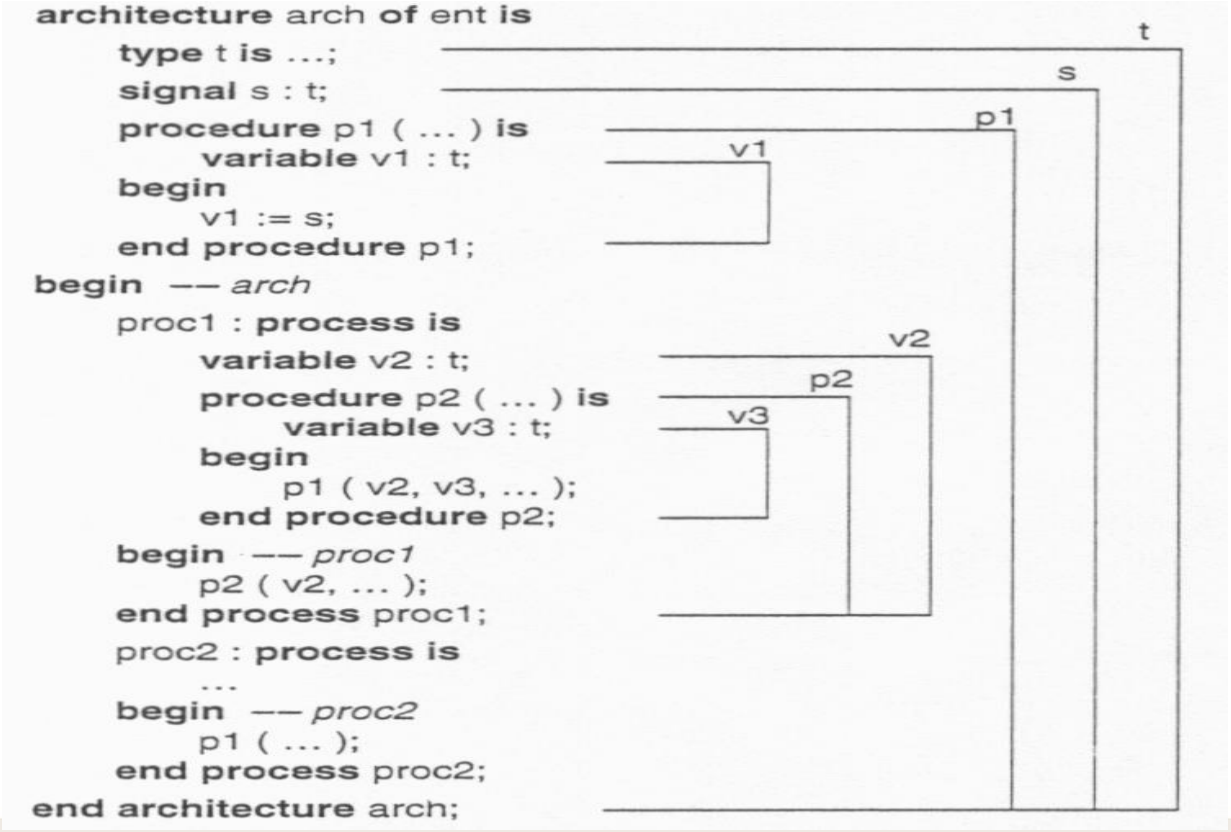
Overloading examples

```
library ieee;  use ieee.std_logic_1164.all;
entity reg_ctrl is
      port ( reg_addr_decoded, rd, wr, io_en, cpu_clk : in std_ulogic;
               reg_rd, reg_wr : out std_ulogic );
end entity reg_ctrl;
-------------------------------------------------------------------
architecture bool_eqn of reg_ctrl is
begin
      rd_ctrl : reg_rd <= reg_addr_decoded and rd and io_en;
      rw_ctrl : reg_wr <= reg_addr_decoded and wr and io_en and not cpu_clk;
end architecture bool_eqn;
```

Visibility of declarations

- It is the text region where we can refer to a declared name.
- Processes, subprograms, architectures are divided in a declarative part and a body composed by statements.
- Every declaration written in a declarative part is visible from the end of the declaration itself to the end of the corresponding body. This is the only region where we will be able to refer to the declared element.

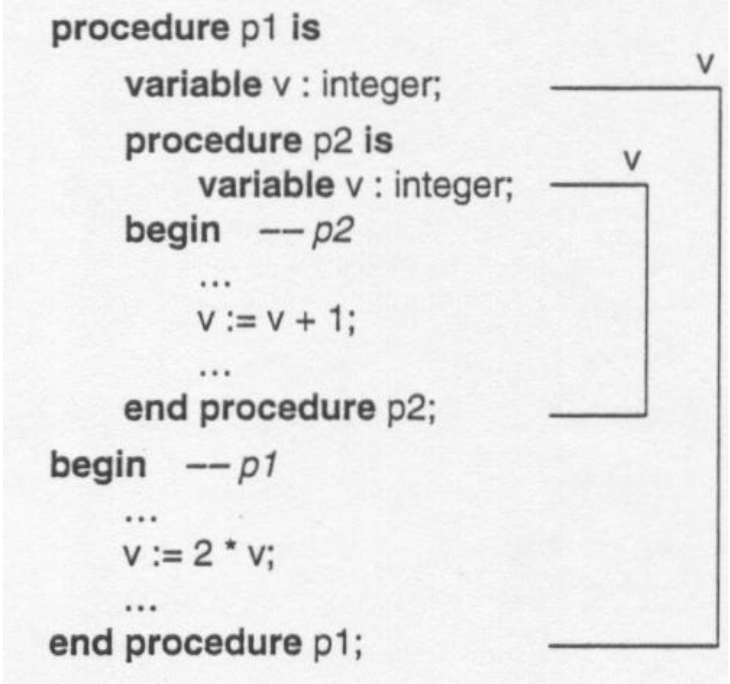
Example of visibility regions



Example with nested procedures

```
architecture behavioral of cache is
begin
    behavior : process is
        . . .
        procedure read_block ( start_address : natural;  entry : out cache_block ) is
            variable memory_address_reg : natural;
            variable memory_data_reg : word;
            procedure read_memory_word is
            begin
                mem_addr <= memory_address_reg;  mem_read <= '1';
                wait until mem_ack = '1';
                memory_data_reg := mem_data_in;  mem_read <= '0';
                wait until mem_ack = '0';
            end procedure read_memory_word;
        begin  -- read_block
            for offset in 0 to block_size - 1 loop
                memory_address_reg := start_address + offset;
                read_memory_word;
                entry(offset) := memory_data_reg;
            end loop;
        end procedure read_block;
    begin  -- behavior
        . . .
        read_block ( miss_base_address, data_store(entry_index) );
        . . .
    end process behavior;
end architecture behavioral;
```

Example of hiding



See:

- Peter Ashenden, «The designers' guide to VHDL» Morgan Kaufmann,
  - Chapter 6