COE 405
Utilities for High-Level Descriptions

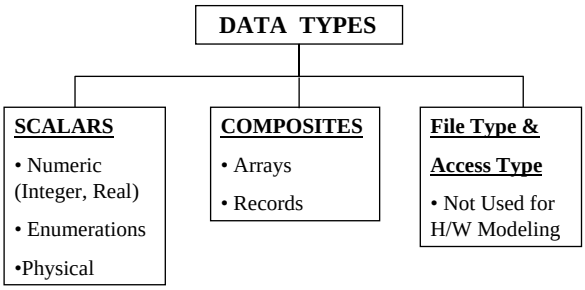
Dr. Aiman H. El-Maleh
Computer Engineering Department
King Fahd University of Petroleum & Minerals

Outline

- **Type Declaration and Usage**
  - Enumeration Type
  - Physical Type
  - Array Declaration
  - File Type
- **VHDL Operators**
- **Subprogram Parameter Types and Overloading**
- **Other Types and Types Related Issues**
  - Subtypes
  - Record Type
  - Access Type
  - Alias Declaration
  - Type Conversion
- **Predefined Attributes**
- **User Defined Attributes**

Data Types

- **A Data Type defines a set of values & a set of operations.**
- **VHDL is a strongly-typed Language. Types cannot be mixed in Expressions or in assigning values to Objects in general**



Scalar Data Types

- **SYNTAX**
  - TYPE Identifier IS Type-Definition
- **Numeric Data Type**
  - Type-Definition is a Range_Constraint as follows:
  - Type-Definition := Range Initial-Value < To | DownTo> Final-Value
- **Examples**
  - TYPE address IS RANGE 0 To 127;
  - TYPE index IS RANGE 7 DownTo 0;
  - TYPE voltage IS RANGE -0.5 To 5.5;

Number Formats

- **Integers have no Decimal Point.**
- **Integers may be Signed or Unsigned (e.g. -5 356)**
- **A Real number must have either a Decimal Point, a -ive Exponent Term (Scientific Notation), or both.**
- **Real numbers may be Signed or Unsigned (e.g. -3.75 1E-9 1.5E-12)**
- **Based Numbers:**
  - Numbers Default to Base 10 (Decimal)
  - VHDL Allows Expressing Numbers Using Other Bases
  - Syntax
    - B#nnnn# -- Number nnnn is in Base B
  - Examples
    - 16#DF2# -- Base 16 Integer (HEX)
    - 8#7134# -- Base 8 Integer (OCTAL)
    - 2#10011# -- Base 2 Integer (Binary)
    - 16#65_3EB.37# -- Base 16 REAL (HEX)

Predefined Numeric Data Types

- **Integer -- Range is Machine limited but At Least $-(2^{31})$ To $(2^{31} - 1)$**

  -2147483648 to 2147483647
- **Positive -- INTEGERS > 0**
- **Natural -- INTEGERS >= 0**
- **Real -- Range is Machine limited**

  -1.0E308 to 1.0E308

Enumeration Data Type

- **Parenthesized ordered list of literals.**
  - Each may be an identifier or a character literal.
  - The list elements are separated by commas
- **A Position # is associated with each element in the List**
- **Position #`s begin with 0 for the Leftmost Element**
- **Variables & Signals of type ENUMERATION will have the leftmost element as their Default (Initial) value unless, otherwise explicitly assigned.**
- **Examples**
  - TYPE Color IS (Red, Orange, Yellow, Green, Blue, Indigo, Violet);
  - TYPE Tri_Level IS (`0`, `1`, `Z`);
  - TYPE Bus_Kind IS (Data, Address, Control);
  - TYPE State IS (Init, Xmit, Receive, Wait, Terminal);

Predefined Enumerated Data Types

- **TYPE BIT IS (`0` , `1`) ;**
- **TYPE BOOLEAN IS (False, True) ;**
- **TYPE CHARACTER IS (128 ASCII Chars.....) ;**
- **TYPE Severity_Level IS (Note, Warning, Error, Failure) ;**
- **TYPE Std_U_Logic IS (**

```
`U` , -- Uninitialized
`X` , -- Forcing Unknown
`0` , -- Forcing 0
`1` , -- Forcing 1
`Z` , -- High Impedence
`W` , -- Weak Unknown
`L` , -- Weak 0
`H` , -- Weak 1
`-` , -- Don`t Care
) ;
```

- **SUBTYPE Std_Logic IS resolved Std_U_Logic ;**

Enumerated Data Types

- **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
  - Can develop logic gates using qit values system

NOT

In:	out
0	1
1	0
Z	0
X	X

NAND

In2: \ In1:	0	1	Z	X
0	1	1	1	1
1	1	0	0	X
Z	1	0	0	X
X	1	X	X	X

out

INV and NAND2 Gates Using *qit*

```
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;
```

Physical Data Type

- **Specifies a Range Constraint, one Base Unit, and 0 or more secondary units.**
- **Base unit is indivisible, i.e. no fractional quantities of the Base Units are allowed.**
- **Secondary units must be integer multiple of the indivisible Base Unit.**
- **Examples**

```
TYPE  Resistance  IS  Range  1  To  Integer'High
  Units
    Ohm;       -- Base Unit
    Kohm = 1000  Ohm;    -- Secondary  Unit
    Mohm = 1000  Kohm;  -- Secondary  Unit
  end  Units ;
```

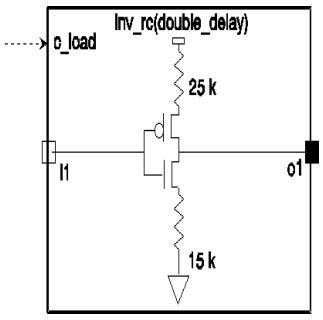

Predefined Physical Data Types

- **Time is the ONLY predefined Physical data type**

```
TYPE  Time  IS Range -2147483647 To 2147483647
  Units
    fs;   --      Base Unit (Femto Second = 1E-15 Second)
    ps = 1000  fs;   -- Pico_Second
    ns = 1000  ps;   -- Nano_Second
    us = 1000  ns;   -- Micro_Second
    ms = 1000  us;  -- Milli_Second
    sec = 1000  ms;-- Second
    min = 60  sec;   -- Minuite
    hr  = 60  min;   -- Hour
  end  Units ;
```

Example on Use of Real and Time

```
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;
```



- *Time is calculated from R and C values*

Physical Type Definition Examples

```
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;
```

Using Physical Types

```
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*

Composite Data Types: Arrays

- **Elements of an Array have the same data type**
- **Arrays may be Single/Multi - Dimensional**
- **Array bounds may be either Constrained or Unconstrained.**
- **Constrained Arrays**
  - Array Bounds Are Specified
  - Syntax:
    - **TYPE id Is Array (Range_Constraint) of Type;**
- **Examples**
  - TYPE word Is Array (0 To 7) of Bit;
  - TYPE pattern Is Array (31 DownTo 0) of Bit;
  - 2-D Arrays
    - **TYPE col Is Range 0 To 255;**
    - **TYPE row Is Range 0 To 1023;**
    - **TYPE Mem_Array Is Array (row, col) of Bit;**
    - **TYPE Memory Is Array (row) of word;**

Unconstrained Arrays

- **Array Bounds not specified through using the notation RANGE<>.**
- **Type of each Dimension is specified, but the exact Range and Direction are not Specified.**
- **Useful in Interface_Lists → Allows Dynamic Sizing of Entities , e.g. Registers.**
- **Bounds of Unconstrained Arrays in such entities assume the Actual Array Sizes when wired to the Actual Signals.**
- **Example**
  - TYPE Screen Is Array (Integer Range<> , Integer Range<>) of BIT;

Predefined Array Types

- **Two UNCONSTRAINED Array Types are predefined**
- **BIT_VECTOR**
  - TYPE Bit_Vector Is Array (Natural Range<>) of Bit;
- **String**
  - TYPE String Is Array (Positive Range<>) of Character;
- **Example**
  - SUBTYPE Pixel Is Bit_Vector (7 DownTo 0);

Use of Unconstrained Arrays

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

```
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*

Unconstrained Comparator

```
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;
```

- *Size of input vectors are not specified*
- *Comparator length is a'LENGTH*
- *Size will be determined when instatntiated*

Unconstrained Comparator Test Bench

```
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;
```

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

Referencing Arrays & Array Elements …

- **VHDL allows referencing an Array in its *Entirety* or by a *SLICE*, or *Element*.**
- **Examples**
  - TYPE clock_state IS (Low, Rising, High, Falling);
  - TYPE Conversion_Array IS Array (Clock_state) of Bit;
  - Signal C_A : Conversion_Array := (`0` , `1`, `0`, `1`) ;
  - C_A <= (`1`, `1`, `0`, `0`); -- Positional Association List
  - C_A <= (Low => `0`, Rising => `1`, High => `1`, Falling => `0`);
    -- Named Association List
  - C_A <= (Low => `0`, High => `0`, OTHERS=> `1`);
    -- Alternative 3
  - C_A(Low) <= `0`;
  - TYPE Register4 IS Array (3 Downto 0) of Bit;
  - TYPE Reg4 IS Array (0 To 3) of Bit;
  - Signal A: Register4 := (`0` , `1`, `0`, `1`) ;--A(0)=`1`, A(3)=`0`
  - Signal B: Reg4 := (`0` , `1`, `0`, `1`) ;--B(0)=`0`, B(3)=`1`

… Referencing Arrays & Array Elements

- **2-D Arrays**
  - TYPE Reg32 Is Array (31 DownTo 0) of Bit;
  - TYPE ROM Is Array (0 To 3) of Reg32;
  - TYPE ROM2 Is Array (0 To 4 , 0 To 2) of Bit;
  - Signal A: ROM := (X``2F3C_5456`` , X``FF32_E7B8`` , X``109A_BA15`` , X``FFFF_FFFF``);
  - Signal B: ROM2 := ((`1`, `0`, `0`), (`0` , `1`, `0`), (`0` , `1`, `1`, (`1` , `0`, `1`), (`1` , `1`, `1`)) ;
  - B(1 , 2) <= `0` ; -- Referencing a 2-D Array Element

Examples on Referencing Arrays …

- **TYPE qit IS (‘0’ , ‘1’ , ‘Z’ , ‘X’);**
  - 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 := "ZZZZZZZZ“ ;
  - SIGNAL sq16 : qit_word ;
  - SIGNAL sq_nibble_8 : qit_nibble_by_8 ;
  - sq8 <= "Z101000Z" ;
  - sq8 <= sq16 (11 DOWNTO 4); -- middle 8 bit slice of sq16 to sq8
  - **sq16** (**15** DOWNTO **12**) <= **sq8(**5 DOWNTO 2**)** ; **-- sq8 middle nibble into left 4 bit slice of sq16**
  - sq4 **<=** sq_nibble_8(2) **;**-- third nibble of sq_nibble_8 into sq4
  - **sq1** <= **sq_nibble_8 (2)(1)** ;

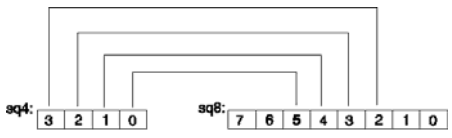
Direction of indexing must be as declared

… Examples on Referencing Arrays …

- **The following signal declarations are equivalent:**
  - Signal sq8 : qit_byte := “ZZZZ1111”;
  - Signal sq8 : qit_byte := (‘Z’, ’Z’, ’Z’, ‘Z’, ‘1’, ‘1’, ‘1’, ‘1’);
  - Signal sq8 : qit_byte := (‘Z’&’Z’&’Z’&‘Z’&‘1’&‘1’&‘1’&‘1’);
  - Signal sq8 : qit_byte := (7 Downto 4 => ‘Z’, OTHERS => ‘1’);
- **Index range can be in any direction as long as it covers valid array indexes**
  - Signal sq8 : qit_byte := “111XX1ZZ”;
  - Signal sq8 : qit_byte := (1 Downto 0 => ‘Z’, 3 To 4 => ‘X’, Others => ‘1’);

… Examples on Referencing Arrays …

- **TYPE qit IS (‘0’ , ‘1’ , ‘Z’ , ‘X’);**
  - 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 ;
    - sq8 <= sq8 (0) & sq8 (7 DOWNTO 1) ; -- right rotate sq8
    - sq4 <= sq8 (2) & sq8 (3) & sq8 (4) & sq8 (5) ; -- reversing sq8 into sq4



Concatenation operator “&”can be used for shift and rotate

… Examples on Referencing Arrays …

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

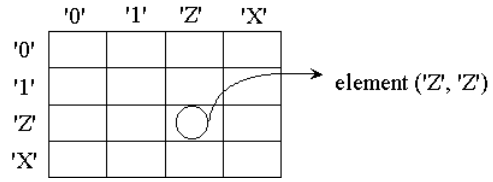
•Use nested parenthesis for multidimensional arrays
•Deepest set of parenthesis corresponds to right most index

… Examples on Referencing Arrays

- **Initializing a two-dimensional array**
  - SIGNAL sq_4_8 : qit_4by8 := (Others => “11000000”);
  - SIGNAL sq_4_8 : qit_4by8 := (Others => (Others => ‘Z’));
  - SIGNAL sq_4_8 : qit_4by8 := (Others => (0 To 1 => ‘1’, Others => ‘0’));
- **Signal assignment using Others**
  - sq_4_8 <= (3=> (Others => ‘X’), 0 => (Others => ‘X’), Others => (0 => ‘X’, 7 => ‘X’, Others => ‘1’));

Using Enumeration Types for Indexing

- **Can use enumeration types for indexing**
  - **TYPE qit_2d IS ARRAY (qit, qit) OF qit;**
  - Use qit for indexing
  - Upper left most element is indexed by ('0','0')



- **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

NAND2 Using *qit_2d*

```
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;
```

```
CONSTANT qit_nand2_table :
qit_2d := ( '0' => (Others =>'1'),
'X' => ('0' => '1', Others => 'X'),
Others => ('0' => '1', 'X' => 'X',
Others => '0'));
```

Records

- **Elements of a Record are Heterogeneous (not Necessarily of the Same Data Type).**
- **Examples**
  - TYPE opcode Is (STA, LDA, ADD, JMP);
  - TYPE mode Is Range 0 To 3;
  - SubType Address Is Bit_Vector(7 DownTo 0);
  - TYPE Instruction Is
    - Record
      OPC: opcode ;
      M : mode ; -- Addressing mode
      OP1, OP2 : Address ;
    - End record ;

Referencing Record Elements

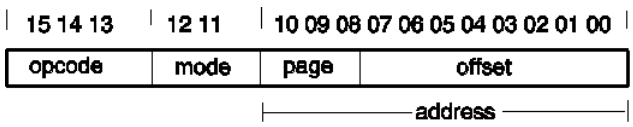
- **TYPE Instr_Q Is Array (0 To 15) of Instruction ;**
- **SIGNAL IQ : Instr_Q ;**
- **IQ(0) <= (LDA, 2, x``F3``, x``13``); --Positional Association**
- **Alternatively**
  - IQ(0).OPC <= LDA;
  - IQ(0).M <= 2;
  - IQ(0).OP1 <= X”F3”;
  - IQ(0).OP2 <= X``13`` ;
- **Alternatively**
  - IQ(0) <= (M => 2, OP2 => X``13``, OPC => LDA, OP1 => X``F3``); --Named Association

Aliasing …

- **Alias declaration selects part of an object**
- **Use for renaming, for convenience**
- **Alias declaration can be used for signals, variables, and constants**
- **Alias declaration specifies an identifier, its type, name of object identifier is alias of**
- **Example**
  - **ALIAS c_flag : BIT IS flag_register (3);**
  - **ALIAS v_flag : BIT IS flag_register (2);**
  - **ALIAS n_flag : BIT IS flag_register (1);**
  - **ALIAS z_flag : BIT IS flag_register (0);**

… Aliasing

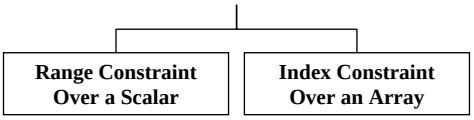
- **ALIAS page : BIT_VECTOR (2 DOWNTO 0) IS instr.adr (10 DOWNTO 8);**
- **ALIAS offset : BIT_VECTOR (7 DOWNTO 0) IS instr.adr (7 DOWNTO 0);**
- **page <= "001"; -- 3 bits**
- **offset <= X"F1"; -- 8 bits**
- **offset <= B"1111_0001"; -- 8 bits**



•Use aliasing to separately name page and offset
•All references to page refer to 10:8 of address
•All references to offset refer to 7:0 of address

Subtypes

- **A SUBTYPE defines a SUBSET of values defined by a TYPE declaration.**
- **Subtypes of Subtypes are also possible**
- **SUBTYPE = Constrained ``TYPE`` or ``SubType``**



Example (i) Range Constrained Subtypes

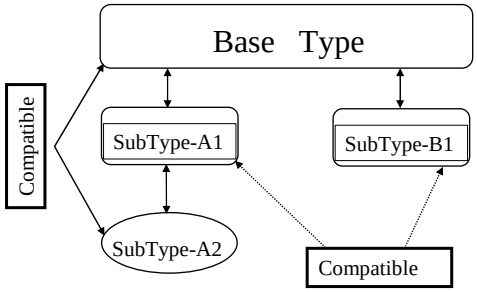
```
SubType  Lower_Case Is Character Range  `a`  To  `z`;
SubType Positive Is Integer Range  1  To  Integer`High;
SubType Natural Is  Integer Range  0  To  Integer`High;
```

Example (ii) Index Constrained Subtypes

```
SubType  Byte  Is  Bit_Vector  (7 DownTo 0);
```


Type Compatibility & Conversion

- **VHDL is a strongly-typed language.**
- **Compiler flags an Error whenever different types are mixed.**
- **Subtypes are Type compatible with their higher level Subtypes and their Root Type.**
- **Two Subtypes of the same type are Type-Compatible**



Important Type Compatibility Rules …

- **Type of an Expression assigned to an Object must be the Same as, or compatible with, the Type of the Object.**
- **Operands of Predefined Operators Must be of the Same or compatible Type**
- **The Type of an Actual Signal is The SAME as, or compatible with, the Type of the Formal Port it Connects To.**
- **Example**

```
SubType  X_int  Is  Integer  Range  1  To  30;
SubType  Y_int  Is  Integer  Range  1  To  30;
Type X2_int  Is Range  1  To  30;
Type Y2_int  Is Range  1  To  30;
SubType  Z_int  Is  X_int      Range  1  To  20;
```

```
Signal  I : Integer;
Signal  x : X_int;
Signal  y : Y_int;
Signal  z : Z_int;
Signal  x2 : X2_int;
Signal  y2 : Y2_int;
```


… Important Type Compatibility Rules

- **x <= z ;** **-- Legal Since Z_int is a Subtype of X**
- **x <= y ;** **--Legal**
- **x2 <= y2;** **--Illegal, Operands are not compatible**
- **I <= x + z ;** **-- Legal, both x and z are Vertical subtypes of Integer**
- **I <= x + y ;** **-- Legal, Operands are subtypes of same type**
- **I <= x2 + y2 ;** **-- Illegal, Operands are not compatible**
- **z <= x ;** **-- Legal, value of x should be within the range of z**
- **z <= y ;** **-- Legal, value of y should be within the range of z**

Closely Related Types …

- **Any two Numeric types are closely related**
- **Closely related types can be Casted to each other**
  - Assume I is integer and X is real
  - I := Integer(X); -- Rounds X
  - X :=Real(I);
  - I := Integer(3.5); -- Ambiguous (Implementation Dependent)
- **Array Types are closely related iff:**
  - Same dimensionality
  - Index Types are closely related for each dimension
  - Elements Types are the same or compatible

… Closely Related Types

- **Example**
  - Type Minuites Is Range 0 To 59;
  - Type Seconds Is Range 0 To 59;
  - SubType X_int Is Integer Range 1 To 30;

  - Variable x : X_int;
  - Variable M : Minuites;
  - Variable S : Seconds;
  - Variable I : Integer;

  - I := 60*M + S; -- Illegal M & S are incompatible types
  - I := 60*Integer(M) + Integer(S); -- Valid
  - I := M; -- Illegal – Minuites is not a “SubType” of Integer
  - M := 60*S; -- Illegal – Minuites & Seconds are not Compatible
  - M := Minuites(S/60); -- legal –Casting Closely Related Types
  - M := x; -- Illegal – Minuites & X_int are not Type Compatible
  - M := Minuites(x); -- legal –Casting Closely Related Types

Mixed Type Arithmetic

- **Explicit Type Conversion is done between Closely-Related Types, e.g. REALs & INTEGERs**
- **Example:**
  - Variable x, y : Real;
  - Variable n,m : Integer;
  - n := INTEGER (x) * m; -- x is first converted to Integer
  - y := REAL (n) * x; -- n is first converted to Real
- **Example:**
  - TYPE qit_byte IS ARRAY (7 DOWNTO 0) OF qit;
  - TYPE qit_octal IS ARRAY (7 DOWNTO 0) OF qit;
  - Signal qb: qit_byte;
  - Signal qo: qit_octal;
  - qb <= qit_byte(qo); - - Explicit Type Conversion (Type Casting)
- - of closely-related types
  - qb <= qit_byte(OTHERS => 'X'); - - Illegal type conversion

Custom Type Conversion

- **Custom Type Conversions can be defined Using either:**
  - Constant Conversion Arrays, or
  - Subprograms (Functions or Procedures)
- **Type conversion Arrays or Subprograms may be placed within packages, e.g. functions already in predefined standard packages may also be used**
  - e.g. the package std_logic_1164 defined within the ``ieee`` Library

Type Conversion Using Functions

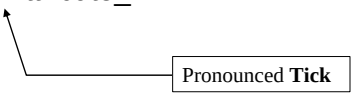
```
Type  MVL4 (`X`, `0`, `1`, `Z`);
Function MVL4_To_Bit(B: in MVL4)
Return Bit IS
Begin
  Case B is
    when `X` => return `0`;
    when `0` => return `0`;
    when `1` => return `1`;
    when `Z` => return `0`;
  End Case;
End MVL4_To_Bit ;
```

```
Function Bit_To_ MVL4
(B: in Bit) Return MVL4 IS
Begin
  Case B is
    when `0` => return `0`;
    when `1` => return `1`;
  End Case;
End MVL4_To_Bit ;
```

```
Signal B4: MVL4;
Signal  B: Bit;
   B <= MVL4_To_Bit(B4);
   B4 <= Bit_To_ MVL4 (B);
```


Type Attributes

- **A Predefined Attribute is a named Feature of a Type**
- **Value of Attribute is Referenced as:**
  - Type_Name`Attribute_ID



Attributes of Types and SubTypes

T'LEFT :	Left Bound of Scalar Type T.
T'RIGHT:	Right Bound of Scalar Type T.
T'HIGH:	Upper Bound of Scalar Type T, i.e. highest value
T'LOW:	Lower Bound of Scalar Type T, i.e. lowest value
T'POS(V):	Position of value V in Type T
T'VAL(P):	Value of Type T at Position P
T'SUCC(V):	Value after value V in Type T, i.e., if V at position X in base of type, SUCC(V) will be the value at position X+1
T'PRED(V):	Value before value V in Type T, i.e., if V at position X in base of type, PRED(V) will be the value at position X-1
T'LEFTOF(V):	Value left of value V in Type T
T'RIGHTOF(V):	Value right of value V in Type T
T'BASE:	Base Type of Type T. Only Legal When Used As Prefix To Another Attribute; e.g.. T'BASE'RIGHT.

Attributes of Types and SubTypes

- **EXAMPLEs**
  - TYPE UP IS Range 0 To 7;
  - TYPE Down IS Range 7 DownTo 0;
  - TYPE Color IS (Red, Orange, Yellow, Green, Blue, Indigo, Violet);

    0 1 2 3 4 5 6
  - SubType ColorUP Is Color Range Orange To Indigo;
  - SubType ColorDN Is Color Range Indigo DownTo Orange;

Attributes of Types and SubTypes

Color`Pos(Green) = 3	**Color`VAl(3) = Green**
Color`Left = Red** **ColorUP`Left = Orange **ColorDN`Left = Indigo**	**Color`Right = Violet** **ColorUP`Right = Indigo** **ColorDN`Right = Orange**
UP`Left = 0** **Down`Left = 7	**UP`Right = 7** **Down`Right = 0**
UP`Low = 0** **Down`Low = 0 **Color`Low = Red** **ColorUP`Low = Orange** **ColorDN`Low = Orange**	**UP`High = 7** **Down` High = 7** **Color` High = Violet** **ColorUP` High = Indigo** **ColorDN` High = Indigo**
UP`Succ(6) = 7** **UP`Pred(6) = 5 **ColorUP` Succ(Blue)= Indigo** **ColorUP` Pred(Blue)= Green**	**UP`RightOf(6) = 7** **UP`LeftOf(6) = 5** **ColorUP` RightOf (Blue)= Indigo** **ColorUP` LeftOf (Blue)= Green**
Down`Succ(6) = 7** **Down `Pred(6) = 5 **ColorDN` Succ(Blue)= Indigo** **ColorDN` Pred(Blue)= Green**	**Down `RightOf(6) = 5** **Down `LeftOf(6) = 7** **ColorDN` RightOf (Blue)= Green** **ColorDN` LeftOf (Blue)= Indigo**
For Descending Ranges	**Pred =** ***RightOf*** **Succ =** ***LeftOf***

Examples

- TYPE qit IS ('0', '1', 'Z', 'X');
- SUBTYPE tit IS qit RANGE '0' TO 'Z';
  - Tit'Base qit
  - Tit'Left '0'
  - Qit'Left '0'
  - Tit'Right 'Z'
  - Qit'Right 'X'
  - Tit'High 'Z'
  - Qit'Low '0'
  - Qit'Pos('X') 3
  - Tit'Pos('X') Error, out of range
  - Qit'Val(3) 'X'
  - Tit'Val(3) Error, out of range
  - Tit'Leftof('1') '0'
  - Tit'Leftof('0') Error
  - Tit'Rightof('Z') Error, out of range

Array Attributes

- **Find Range, Length, Boundary of an Array Object or Array Type**
- Follow attribute by ( ) to specify index

A'LEFT(N)	Left Bound of The Nth Index of the Array Object or Subtype. Optional Parameter. Default is 1.
A'RIGHT(N)	Right Bound of The Nth Index of Array Object or Subtype. Optional Parameter. Default is 1.
A'HIGH(N)	Upper Bound of Nth Index of Array Object or Subtype. Optional Parameter. Default is 1.
A'LOW(N)	Lower Bound of Nth Index of Array Object or Subtype. Optional Parameter. Default is 1.
A'RANGE(N)	Range of Nth Index of Array Object or Constrained Array Subtype Ascending: *Left Bound* To *Right Bound* Descend: *Left Bound* Downto *Right Bound*
A'REVERSE _ RANGE(N)	Identical To A'RANGE(N) *Except* Range Is Reverse; *i.e.*. Ascending, *Right Bound* Downto *Left Bound* Descending, *Right Bound* To *Left Bound*.
A'LENGTH(N)	Number of Values In The Nth Index Of Array Object or Constrained Array Subtype. Optional Parameter. Default is 1.

Array Attributes

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

Attribute	Description	Example		Result
'LEFT	Left bound	sq_4_8'LEFT(1)	3	
'RIGHT	Right bound	sq_4_8'RIGHT	0	
		sq_4_8'RIGHT(2)	7	
'HIGH	Upper bound	sq_4_8'HIGH(2)	7	
'LOW	Lower bound	sq_4_8'LOWS(2)	0	
'RANGE	Range	sq_4_8'RANGE(2)	0	TO 7
		sq_4_8'RANGE(1)	3	DOWNTO 0
'REVERSE_RANGE	Reverse Range	sq_4_8'REVERSE_RANGE(2)	7	DOWNTO 0
		sq_4_8'REVERSE_RANGE(1)	0	TO 3
'LENGTH	Length	sq_4_8'LENGTH	4	

Signal Attributes

- **SIGNAL s1 : BIT;**

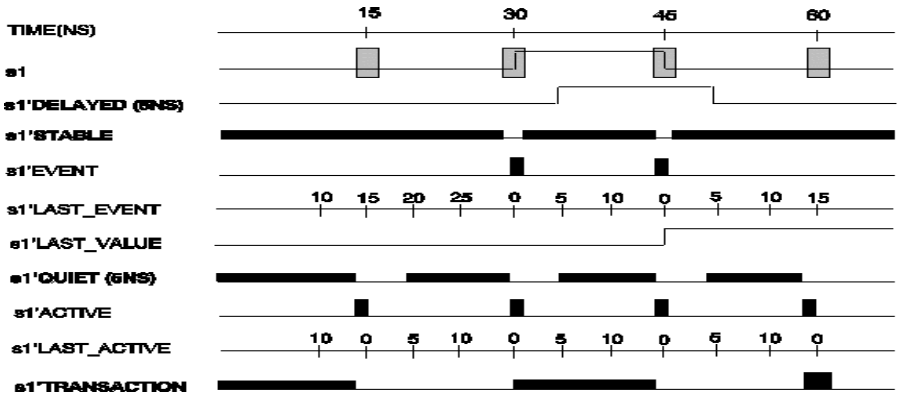
Attribute	T/E	Example	Kind	Type
		Description		
'DELAYED	–	s1'DELAYED(5 NS)	SIGNAL	As s1
A copy of s1, but delayed by 5 NS. If no parameter or 0, delayed by delta. Equivalent to TRANSPORT delay of s1.				
'STABLE	EV	s1'STABLE(5 NS)	SIGNAL	BOOLEAN
A signal that is TRUE if s1 has not changed in the last 5 NS. If used with no parameter or 0, the resulying signal is TRUE if s1 has not changed in the current simulation time.				
'EVENTS	EV	s1'EVENT	Value	BOOLEAN
If s1 changes in the current simulation cycle, s1'EVENT will be TRUE for this cycle (delta time).				
'LAST_EVENT	EV	s1'LAST_EVENT	Value	Time
The amount of time since the last value change on s1. If s1'EVENT is TRUE, the value of s1'LAST_EVENT is 0.				
'LAST_VALUE	EV	s1'LAST_VALUE	Value	As s1
The value of s1 before the most recent event occurs on it.				

Signal Attributes

Attribute	T/E	Example	Kind	Type
		Description		
'QUIET	TR	s1'QUIET(5 NS)	SIGNAL	BOOLEAN
A signal that is TRUE if no transaction has been placed on s1 in the last 5 NS. If no parameter of 0, for current simulation cycle is assumed.				
'ACTIVE	TR	s1'ACTIVE	Value	BOOLEAN
If s1 has had a transaction in the current simulation cycle, s1'ACTIVE will be TRUE for this simulation cycle, for * time.				
'LAST_ACTIVE	TR	s1'LAST_ACTIVE	Value	Time
Tthe amount of time since the last transaction occurred on s1. If s1'ACTIVE is TRUE, s1'LAST_ACTIVE is 0.				
'TRANSACTION	TR	s1'TRANSACTION	SIGNAL	BIT
A signal that toggles each time a transaction occurs on s1.				

Signal Attribute Examples

- **Attributes resulting in signal are in bold**
- **Attributes resulting in BIT are** 
- **Attributes resulting in BOOLEAN are** 



Signal Attributes Example

```
ENTITY find_out IS END find_out;
ARCHITECTURE comparing OF find_out IS
SIGNAL c, x1, x2, diff : BIT :='0';
BEGIN
c <= '0', '1' AFTER 60 NS, '0' AFTER 120 NS;
x1 <= '1' AFTER 6 NS WHEN c'EVENT ELSE x1;
x2 <= '1' AFTER 6 NS WHEN NOT c'STABLE ELSE x2;
diff <= x1 XOR x2;
END comparing;
```

ns	c	x1	x2	diff	c'STABLE
0	0	0	0	0	true
60	**1**	0	0	0	**false**
60+d	1	0	0	0	**true**
66	1	1	0	0	true
66+d	1	1	0	1	true
120	**0**	1	0	1	**false**
120+d	0	1	0	1	**true**

•c'EVENT is value
•c'STABLE is signal
•c'STABLE changes twice for a change on c

Falling-Edge Triggered D-FF Example

```
ENTITY brief_d_flip_flop IS
PORT (d, c : IN BIT; q : OUT BIT);
END brief_d_flip_flop;
ARCHITECTURE falling_edge OF brief_d_flip_flop IS
SIGNAL tmp : BIT;
BEGIN
tmp <= d WHEN (c = '0' AND NOT c'STABLE) ELSE tmp;
q <= tmp AFTER 8 NS;
END falling_edge;
```

```
q <= d AFTER 8 ns WHEN (c = '0' AND NOT c'STABLE) ELSE q;
```

incorrect

```
q <= d AFTER 8 ns WHEN (c = '0' AND c'EVENT) ELSE q;
```

correct

Toggle FF Example

```
ENTITY brief_t_flip_flop IS
PORT (t : IN BIT; q : OUT BIT);
END brief_t_flip_flop;
ARCHITECTURE toggle OF brief_t_flip_flop IS
SIGNAL tmp : BIT;
BEGIN
tmp <= NOT tmp WHEN ( (t = '0' AND NOT t' STABLE)
AND (t'DELAYED' STABLE(20 NS)) ) ELSE tmp;
q <= tmp AFTER 8 NS;
END toggle;
```

- *T flip flop toggles when pulse on t ≥ 20 NS*
- *t = '0' AND NOT t'STABLE checks fall on t*
- *t' DELAYED'STABLE (20 NS) check for 20 NS before fall*
- *t'DELAYED is attributed because it is signal*

File Type & External File I/O …

- **Specifying files is a two step process of**
  - File type declaration
  - File declaration
- **Data is associated with an identifier that is defined as a file type.**
- **File Type Declaration Example**
  - Type TEXT is FILE of string;
  - Type logic_data is FILE of Character;
- **File Declaration Examples**
  - FILE file1 : logic_data; -- file must be opened to be associated with a physical file
  - FILE file2 : logic_data IS “Input.dat”;
  - FILE file3 : logic_data OPEN READ_MODE IS “Input.dat”;
    - File can be opened in READ_MODE, WRITE_MODE, APPEND_MODE

… File Type & External File I/O …

  - FILE file4 : logic_data OPEN WRITE_MODE IS “output.dat”;
- **Opening and Closing Files**
  - FILE_OPEN(file1, “input.dat”, READ_MODE);
  - FILE_OPEN(file1, “output.dat”, WRITE_MODE);
  - FILE_CLOSE(file1);
- **File Read and Write Operations**
  - VHDL provides three operations READ, WRITE and ENDFILE for file types
  - READ takes an object of file data type as its argument, reads next data from file to its data argument
  - WRITE takes an object of file data type as its argument, write next data from its data argument to file
  - ENDFILE takes an object of file data type as its argument and returns TRUE if a subsequent read cannot be done

…File Type & External File I/O …

```
Entity filetry is end;
Architecture example of filetry IS
  Type cfile is file of character;
File f1i, f1o : cfile;
Begin
process
 variable char: character;
begin
-- opening files
 FILE_OPEN (f1i, "f1i.txt", READ_MODE);
 FILE_OPEN (f1o, "f1o.txt", WRITE_MODE);
while not endfile(f1i) loop
  read (f1i, char); write (f1o, char);
 end loop;
 -- closing files
 FILE_CLOSE (f1i);  FILE_CLOSE (f1o);
wait;
 end process;
End example;
```


… File Type & External File I/O

```
-- 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  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;
```

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
- Use *qit* AND, OR, NOT table to overload operators

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

z=a.b

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

z=a+b

a:	
0	1
1	0
Z	0
X	X

z=a'

… Overloading

```
-- Type and subprogram declarations:
TYPE qit IS ('0', '1', 'Z', 'X');
TYPE qit_2d IS ARRAY (qit, qit) OF qit;
TYPE qit_1d IS ARRAY (qit) OF qit;
FUNCTION "AND" (a, b : qit) RETURN qit;
FUNCTION "OR" (a, b : qit) RETURN qit;
FUNCTION "NOT" (a : qit) RETURN qit;
```

```
FUNCTION "NOT" (a : qit)
RETURN qit IS
CONSTANT qit_not_table :
qit_1d := ('1','0','0','X');
BEGIN
RETURN qit_not_table (a);
END "NOT";
```

```
FUNCTION "OR" (a, b : qit) RETURN qit IS
CONSTANT qit_or_table : qit_2d := (
('0','1','1','X'),
('1','1','1','1'),
('1','1','1','1'),
('X','1','1','X'));
BEGIN
RETURN qit_or_table (a, b);
END "OR";
```

New Gates Based on Qit

```
USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: qit, "NOT"
ENTITY inv_q IS
GENERIC (tplh : TIME := 5 NS; tphl :
TIME := 3 NS);
PORT (i1 : IN qit; o1 ; OUT qit);
END inv_q;
ARCHITECTURE average_delay OF
inv_q IS
BEGIN
o1 <= NOT i1 AFTER (tplh + tphl) / 2;
END average_delay;
```

```
USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: qit,
"AND"
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
BEGIN
o1 <= NOT ( i1 AND i2 ) AFTER
(tplh + tphl) / 2;
END average_delay;
```


Overload “*” To Calculate RC Time

```
-- Subprogram declaration:
FUNCTION "*" (a : resistance; b : capacitance) RETURN TIME;
-- Subprogram body
FUNCTION "*" (a : resistance; b : capacitance) RETURN TIME IS
BEGIN
RETURN ( ( a / ohms) * ( b / 1 ffr ) * 1 ns ) / 1000000;
END "*";
```

- *Turn resistance to ohms*
- *Turn capacitance to its base unit*

Overload “*” To Calculate RC Time

```
USE WORK.basic_utilities.ALL;
-- FROM PACKAGE USE: qit, capacitance, resistance, "*"
ENTITY inv_rc IS
   GENERIC (c_load : capacitance := 66 ffr);
   PORT (i1 : IN qit; o1 : OUT qit);
   CONSTANT rpu : resistance := 25 k_o;
   CONSTANT rpd : resistance := 15 k_o;
END inv_rc;
--
USE WORK.basic_utilities.ALL;
ARCHITECTURE double_delay OF inv_rc IS
   CONSTANT tplh : TIME := rpu * c_load * 3;
   CONSTANT tphl : TIME := rpd * c_load * 3;
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;
```