PART 6

Utilities_for_high_level_descriptions

- Language aspects
- Types
    - declarations
    - enumeration
    - scalars
    - physical
    - arrays
    - files, simple IO
    - composites
    - subtypes
    - aliasing
- Overloading subprogram operators
- Predefined Attributes for
    - arrays
    - types
    - signals
- User defined attributes

In General:

- Language aspects will be discussed
- Examples will clarify language aspects
- Hardware related examples are used
- Attributes are important in modeling

Utilities.types.scalars

```
TYPE qit IS ('0', '1', 'Z', 'X');
```

* * *

- New type for multi-value logic system
- *qit* is type identifier
- '0', '1', 'Z', 'X' are enumeration elements
- This is enumeration type definition
- '0' is default initial values
- Let's assume *qit* goes in *basic_utilities*

Utilities.types.scalars

In:	
0	1
1	0
Z	0
X	X
	out

	In1: 0	1	Z	X
In2: 0	1	1	1	1
1	1	0	0	X
Z	1	0	0	X
X	1	X	X	X
		out		

* * *

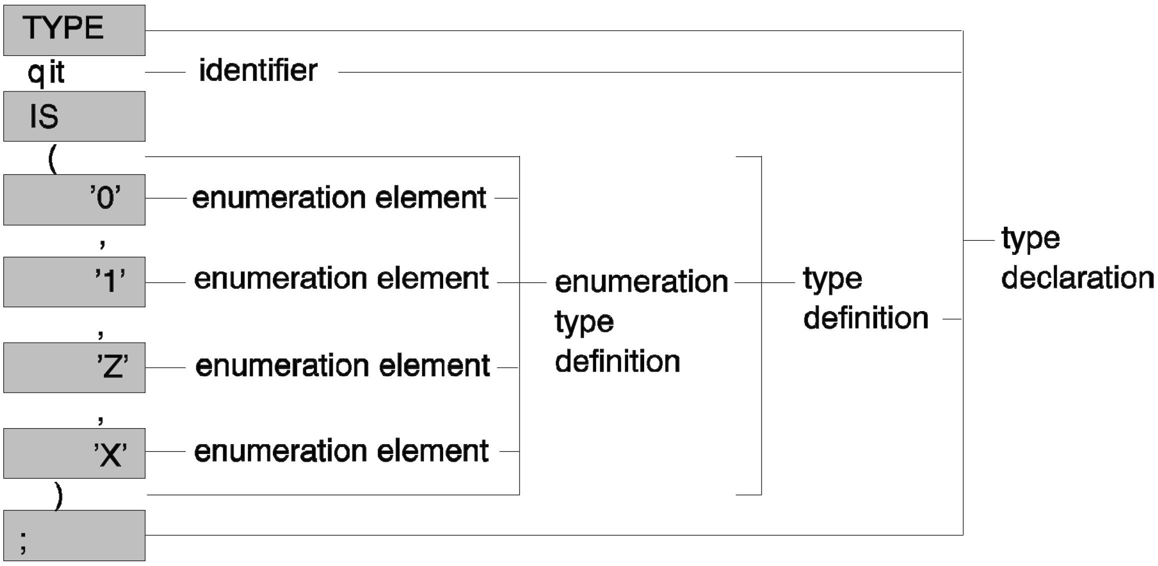
- Can develop logic gates using *qit* values system
- NOT and NAND tables in *qit*

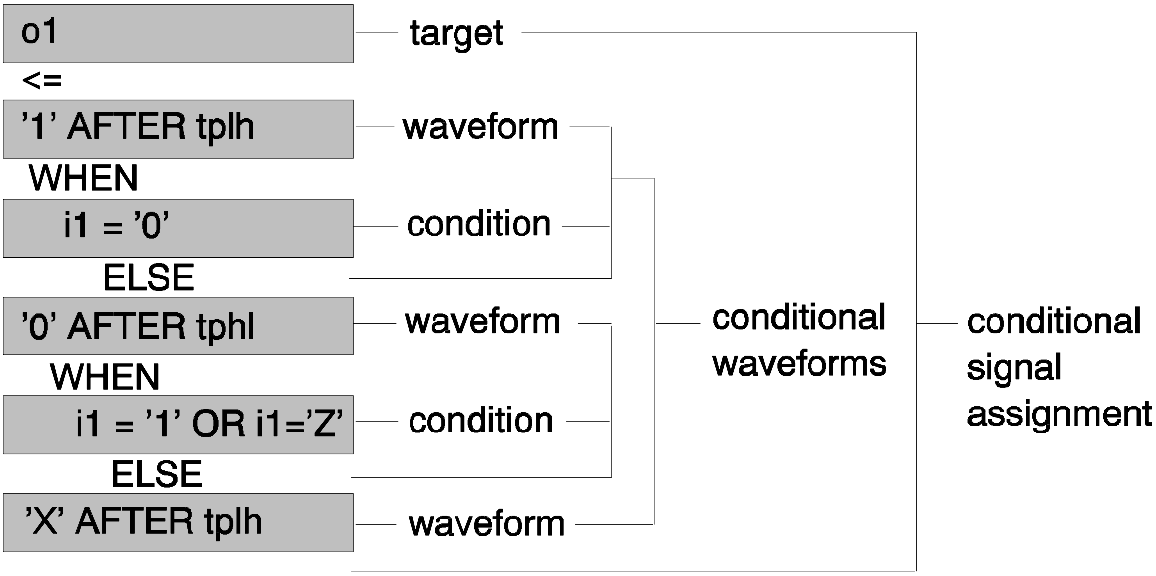
Utilities.types.scalars

```
USE WORK.basic_utilities.ALL;
ENTITY inv_q IS
  GENERIC (tplh : TIME := 5 NS; tphl : TIME := 3 NS);
  PORT (i1 : IN qit; o1 : OUT qit);
END inv_q;
--
ARCHITECTURE double_delay OF inv_q IS
BEGIN
  o1 <= '1' AFTER tplh WHEN i1 = '0' ELSE
       '0' AFTER tphl WHEN i1 = '1' OR i1 = 'Z' ELSE
       'X' AFTER tplh;
END double_delay;

USE WORK.basic_utilities.ALL;
ENTITY nand2_q IS
  GENERIC (tplh : TIME := 7 NS; tphl : TIME := 5 NS);
  PORT (i1, i2 : IN qit; o1 : OUT qit);
END nand2_q;
--
ARCHITECTURE double_delay OF nand2_q IS
BEGIN
  o1 <= '1' AFTER tplh WHEN i1 = '0' OR i2 = '0' ELSE
       '0' AFTER tphl WHEN (i1 = '1' AND i2 = '1') OR
                       (i1 = '1' AND i2 = 'Z') OR
                       (i1 = 'Z' AND i2 = '1') OR
                       (i1 = 'Z' AND i2 = 'Z') ELSE
       'X' AFTER tplh;
END double_delay;
```

- INV and NAND2 gates using *qit*
- Use conditional signal assignments
- *Double_delay* values are used

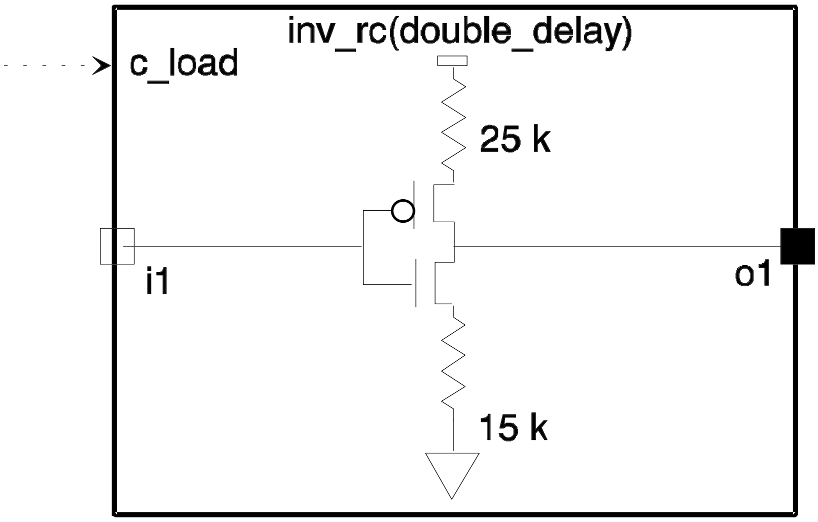
Utilities.types.scalars





- Syntax for the new constructs

Utilities.types.scalars



```
USE WORK.basic_utilities.ALL;   --  FROM PACKAGE USE : qit
ENTITY inv_rc IS
  GENERIC (c_load : REAL := 0.066E-12);  -- Farads
  PORT (i1 : IN qit; o1 : OUT qit);
  CONSTANT rpu : REAL := 25000.0;  -- Ohms
  CONSTANT rpd : REAL := 15000.0;  -- Ohms
END inv_rc;
--
ARCHITECTURE double_delay OF inv_rc IS
  CONSTANT tplh : TIME := INTEGER ( rpu * c_load * 1.0E15) * 3 FS;
  CONSTANT tphl : TIME := INTEGER ( rpd * c_load * 1.0E15) * 3 FS;
BEGIN
  o1 <= '1' AFTER tplh WHEN i1 = '0' ELSE
  '0' AFTER tphl WHEN i1 = '1' OR i1 = 'Z' ELSE     'X' AFTER tplh;
END double_delay;
```

- Other types can be used for signals, constants and variables
- Example shows use of REAL
- Time is calculated from R and C values

Utilities.types.scalars

```
CONSTANT tplh : TIME := INTEGER (rpu* c_load* 1.0E15)*3 FS;
```

* * *

- REAL *rpu* multiplied by REAL *c_load*
- 1.0E15 provides FS scaling
- INTEGER(...) converts REAL to INTEGER
- INTEGER and REAL are closely-related types
- 3*RC is for the exponential delay

Utilities.types.scalars

```
TYPE capacitance IS RANGE 0 TO 1E16
 UNITS
   ffr;  -- Femto Farads (base unit)
   pfr = 1000 ffr;
   nfr = 1000 pfr;
   ufr = 1000 nfr;
   mfr = 1000 ufr;
   far = 1000 mfr;
   kfr = 1000 far;
 END UNITS;


TYPE resistance IS RANGE 0 TO 1E16
 UNITS
   l_o;  -- Milli-Ohms (base unit)
   ohms = 1000 l_o;
   k_o = 1000 ohms;
   m_o = 1000 k_o;
   g_o = 1000 m_o;
 END UNITS;
```

- Use physical types for better correspondence
- Capacitance for C values
- Resistance for R values
- Put both type definitions in *basic_utilities*

Utilities.types.scalars

```
USE WORK.basic_utilities.ALL;
--  FROM PACKAGE USE : qit, resistance, capacitance
ENTITY inv_rc IS
  GENERIC (c_load : capacitance := 66 ffr);
  PORT (i1 : IN qit; o1 : OUT qit);
  CONSTANT rpu : resistance := 25000 ohms;
  CONSTANT rpd : resistance := 15000 ohms;
END inv_rc;
--
ARCHITECTURE double_delay OF inv_rc IS
  CONSTANT tplh : TIME := (rpu / 1 l_o) * (c_load / 1 ffr) * 3 FS / 1000;
  CONSTANT tphl : TIME := (rpd / 1 l_o) * (c_load / 1 ffr) * 3 FS / 1000;
BEGIN
  o1 <= '1' AFTER tplh WHEN i1 = '0' ELSE
       '0' AFTER tphl WHEN i1 = '1' OR i1 = 'Z' ELSE
       'X' AFTER tplh;
END double_delay;
```

- Use physical R and C types
- Equation converts values to their base units
- l_o * ffr requires 10^{-18} adjustment

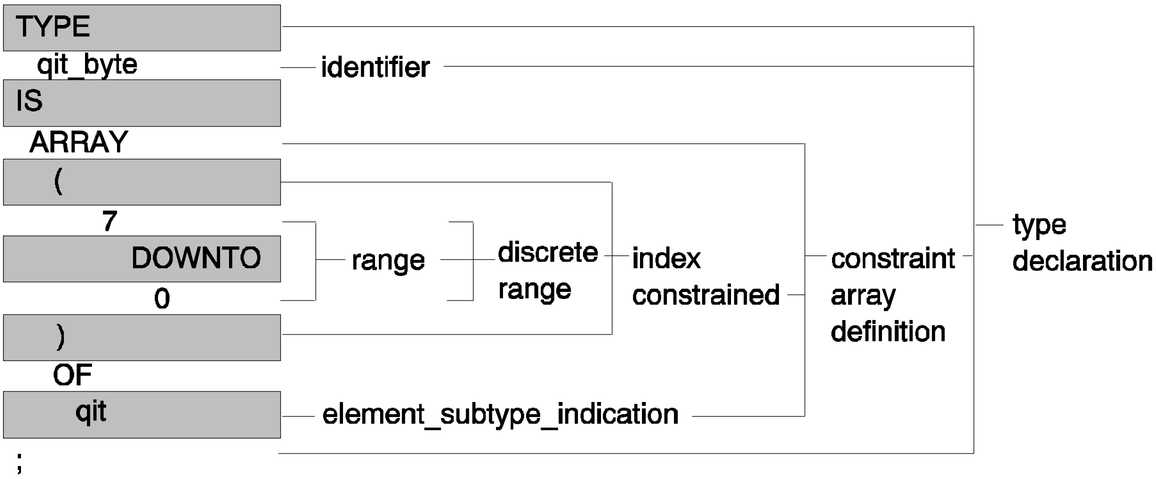
Utilities.type.arrays

```
TYPE qit_nibble IS ARRAY ( 3 DOWNTO 0 ) OF qit;

TYPE qit_byte IS ARRAY ( 7 DOWNTO 0 ) OF qit;

TYPE qit_word IS ARRAY ( 15 DOWNTO 0 ) OF qit;

TYPE qit_4by8 IS ARRAY ( 3 DOWNTO 0, 0 TO 7 ) OF qit;

TYPE qit_nibble_by_8 IS ARRAY ( 0 TO 7 ) OF qit_nibble;
```

```
SIGNAL sq8 : qit_byte := "ZZZZZZZZ";

sq8 <= "Z101000Z";
```

- Various array types can be based on *qit*
- Type *qit_4by8* is two dimensional
- Type *qit_nibble_by_8* is one dimensional of *qit_nibble*
- *qit_4by8* and *qit_nibble_by_8* specify different indexings
- String of enumeration elements forms an array
- The *sq8* signal is initialized to "ZZZZZZZZ"
- Signal value can consist of a string of enumaration elements

Utilities.type.arrays



TYPE
qit_byte
IS
ARRAY
(
7
DOWNTO
0
)
OF
qit
;

identifier
range
discrete range
index constrained
constraint array definition
element_subtype_indication
type declaration

* * *

- Syntax of array type declaration

Utilities.type.arrays

```
TYPE qit_nibble IS ARRAY ( 3 DOWNTO 0 ) OF qit;
TYPE qit_byte IS ARRAY ( 7 DOWNTO 0 ) OF qit;
TYPE qit_word IS ARRAY ( 15 DOWNTO 0 ) OF qit;
TYPE qit_4by8 IS ARRAY ( 3 DOWNTO 0, 0 TO 7 ) OF qit;
TYPE qit_nibble_by_8 IS ARRAY ( 0 TO 7 ) OF qit_nibble;
```

```
SIGNAL sq1  :  qit;
SIGNAL sq4  :  qit_nibble;
SIGNAL sq8  :  qit_byte;
SIGNAL sq16  :  qit_word;
SIGNAL sq_4_8  :  qit_4by8;
SIGNAL sq_nibble_8  :  qit_nibble_by_8;
```

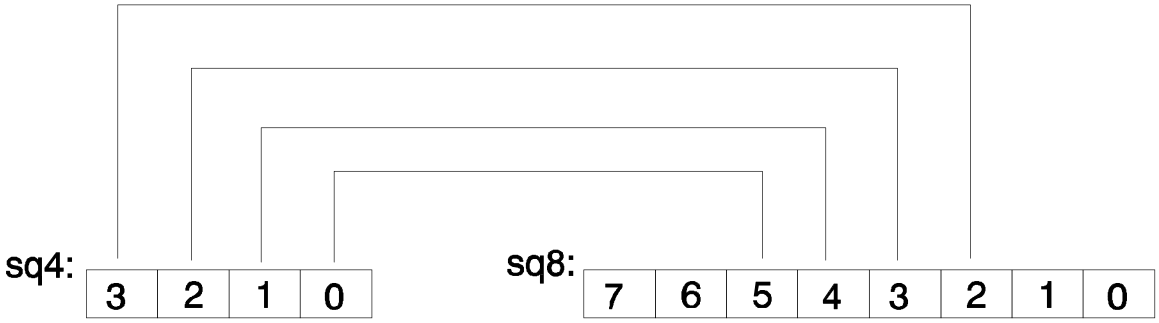
```
sq8 <= sq16 (11 DOWNTO 4);              -- middle 8 bit slice of sq16 to sq8
sq16 (15 DOWNTO 12) <= sq4;             --  sq4 into left 4 bit slice of sq16
sq1 <= sq_4_8 (0, 7);                   -- lower right bit of sq_4_8 to sq1
sq4 <= sq_nibble_8 (2);                 -- third nibble of  sq_nibble_8 into
    sq4
sq8 <= sq8 (0) & sq8 (7 DOWNTO 1);             -- right rotate sq8
sq4 <= sq8 (2) & sq8 (3) & sq8 (4) & sq8 (5);    -- reversing sq8 into sq4
```

- Signal declarations based on defined types
- Examples of signal assignments
- Concatenation operator can be used for shift and rotate
- Direction of indexing must be as declared

Utilities.type.arrays

```
TYPE qit_nibble IS ARRAY ( 3 DOWNTO 0 ) OF qit;
TYPE qit_byte IS ARRAY ( 7 DOWNTO 0 ) OF qit;
```

```
SIGNAL sq4  :  qit_nibble;
SIGNAL sq8  :  qit_byte;
```



```
sq4 <= sq8(2) & sq8(3) & sq8(4) & sq8(5);
```

* * *

- Use concatenation if reversing is required

Utilities.types.arrays

```
TYPE qit_4by8 IS ARRAY ( 3 DOWNTO 0, 0 TO 7 ) OF qit;
```

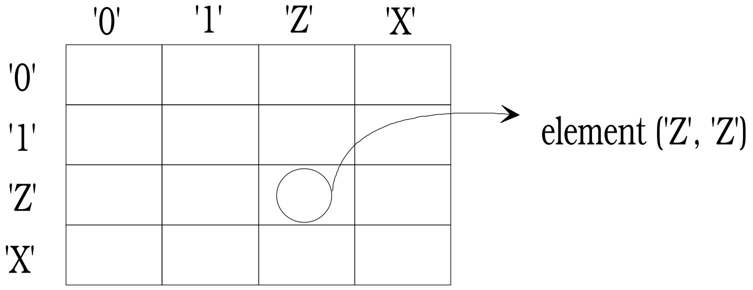
```
SIGNAL sq_4_8 : qit_4by8 :=
                (
        ( '0', '0', '1', '1', 'Z', 'Z', 'X', 'X' ),              -- sq_4_8 (3, 0 TO 7)
        ( 'X', 'X', '0', '0', '1', '1', 'Z', 'Z' ),              -- sq_4_8 (2, 0 TO 7)
        ( 'Z', 'Z', 'X', 'X', '0', '0', '1', '1' ),              -- sq_4_8 (1, 0 TO 7)
        ( '1', '1', 'Z', 'Z', 'X', 'X', '0', '0' )               -- sq_4_8 (0, 0 TO 7)
                );
```

* * *

- Use nested parenthesis for multidimensional arrays
- Deepest set of parenthesis corresponds to right most index
- Example shows initialization of *sq_4_8*
- Type of signal is two dimensional *qit_4by8*

Utilities.types.indexing

```
TYPE qit_2d IS ARRAY (qit, qit) OF qit;
```



* * *

- Can use enumaration types for indexing
- If *qc* is a signal of type *qit_char*,
  TYPE qit_char IS ARRAY (CHARACTER) OF qit;
- Then, *qc('A')* indexes a *qit* value in the *qc* array

- Use *qit* for indexing
- Upper left most element is indexed by ('0','0')
- Put *qit_2d* in *basic_utilities*

Utilities.types.indexing

```
USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: qit, qit_2d
ENTITY nand2_q IS
  GENERIC (tplh : TIME := 7 NS; tphl : TIME := 5 NS);
  PORT (i1, i2 : IN qit; o1 : OUT qit);
END nand2_q;
--
ARCHITECTURE average_delay OF nand2_q IS
  CONSTANT qit_nand2_table : qit_2d := (

                 --      '0'     '1'     'Z'     'X'
                 --      ------------------------------------------
                         ('1',   '1',    '1',    '1'),          -- '0'
                         ('1',   '0',    '0',    'X'),          -- '1'
                         ('1',   '0',    '0',    'X'),          -- 'Z'
                         ('1',   'X',    'X',    'X'));         --'X'
BEGIN
  o1 <= qit_nand2_table (i1, i2) AFTER (tplh + tphl) / 2;
END average_delay;
```

* * *

- Example shows NAND2 using *qit_2d*
- Constant values are set according to NAND table

Utilities.arrays.unconstrained

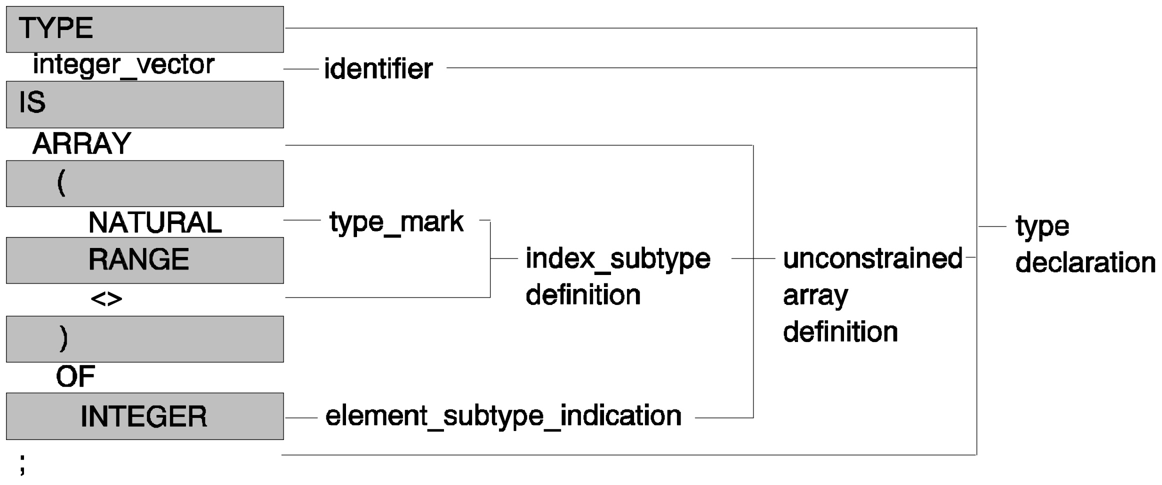
```
TYPE BIT_VECTOR IS ARRAY (NATURAL RANGE <>) OF BIT;
```

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

* * *

- NATURAL is defined in STD 0 TO 2147483047
- BIT_VECTOR is defined in STD
- Can index BIT_VECTOR with any NATURAL range
- We define unconstrained *integer_vector*

Utilities.arrays.unconstrained



- Syntax details of unconstrained array declarations

Utilities.arrays.unconstrained

```
PROCEDURE apply_data (
  SIGNAL target : OUT BIT_VECTOR;
  CONSTANT values : IN integer_vector; CONSTANT period : IN TIME)
IS
  VARIABLE buf : BIT_VECTOR (target'RANGE);
BEGIN
  FOR i IN values'RANGE LOOP
    int2bin (values(i), buf);
    target <= TRANSPORT buf AFTER i * period;
  END LOOP;
END apply_data;
```

* * *

- Example shows use of unconstrained:
  target : BIT_VECTOR
  values : integer_vector
- Use 'RANGE to look up passed range
- Range will be determined when procedure is called

Utilities.arrays.unconstrained

```
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;
```

- A new unconstrained comparator is shown
- Size of input vectors are not specified
- Comparator length is a'LENGTH
- Size will be determined when instantiated

Utilities.arrays.unconstrained

```
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;
```

- Testbench applies data to comparator
- SIGNAL *a* determines size of comparator
- SIGNAL *a* determines target size of *apply_data*
- SIGNAL *b* determines target size of *apply_data*
- Concatenated integers form *integer_vector*

Utilities.types.IO

File type declaration:

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

File declaration:

```
FILE input_logic_value_file : logic_data IS IN "input.dat";
```

Usage:

```
READ (input_logic_values_file, x_char);
```

* * *

- Declare a file type indicating type of data
- Declare a file indicating type, mode, logical_name
- Use READ, WRITE, ENDFILE on a declared file

Utilities.types.IO

```
-- File Type  logic_data  is Visible
 PROCEDURE assign_bits (
   SIGNAL target : OUT BIT; file_name : IN STRING; period : IN TIME)
  IS
   VARIABLE char : CHARACTER;
   VARIABLE current : TIME := 0 NS;
   FILE input_value_file : logic_data IS IN file_name;
  BEGIN
   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
         target <= TRANSPORT '0' AFTER current;
       ELSIF char = '1' THEN
         target <= TRANSPORT '1' AFTER current;
       END IF;
     END IF;
   END LOOP;
  END assign_bits;
```

* * *

- Example shows use of file for data input
- Read character, find binary values, schedule to target
- Using *assign_bits* in a testbench can be done as:

  assign_bits (a_signal, "afile.bit", 1500NS)

Utilities.overloading

- VHDL allows Overloading subprograms
- Overloading : Operand types determine exact function
- Use overloading to redefine operators
- Use overloading for various logic values systems
- Use overloading with user defined subprograms

Utilities.overloading

a:	0	1	Z	X
b: 0	0	0	0	0
1	0	1	1	X
Z	0	1	1	X
X	0	X	X	X

z=a.b

a:	0	1	Z	X
b: 0	0	1	1	X
1	1	1	1	1
Z	1	1	1	1
X	X	1	1	X

z=