ECE381(CAD), Lecture 13:

Sub-programs in VHDL

Mehdi Modarressi

Department of Electrical and Computer Engineering,

University of Tehran

Pictures and examples are taken from the slides of "VHDL: Analysis & Modeling of Digital Systems" and also from "VHDL by Example"

Introduction

- Procedures and functions in VHDL
- Differences between procedures and functions
- Readings:
  1. “VHDL: Analysis & Modeling of Digital Systems”: Chapter 6.1 and 8.1
  2. “VHDL by Example”: Chapter 3
  3. “Designers Guide To VHDL”: chapter 7 and 11

Introduction

- Procedures/Subroutines/Functions in SW programming languages
  - The same functionality, in different places
  - Use function call instead of repeating the code
- VHDL equivalence:
  - *Procedures* and *Functions*

Subprograms

- The statements inside a subprogram are executed sequentially
- The same statements that exist in a process statement can be used in a subprogram
  - Such as loops and WAIT statements
- When a function is called, it returns the value in zero time
- Declaration and call:
  - First the subprogram is *declared*, then elsewhere it is *called*
- We can write a subprogram declaration in the declarative part of an architecture or the declarative part of a process

Functions

- Recall the comparator example:

GT Equation	a_gt_b = a . gt + b' . gt + a . b'
EQ Equation	a_eq_b = a . b . eq + a' . b' . eq
LT Equation	a_lt_b = b . lt + a' . lt + b . a'

- *a_gt_b* and *a_lt_b* have the same functionality but on different inputs

Functions

- Use function call instead of Boolean expressions to reuse the code

GT Equation	a_gt_b = a . gt + b' . gt + a . b'
EQ Equation	a_eq_b = a . b . eq + a' . b' . eq
LT Equation	a_lt_b = b . lt + a' . lt + b . a'

```
ARCHITECTURE functional OF bit_comparator IS
  FUNCTION fgl (w, x, gl : BIT) RETURN BIT IS
  BEGIN
    RETURN  (w AND gl) OR (NOT x  AND  gl) OR (w  AND  NOT x);
  END fgl;
  FUNCTION feq (w, x, eq : BIT) RETURN BIT IS
  BEGIN
    RETURN  (w AND x AND eq) OR (NOT w  AND  NOT x  AND  eq);
  END feq;
BEGIN
  a_gt_b <= fgl (a, b, gt) AFTER 12 NS;
  a_eq_b <= feq (a, b, eq) AFTER 12 NS;
  a_lt_b <= fgl (b, a, lt) AFTER 12 NS;
END functional;
```

Functions

- Two typical applications of functions in VHDL:
  - Conversion: convert different types
  - Resolution: resolve the signal value from different values assigned by different drivers

Conversion functions

- A function to convert *bit_vector* to *integer*

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

Functions have declarative and body parts

- *to_integer(x)* is an integer
  - Can be used anywhere an integer is expected

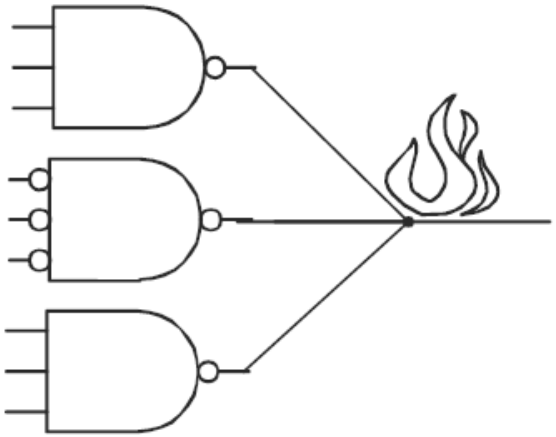
```
variable y: integer;
variable x: bit_vector(0 to 7);
y=to_integer(x)+5;
```

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

Functions have declarative and body parts

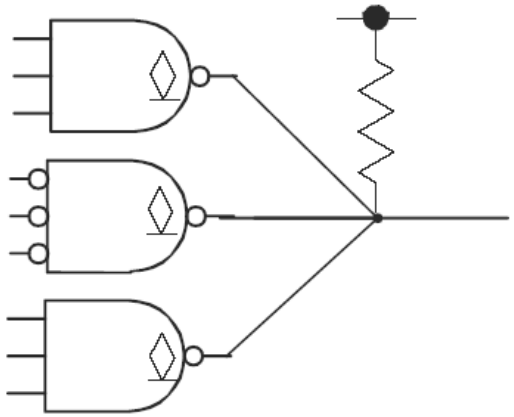
Resolution function

- Normally several sources cannot drive a signal
  - Real circuits smoke!
  - An error in VHDL



Resolution function

- Multiple drivers is possible only if a resolution exists
- Example in hardware is "open collector“
- Pull_up provides resolution

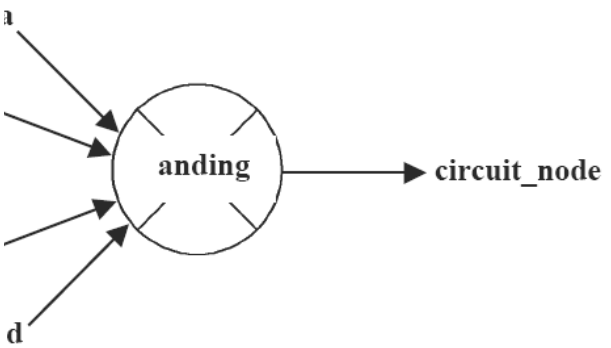


Resolution function

- To allow multiple drivers for a signal, we must resolve the signal
  - Get a function that takes different values and resolve the final signal value from them

→ Resolution function

```
FUNCTION anding ( drivers : Bit_Vector ) RETURN Bit IS
  VARIABLE accumulate : BIT := '1';
BEGIN
  FOR i IN drivers'RANGE LOOP
    accumulate := accumulate AND drivers(i);
  END LOOP;
  RETURN accumulate;
END anding;
```



The *anding* resolution function, ANDs all its drivers

Resolution function

- Define the resolution function for a signal

```
USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: qit
ARCHITECTURE wired_and OF y_circuit IS
  FUNCTION anding (drivers : qit_vector) RETURN qit IS
    VARIABLE accumulate : qit := '1';
  BEGIN
    FOR i IN drivers'RANGE LOOP
      accumulate := accumulate AND drivers(i);
    END LOOP;
    RETURN accumulate;
  END anding;
  SIGNAL circuit_node : anding qit;
BEGIN
  circuit_node <= a;
  circuit_node <= b;
  circuit_node <= c;
  circuit_node <= d;
  z <= circuit_node;
END wired_and;
```

! Qit is a 4-value data that is introduced in basic_utils package in "VHDL, Analysis and modeling…." book. It takes 0, 1, X, and Z

```
USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: qit
ARCHITECTURE wired_and OF y_circuit IS
  FUNCTION anding (drivers : qit_vector) RETURN qit IS
    VARIABLE accumulate : qit := '1';
  BEGIN
    FOR i IN drivers'RANGE LOOP
      accumulate := accumulate AND drivers(i);
    END LOOP;
    RETURN accumulate;
  END anding;
  SIGNAL circuit_node : anding qit;
BEGIN
  circuit_node <= a;
  circuit_node <= b;
  circuit_node <= c;
  circuit_node <= d;
  z <= circuit_node;
END wired_and;
```

Resolution function

- Types and subtypes can also be resolved
- All signals from that subtype are resolved
- Example: *std_logic* is a resolved type
  - In *IEEE* library, *std_logic_1164* package

Std_logic type

- Similar to bit, but a 9-value type
- To model the signals more precisely
- First define std_ulogic: an unresolved type

```
TYPE std_ulogic IS ( 'U',   -- Uninitialized
                     'X',   -- Forcing  Unknown
                     '0',   -- Forcing  0
                     '1',   -- Forcing  1
                     'Z',   -- High Impedance
                     'W',   -- Weak     Unknown
                     'L',   -- Weak     0
                     'H',   -- Weak     1
                     '-'    -- Don't care
                   );
```

Std_logic type

- Then define std_logic: a subtype of std_ulogic which is resolved
- Using a resolution function with the name *resolved*
- Part of std_logic package declaration:

```
-------------------------------------------------------
SUBTYPE std_logic IS resolved std_ulogic;

-------------------------------------------------------
-- unconstrained array of std_logic for use in
-- declaring signal arrays
-------------------------------------------------------
TYPE std_logic_vector IS ARRAY ( NATURAL RANGE <>) OF
  std_logic;
```

std_logic resolution function

- To resolve the final value, when several drivers drive the signal at the same time
  - An array containing several values is passed to the function and a single bit is returned

```
    ------------------------------------------------------
    -- resolution function
    ------------------------------------------------------
    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
        -- the test for a single driver is essential
        -- otherwise the loop would return 'X' for a
        -- single driver of '-' and that would conflict
        -- with the value of a single driver unresolved
        -- signal.
        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 resolved;
```

Procedures

- Like functions, but accept multiple inputs and outputs and inouts
- A procedure call is considered a statement of its own
  - function usually exists as part of an expression

```
.....
y<=func(x);
Proc(x);
....
```

- The most usual case of using a procedure is when more than one value is returned
- Procedures have roughly the same syntax and rules as function

A procedure without parameters

```
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;
```

- Procedure is declared in the declarative part of the process
- It has no input, just works on the global signals of the architecture

Procedures

- Procedure with input parameter

Declaration:

```
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;
```

Call:

```
do_arith_op ( func );
        ⋮
do_arith_op ( add );
```

Procedure- IN and OUT parameters

Procedure declaration

```
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;
```

Procedure call

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

Procedure parameter list

- The parameter mode should be defined: IN, OUT, INOUT
- The rules of the entity ports are applied:
  - e.g.: the statements in the procedure can use parameter values in the IN mode but cannot modify it
- Mode is an optional part
  - If we leave it out, mode **IN** is assumed

Procedure parameter list

- In addition to mode, the class of parameters should be defined
- Parameters can have three classes: Variable, Signal, Constant
- If we don't specify the class, default classes will be used:
  - Constant for IN parameters
  - Variable for out parameters

```
procedure receive_packet ( signal rx_data : in bit;
                           signal rx_clock : in bit;
                           data_buffer : out packet_array ) is
```

Signal parameters

- A signal parameter can be of any of the modes in, out or inout
- The way that signal parameters work is somewhat different from constant and variable parameters
  - Constant and variable IN parameters: the parameter value is sampled at the call time and the value is passed to the procedure
  - Signal: Instead of passing the value of the signal, the signal object itself is passed to procedure
- Signal parameter: If the procedure executes a wait statement, the signal value may be different after the wait statement completes and the procedure resumes

Signal parameters

- The process *packet_assembler* calls the procedure *receive_packet*
- After each wait, the current value of *rx_data* is read
- When the procedure reaches the wait statement, it is really the calling process that suspends

```
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;
```