

VHDL Subprograms

Subprograms

- Two types of subprograms
 - **Function**
 - Called within expression
 - Compute and return “**single**” output value
 - **Procedure**
 - Can be sequential or concurrent
 - Useful in partitioning large behavioral VHDL codes
 - Compute and return “**zero or more**” output values

Return Statement

- Operations in subprograms executed by sequential statements
- Subprograms finished by return statement
 - Termination of subprogram
 - Return control over program execution to calling program
 - Functions – require return statement
 - Value of expression returned to calling program

Functions - Format

1. Not necessarily variables
(if no type mentioned then Constant assumed)
2. Each parameter of in mode
(if mode not explicitly mentioned then in assumed)

Declaration of items used locally

Used to pass function result to function caller

```
function identifier
[parameter list] return object type is
subprogram declarative items
begin
sequential statements
return expression
end [function] identifier;
```

4b-binary-to-integer

function name

type of returned result

4-b binary → integer

Argument Class of Parameters for Subprograms

- Every parameter of subprogram has associated argument class
- Argument class
 - Determines what information is passed to subprograms
 - Determines whether passed parameter is constant, signal or variable
 - Three kinds of argument classes:
 - Constants, variables and signals

Example

```

library IEEE;
use IEEE.std_logic_1164.all;
entity dff is
  port(d, clock: in std_logic; q, qbar: out std_logic);
end entity dff;
architecture beh of dff is
  function rising_edge(signal clock:std_logic) return Boolean is
    variable edge: Boolean:=FALSE;
begin
  edge:=(clock = '1' and clock'event);
  return (edge);
end function rising_edge;
begin
  output: process is
  begin
    wait until (rising_edge(clock));
    q <= d after 5 ns;
    qbar <= not d after 5 ns;
  end process output;
end architecture beh;

```

Diagram illustrating the behavior of the DFF entity:

- Entity Declaration:** Shows the entity `dff` with ports `d`, `clock`, `q`, and `qbar`.
- Architecture Declaration:** Shows the architecture `beh` of `dff`.
- Function `rising_edge`:** A function that takes a signal `clock` as input and returns a Boolean value indicating if a rising edge has occurred.
- Process `output`:** A process that waits for a rising edge on the `clock` signal and then updates the `q` and `qbar` outputs based on the `d` input.
- Timing Diagrams:**
 - The top diagram shows a clock signal with three rising edges at 5 ns, 10 ns, and 15 ns. The `q` output is high between 5-10 ns and 10-15 ns, and low between 10-15 ns and 15-20 ns.
 - The bottom diagram shows the `rising_edge(clock)` signal, which is high whenever the clock signal transitions from 0 to 1. It has pulses at 5 ns, 10 ns, and 15 ns.

Annotations and highlights in the code:

- function name:** `rising_edge`
- parameter list:** `(signal clock:std_logic)`
- return Boolean:** `return Boolean`
- returned data type:** `Boolean`
- declarative region of architecture:** The region enclosed by the `begin` and `end architecture` keywords.
- no sensitivity list:** A note indicating that the process does not have a sensitivity list.
- clock signal timing:** A graph showing the `clock` signal with time markers at 5 ns, 10 ns, and 15 ns.
- process output timing:** A graph showing the `q` and `qbar` signals over time, corresponding to the rising edges of the clock.

① CLOCK'EVENT ② bin'RANGE

Example

③ bin'REVERSE RANGE

- A function to convert bit_vector to integer

```
FUNCTION to_integer (bin : BIT_VECTOR) RETURN INTEGER IS
  VARIABLE result: INTEGER;
BEGIN
  result := 0; x'Range
  FOR i IN bin'RANGE LOOP
    IF bin(i) = '1' THEN
      result := result + 2**i;
    END IF;
  END LOOP;
  RETURN result;
END to_integer;
```

- to_integer(x) is an integer

- Can be used anywhere an integer is expected

variable y: integer;

variable x: bit_vector(0 to 7);

y = to_integer(x) + 5;

① bin: BIT_VECTOR (0 to 7) ← Constrained input param.
② bin: BIT_VECTOR ← unconstrained
VHDL attributes BIT RANGE

unconstrained Vector Size
flexible

Functions have
declarative
and
body
parts

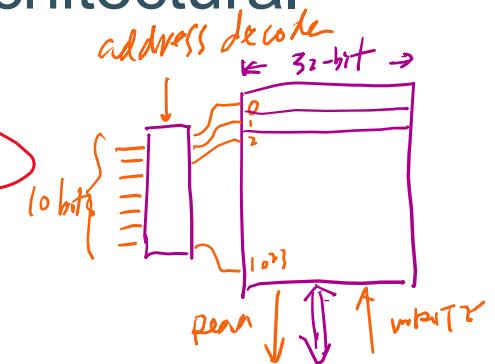
bin'REVERSE RANGE
7 down to 0
0 to 7

$$\begin{array}{r} 1+2*2=3 \\ \uparrow \\ 0+2*D=1 \end{array}$$

Ex: $x = 00010111$
to_integer(x)
 $y = 25 + 5 = 30$

Procedures

- Allowing on modularization of large architectural bodies
- Returning **zero or more output values**
 - **In:** only readable by procedure
 - **Out:** only writable
 - **Inout:** both readable and writable
 - Returning parameters achieved through using parameters of **out** and **inout**
 - Procedures may change value of one or more parameters passed to it



Example: Procedure for Addition of Two Unsigned Numbers

VHDL attributes
'RANGE
'REVERSE_RANGE

```

procedure add_unsigned(in1,in2: in bit_vector(7 downto 0);
                      sum: out bit_vector(7 downto 0); Cout: out boolean) is
variable sum_tmp: bit_vector(7 downto 0);
variable carry: bit := '0';
begin
  for i in sum_tmp'reverse_range loop
    sum_tmp(i) := in1(i) xor in2(i) xor carry;
    new carry := (in1(i) and in2(i)) or (carry and (in1(i))) or (carry and (in2(i)));
  end loop;
  sum := sum_tmp;
  Cout := carry = '1';
end procedure add_unsigned;

```

Can be bit
boolean
16

sum_tmp' RANGE = 7 downto 0
sum_tmp' REVERSE_RANGE = 0 to 7

Diagram illustrating the addition process:

- Inputs: $in1(7) \dots in1(0)$ and $in2(7) \dots in2(0)$
- Carry: $carry(7) \dots carry(0)$
- Sum: $sum(7) \dots sum(0)$
- Intermediate Result: $sum_tmp(7) \dots sum_tmp(0)$

The diagram shows the flow of data through the addition logic, with arrows indicating the flow from inputs to intermediate results and finally to the output sum and carry.

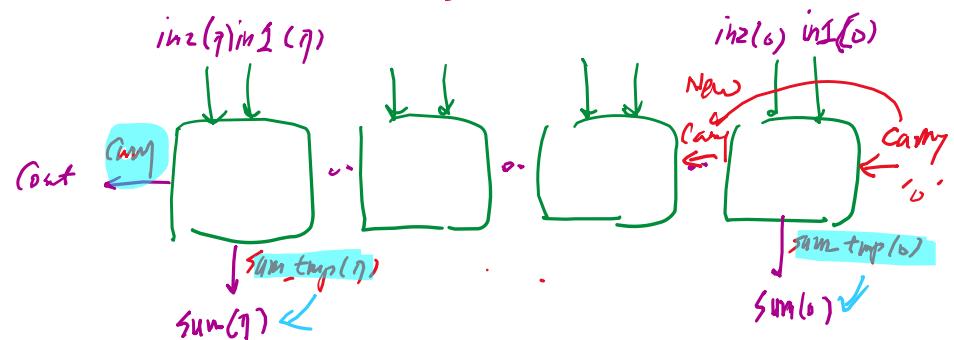
Note: The handwritten annotations and diagrams are part of the original image and do not have a formal structure; they serve as explanatory notes for the code example.

Example: Procedure for Addition of Two Unsigned Numbers

↑
no initial carry_in

```
procedure add_unsigned(in1,in2: in bit_vector(7 downto 0);
                      sum: out bit_vector(7 downto 0); Cout: out bit) is
variable sum_tmp: bit_vector(7 downto 0);
variable carry: bit := '0';
begin
  for i in sum_tmp'reverse_range loop
    sum_tmp(i) := in1(i) xor in2(i) xor carry;
    carry := (in1(i) and in2(i)) or (carry and (in1(i))) or (carry and (in2(i)));
  end loop;
  sum := sum_tmp;
  Cout := carry;
end procedure add_unsigned;
```

sum_tmp' REVERSE_RANGE = 0 to 7



Wait Statement in Procedures

- ❑ Wait statement allowed in procedure
 - “Wait” is forbidden in functions
- ❑ Functions used to compute values to be available instantaneously
 - Cannot wait
 - Not allowed to call procedure with wait statement from within function body