ECE381(CAD), Lecture 9:

Data Types and Operators 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"

Data Types and Operators in VHDL

- Readings:
  1. “VHDL: Analysis & Modeling of Digital Systems”: Chapter 7
  2. “VHDL by Example”: Chapters 4
  3. “Designers Guide To VHDL”: Chapter 2

Object classes in VHDL

- Recall: 4 object classes:
  - Signal
  - Variable
  - Constant
  - file

Scope of signals

- Global to entities
  - If defined in an entity is visible to all architectures of that entity
- Local to architectures
  - If defined in an architecture is visible inside that architecture

Scope of signals

```
ENTITY board_design is
  PORT( data_in : IN bus_type;
        data_out : OUT bus_type);

  SIGNAL sys_clk : std_logic := '1';

END board_design;

ARCHITECTURE data_flow OF board_design IS
  SIGNAL int_bus : bus_type;
  CONSTANT disconnect_value : bus_type
   := ('X', 'X', 'X', 'X', 'X', 'X', 'X', 'X');
BEGIN
  int_bus <= data_in WHEN sys_clk = '1'
   ELSE int_bus;

  data_out <= magic_function(int_bus) WHEN sys_clk = '0'
   ELSE disconnect_value;

  sys_clk <= NOT(sys_clk) after 50 ns;
END data_flow;
```

- *sys_clk* is visible to each architecture of *board_design*
- *int_bus* is only visible inside *data_flow*

Constants

```
CONSTANT PI: REAL := 3.1414;
```

- The same scoping as signals
- If a constant is defined in a package it is visible to all entities that use that package

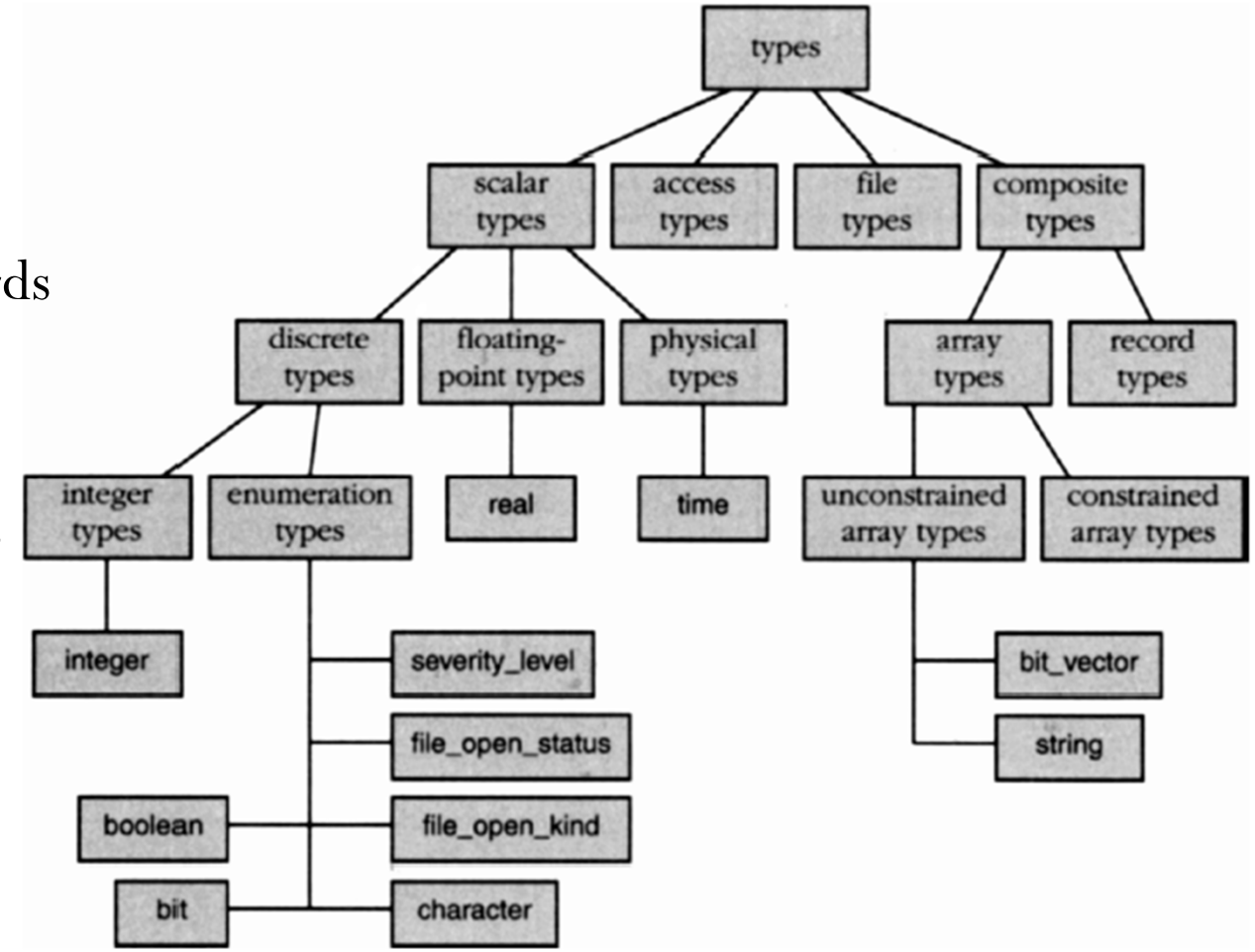
Data types

- All objects have a type
- VHDL contains a wide range of types that can be used to create simple or complex objects
- The VHDL Standard package defines all of the predefined VHDL types provided with the language
- We can define new types with this general format:

TYPE *identifier* IS *type_definition* ;

Diagram of data types

- Scalar types
  - Simple types
- Composite types
  - Arrays and records
- Access types
  - Equivalent of pointers in C++
- File types
  - External in/out



Scalar types

- Scalar: Objects that can hold, at most, one value at a time
  - Integer types
  - Real types
  - Enumerated types
  - Physical types
  - Chatacter

Scalar type- integer

- Exactly like the mathematical integers
- All of the normal predefined mathematical functions like add, subtract, multiply, and divide apply to integer types

```
ARCHITECTURE test OF test IS
BEGIN
PROCESS(X)
 VARIABLE a : INTEGER;
 VARIABLE b : int_type;
BEGIN
 a := 1;     --Ok     1
 a := -1;    --Ok     2
 a := 1.0;   --error 3
END PROCESS;
END test;
```

Scalar type- real

- Exactly like the mathematical floating-point numbers
- Can be used in both scientific and decimal representations

```
ARCHITECTURE test OF test IS
  SIGNAL a : REAL;
BEGIN
  a <= 1.0;          --ok   1
  a <= 1;            --error 2
  a <= -1.0E10;      --ok    3
  a <= 1.5E-20;      --ok    4
  a <= 5.3 ns;       --error 5
END test;
```

Scalar type- enumerated

- A user-defined type: the type values are defined by user
- Powerful tool for code readability

```
TYPE instruction IS ( add, sub, lda, ldb, sta, stb, outa,
     xfr );

                                --select instruction to
CASE instr is                   --execute
  WHEN lda =>
    a := data;                  --load a accumulator

  WHEN ldb =>
    b := data;                  --load b accumulator

  WHEN add =>
    a := a 1 b;                 --add accumulators

  WHEN sub =>
    a := a -b;                  --subtract accumulators

  WHEN sta =>
    reg(addr) := a;             --put a accum in reg array

  WHEN stb =>
    reg(addr) := b;             --put b accum in reg array

  WHEN outa =>
    data <= a;                  --output a accum

  WHEN xfr =>                   --transfer b to a
    a := b;

END CASE;
```

Scalar type- physical

- Represent physical quantities such as distance, current, time,…
- Has a base unit
  - Successive units are then defined in terms of this unit
- The smallest unit representable is one base unit
- Range: minimum and maximum values in base units

```
TYPE current IS RANGE 0 to 1000000000

UNITS
   na;                  --nano amps
   ua = 1000 na;        --micro amps
   ma = 1000 ua;        --milli amps
   a  = 1000 ma;        --amps
END UNITS;
```

Time data type

- A predefined physical type
- We have used it before when determining delays

```
TYPE TIME IS RANGE <implementation defined>
UNITS
 fs;                    --femtosecond
 ps     =   1000 fs;    --picosecond
 ns     =   1000 ps;    --nanosecond
 us     =   1000 ns;    --microsecond
 ms     =   1000 us;    --millisecond
 sec    =   1000 ms;    --second
 min    =   60 sec;     --minute
 hr     =   60 min;     --hour
END UNITS;
```

Character

- A predefined enumeration type
- See page 41 of "The designers guide to VHDL" for the declaration

```
VARIABLE cmd_char: character;
…
cmd_char := 'P';
```

Boolean

- One of the most important predefined enumeration types in VHDL

```
TYPE boolean IS (false, true);
```

- Used to represent condition values, which can control execution of a conditional statement
- Relational and logical operators work on boolean
  - To be introduced soon!

Bit

- A predefined enumeration to model a bit in digital systems

```
TYPE bit IS ('0', '1 ');
```

- The VHDL logical operators can be applied to values of type bit, and they produce results of type bit

Composite types

- Arrays
  - Groups of elements of the same type
- Records
  - Groups of elements of different types.

Composite types- arrays

- Much like the arrays in C++
- Elements are accessed by their index and "()"

```
TYPE data_bus IS ARRAY(0 TO 31) OF BIT;

VARIABLE X: data_bus;
VARIABLE Y: BIT;

Y := X(0);    --line 1
Y := X(15);   --line 2
```

Memory by arrays

- Array of arrays

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
PACKAGE memory IS
  CONSTANT width   : INTEGER := 3;
  CONSTANT memsize : INTEGER := 7;

  TYPE data_out IS ARRAY(0 TO width) OF std_logic;
  TYPE mem_data IS ARRAY(0 TO memsize) OF data_out;
END memory;
```

```
CONSTANT rom_data : mem_data :=
 ( ( '0', '0', '0', '0'),
   ( '0', '0', '0', '1'),
   ( '0', '0', '1', '0'),
   ( '0', '0', '1', '1'),
   ( '0', '1', '0', '0'),
   ( '0', '1', '0', '1'),
   ( '0', '1', '1', '0'),
   ( '0', '1', '1', '1') );
```

Arrays

- Addressing a single element in an array of array

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
PACKAGE memory IS
  CONSTANT width   : INTEGER := 3;
  CONSTANT memsize : INTEGER := 7;

  TYPE data_out IS ARRAY(0 TO width) OF std_logic;
  TYPE mem_data IS ARRAY(0 TO memsize) OF data_out;
END memory;
```

```
bit_value := rom_data(addr) (bit_index);
```

Multi-dimensional arrays

- Initializing remains the same as array of array
- Addressing is different

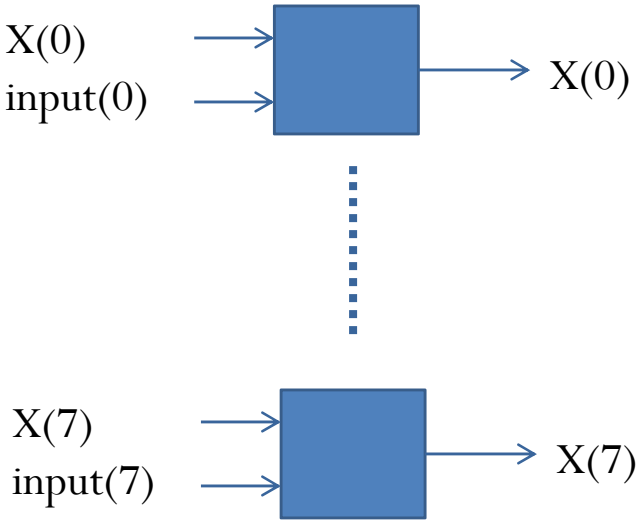
```
TYPE mem_data_md IS ARRAY(0 TO memsize, 0 TO width) OF
      std_logic;

CONSTANT rom_data_md : mem_data :=
( ( '0', '0', '0', '0'),
  ( '0', '0', '0', '1'),
  ( '0', '0', '1', '0'),
  ( '0', '0', '1', '1'),
  ( '0', '1', '0', '0'),
  ( '0', '1', '0', '1'),
  ( '0', '1', '1', '0'),
  ( '0', '1', '1', '1') );
```

```
X := rom_data_md(3, 3);
```

Arrays and LOOP

- Arrays are often processed by loops
- In synthesis, loops are unrolled and synthesized by replicating the corresponding hardware

```
PROCESS ()
BEGIN
FOR i IN 0 TO 7 LOOP
   X(i)<=X(i)+input(i);
END LOOP
END PROCESS;
```



Arrays and GENERATE

- We can use the for-generate statement that was originally used for component instantiation in structural-level coding for concurrent array processing

```
ARCHITECUTR x OF y IS
BEGIN
....
g: FOR I IN 0 TO 7 GENERATE
    X(i)<=X(i)+input(i);
END GENERATE g;
.....
END ARCHITECTURE;
```

Composite types- records

- Group objects of many types together as a single object
- Like *struct* in C
- Each element of the record can be accessed by its field name

```
TYPE optype IS ( add, sub, mpy, div, jmp );

TYPE instruction IS
RECORD
 opcode  : optype;
 src     : INTEGER;
 dst     : INTEGER;
END RECORD;
```

```
PROCESS(X)
  VARIABLE inst : instruction;
  VARIABLE source, dest : INTEGER;
  VARIABLE operator : optype;
BEGIN
  source := inst.src;              --Ok line 1
  dest   := inst.src;              --Ok line 2

  source := inst.opcode;           --error line 3
  operator := inst.opcode;         --Ok line 4

  inst.src  := dest;               --Ok line 5
  inst.dst := dest;                --Ok line 6

  inst := (add, dest, 2);          --Ok line 7
  inst := (source);                --error line 8
END PROCESS;
```

Composite types- records

- Records may have composite fields

```
TYPE word IS ARRAY(0 TO 3) OF std_logic;
TYPE t_word_array IS ARRAY(0 TO 15) OF word;
TYPE addr_type IS
RECORD
 source : INTEGER;
 key    : INTEGER;
END RECORD;

TYPE data_packet IS
RECORD
 addr : addr_type;
 data : t_word_array;
 checksum : INTEGER;
 parity : BOOLEAN;
END RECORD;
```

```
packet.addr.key := 5;
packet.addr := (10, 20);
```

Files and external I/O

- Using external I/O
- Very useful for initialization and test vector generation
- How to use file I/O
  - First, the file has to be declared as what goes into a file
  - Then a logical file name must be declared
  - Then, the file should be opened
  - Finally the file should be written to and read from

Files and external I/O

- The first two steps:
  - File type declaration

```
TYPE logic_data IS FILE OF CHARACTER;
```

  - Logical file name declaration
    - Can have more fields

```
FILE input_logic_value_file1: logic_data;
--Just declare a logical file


FILE input_logic_value_file2: logic_data IS “input.dat”;
--Declare a logical file and open in READ_MODE


FILE input_logic_value_file3: logic_data OPEN READ_MODE
        IS “input.dat”;
--Declare a logical file and open with the specified mode
```

Files and external I/O

- File declaration:

```
FILE output_logic_value_file1: logic_data;
--Just declare a logical file, open later
```

- File opening:

```
FILE_OPEN (input_logic_value_file, "input.dat", READ_MODE);

FILE_OPEN (output_logic_value_file, "output.dat", WRITE_MODE);
```

- Can open a file in READ_MODE, WRITE_MODE or APPEND_MODE
- Closing a file:

```
FILE_CLOSE (input_logic_value_file);
FILE_CLOSE (output_logic_value_file);
```

Files and external I/O

- Files can be used in sequential codes: (for example inside *process)*
- Three functions: *READ,WRITE, ENDFILE*

```
  SIGNAL s : OUT BIT; file_name : IN STRING; period : IN TIME)  IS
  VARIABLE char : CHARACTER;
  VARIABLE current : TIME := 0 NS;
  FILE input_value_file : logic_data;
BEGIN
  FILE_OPEN (input_value_file, file_name, READ_MODE);
  WHILE NOT ENDFILE (input_value_file) LOOP
    READ (input_value_file, char);
    IF char = '0' OR char = '1' THEN
      current := current + period;
      IF char = '0' THEN
        s <= TRANSPORT '0' AFTER current;
      ELSIF char = '1' THEN
        s <= TRANSPORT '1' AFTER current;
      END IF;
    END IF;
  END LOOP;
```

New type

This code is written inside a FUNCTION. Will be introduced soon!

Subtypes

- Define subtype of a type
  - To add constraints for selected signal assignment statements or case statements
  - To create a resolved subtype (will be discussed later)

```
TYPE INTEGER IS -2,147,483,647 TO +2,147,483,647;
SUBTYPE NATURAL IS INTEGER RANGE 0 TO +2,147,483,647;
```

- Subtypes and base types allow assignment between the two types

Std_logic type

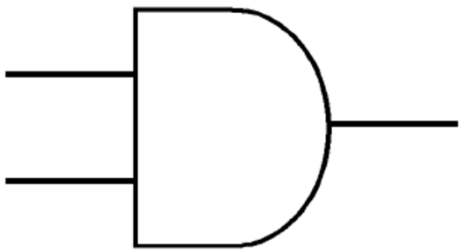
- Similar to bit, but a 9-value type
- An enumeration type in the IEEE library

Value	Representing
'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

Operators for *Std_logic*

- All logic operators can be applied on std_logic
- Example: AND gate

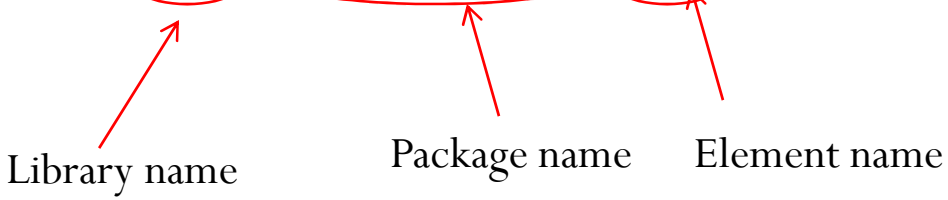
.	U	X	0	1	Z	W	L	H	-
U	'U'	'U'	'0'	'U'	'U'	'U'	'0'	'U'	'U'
X	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
0	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
1	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
Z	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
W	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'
L	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'	'0'
H	'U'	'X'	'0'	'1'	'X'	'X'	'0'	'1'	'X'
-	'U'	'X'	'0'	'X'	'X'	'X'	'0'	'X'	'X'



Std_logic type

- You should add the IEEE library to use std_logic

```
LIBRARY IEEE
USE IEEE.std_logic_1164.ALL
```



Operators

VHDL operators

- Logical
- Relational
- Shift
- Arithmetic
- See page 54 of "The designers guide to VHDL" for a complete list of the VHDL operators

VHDL operators- logical

Logical Operators:

AND, OR, NAND, NOR, XOR, XNOR, NOT

Examples of use:

```
x <= a XNOR b;

x_vector <= a_vector AND b_vector;


x <= "XOR" (a, b);

x_vector <= "AND" (a_vector, b_vector);
```

VHDL operators- relational

Return true or false

Relational operators:

=, /=, <, <=, >, >=

Examples of use:

```
a_boolean <= i1 > i2;

b_boolean <= i1 /= i2;


--if a_bit_vector is "00011" and b_bit_vector is "00100"

a_bit_vector < b_bit_vector          returns TRUE
```

VHDL operators- shift

	Shift/Rotate	*Left/Right*	*Logical/Arithmetic*
SLL	Shift	Left	Logical
SLA	Shift	Left	Arithmetic
SRL	Shift	Right	Logical
SRA	Shift	Right	Arithmetic
ROL	Rotate	Left	Logical
ROR	Rotate	Right	Logical

Start with *aq*	Z	0	1	X	Z	1	0	X
aq SLL 1	0	1	X	Z	1	0	X	**0**
aq SLA 1	0	1	X	Z	1	0	X	**X**
aq SRL 1	**0**	Z	0	1	X	Z	1	0
aq SRA 1	**Z**	Z	0	1	X	Z	1	0
aq ROL 1	0	1	X	Z	1	0	X	Z
aq ROR 1	X	Z	0	1	X	Z	1	0

VHDL operators- arithmetic

- +, -, *, /, *MOD*, *REM*
- Addition and subtraction are defined on *INTEGER*, *REAL*, *BIT_VECTOR*, and *STD_LOGIC_VECTOR*
- *MOD* and *REM* are defined on *INTEGER*
- * and / are defined on *INTEGER* and *REAL*

```
a + b

"+" (a, b)

a_int MOD b_int  -- both integers

a_int REM b_int  -- returns remainder of absolute value division
```

Unconstrained Arrays

Attributes

- Gives information about the types, signals, arrays, entities,…
- Array attributes: to find range, length, and boundaries of arrays
- Signal attributes: to find the activity of signals
- Type attributes: to find information about the values a type can get

Array attributes

- ‘length: returns the array length
- ‘range: returns the array range

```
PROCESS(a)
 TYPE bit4 IS ARRAY(0 TO 3) of BIT;
 TYPE bit_strange IS ARRAY(10 TO 20) OF BIT;
 VARIABLE len1, len2 : INTEGER;
BEGIN
 len1 := bit4'LENGTH;          -- returns 4
 len2 := bit_strange'LENGTH; -- returns 11
END PROCESS;
```

bit4’range = 0 to 3

We can use it in a loop

FOR I IN bit4’range LOOP

……

Array attributes

- ‘left, ‘right, ‘high, ‘low: returns the array boundaries

```
PROCESS(a)
 TYPE bit_range IS ARRAY(31 DOWNTO 0) OF BIT;
 VARIABLE left_range, right_range, uprange, lowrange :
      integer;
BEGIN
 left_range  := bit_range'LEFT;
 -- returns 31

 right_range := bit_range'RIGHT;
 -- returns 0

 uprange     := bit_range'HIGH;
 -- returns 31

 lowrange    := bit_range'LOW;
 -- returns 0
END PROCESS;
```

Unconstrained arrays

- We just indicate the type of the index values, without specifying bounds

  TYPE sample IS ARRAY (NATURAL RANGE <>) OF integer;

- The symbol "<>", often called "box"
- All unconstrained arrays will be fixed according to the parameters passed to them, when they are used

  VARIABLE short_sample_buf : sample(0 to 63);

Unconstrained arrays

- BIT_VECTOR, STRING, STD_LOGIC_VECTOR and INTEGER_VECTOR are predefined unconstrained array

```
TYPE BIT_VECTOR IS
      ARRAY (NATURAL RANGE <>) OF BIT;

TYPE STRING IS
      ARRAY (POSITIVE RANGE <>) OF CHARACTER;


TYPE integer_vector IS
      ARRAY (NATURAL RANGE <>) OF INTEGER;
```

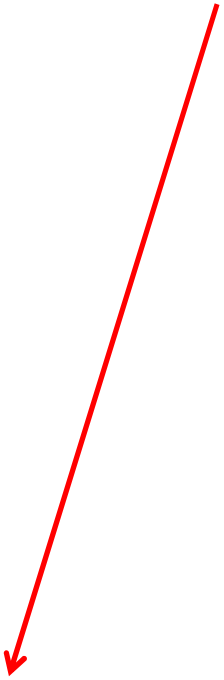
Unconstrained array ports

- We can use unconstrained arrays to design very general entity codes

```
entity and_multiple is
    port ( i : in bit_vector;  y : out bit );
end entity and_multiple;
-------------------------------
architecture behavioral of and_multiple is
begin
    and_reducer : process ( i ) is
        variable result : bit;
    begin
        result := '1';
        for index in i'range loop
            result := result and i(index);
        end loop;
        y <= result;
    end process and_reducer;
end architecture behavioral;
```

Unconstrained array ports

- Actual entity size is determined when the entity is instantiated



```
ENTITY High_level IS
……
END high_level;

ARCHITECTURE example OF high_level IS
        ……
        SIGNAL count_value: BIT_VECTOR(7 DOWNTO 0);
        SIGNAL terminal_count: bit;
        …….
BEGIN
        ….
        n1: ENTITY WORK.and_multiple(behavioral) PORT MAP (count_value, terminal_count);
        …..
END example:;
```

Unconstrained array ports

- Another example
- Using the unconstrained arrays to design a general n-bit comparator

Design parameterization by unconstraint arrays

- We have seen this example before: 4-bit comparator
- Use a constant for parameterization
- Need to change the constant by designer to change comparator bit-width
  - needs recompiling the code

```
ENTITY nibble_comparator IS
PORT (a, b : IN BIT_VECTOR (3 DOWNTO 0);-- a and b data inputs
    gt,                                  -- previous greater than
    eq,                                  -- previous equal
    lt : IN BIT;                         -- previous less than
    a_gt_b,                              -- a > b
    a_eq_b,                              -- a = b
    a_lt_b : OUT BIT);                   -- a < b
END nibble_comparator;

ARCHITECTURE iterative OF nibble_comparator IS
  COMPONENT comp1
    PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
  END COMPONENT;
  FOR ALL : comp1 USE ENTITY WORK.bit_comparator (gate_level);
  CONSTANT n : INTEGER := 4;
  SIGNAL im : BIT_VECTOR ( 0 TO (n-1)*3-1);
BEGIN
  c_all: FOR i IN 0 TO n-1 GENERATE
    l: IF i = 0 GENERATE
      least: comp1 PORT MAP (a(i), b(i), gt, eq, lt, im(0), im(1), im(2) );
    END GENERATE;
    m: IF i = n-1 GENERATE
      most: comp1 PORT MAP (a(i), b(i),
          im(i*3-3), im(i*3-2), im(i*3-1), a_gt_b, a_eq_b, a_lt_b);
    END GENERATE;
    r: IF i > 0 AND i < n-1 GENERATE
    rest: comp1 PORT MAP (a(i), b(i), im(i*3-3), im(i*3-2), im(i*3-1),
          im(i*3+0), im(i*3+1), im(i*3+2) );
    END GENERATE;
  END GENERATE;
END iterative;
```

Design parameterization by unconstraint arrays

- Using unconstraint arrays: not specifying bit width of inputs
- Set when the entity is instantiated

```
ENTITY n_bit_comparator IS
  PORT (a, b : IN BIT_VECTOR;  gt, eq, lt : IN BIT;
        a_gt_b,  a_eq_b,  a_lt_b : OUT BIT);
END n_bit_comparator;
--
ARCHITECTURE structural OF n_bit_comparator IS
  COMPONENT comp1
    PORT (a, b, gt, eq, lt : IN BIT; a_gt_b, a_eq_b, a_lt_b : OUT BIT);
  END COMPONENT;
  FOR ALL : comp1 USE ENTITY WORK.bit_comparator (functional);
  CONSTANT n : INTEGER := a'LENGTH;
  SIGNAL im : BIT_VECTOR ( 0 TO (n-1)*3-1);
BEGIN
  c_all: FOR i IN 0 TO n-1 GENERATE
    l: IF i = 0 GENERATE
      least: comp1 PORT MAP (a(i), b(i), gt, eq, lt, im(0), im(1), im(2) );
    END GENERATE;
    m: IF i = n-1 GENERATE
      most: comp1 PORT MAP
          (a(i), b(i), im(i*3-3), im(i*3-2), im(i*3-1), a_gt_b, a_eq_b, a_lt_b);
    END GENERATE;
    r: IF i > 0 AND i < n-1 GENERATE
      rest: comp1 PORT MAP
       (a(i), b(i), im(i*3-3), im(i*3-2), im(i*3-1), im(i*3+0), im(i*3+1), im(i*3+2) );
    END GENERATE;
  END GENERATE;
END structural;
```

Design parameterization by unconstraint arrays

- When the entity is instantiated, its bit width is set based on the passed parameter (6 in this example)

```
ENTITY n_bit_comparator_test_bench IS
END n_bit_comparator_test_bench ;
--
USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: apply_data which uses integer_vector
ARCHITECTURE procedural OF n_bit_comparator_test_bench IS
  COMPONENT comp_n PORT (a, b : IN bit_vector;
                         gt, eq, lt : IN BIT;
                         a_gt_b, a_eq_b, a_lt_b : OUT BIT);
  END COMPONENT;
  FOR a1 : comp_n USE ENTITY
WORK.n_bit_comparator(structural);
  SIGNAL a, b : BIT_VECTOR (5 DOWNTO 0);
  SIGNAL eql, lss, gtr : BIT;
  SIGNAL vdd : BIT := '1';
  SIGNAL gnd : BIT := '0';
BEGIN
  a1: comp_n PORT MAP (a, b, gnd, vdd, gnd, gtr, eql, lss);
  apply_data (a, 00&15&57&17, 500 NS);
  apply_data (b, 00&43&14&45&11&21&44&11, 500 NS);
END procedural;
```