Chapter 6
VHDL Language Utilities and Packages

VHDL Language Utilities and Packages

VHDL Language Utilities and Packages

6.3 Operator and Subprogram Overloading
- 6.3.1 Operator Overloading
- 6.3.2 Subprogram Overloading

6.4 Other Types and Type-Related Issues
- 6.4.1 Subtypes
- 6.4.2 Record Types
- 6.4.3 Alias Declaration
- 6.4.4 Access Types
- 6.4.5 Global Objects
- 6.4.6 Type Conversions
- 6.4.7 Standard Nine-Value Logic

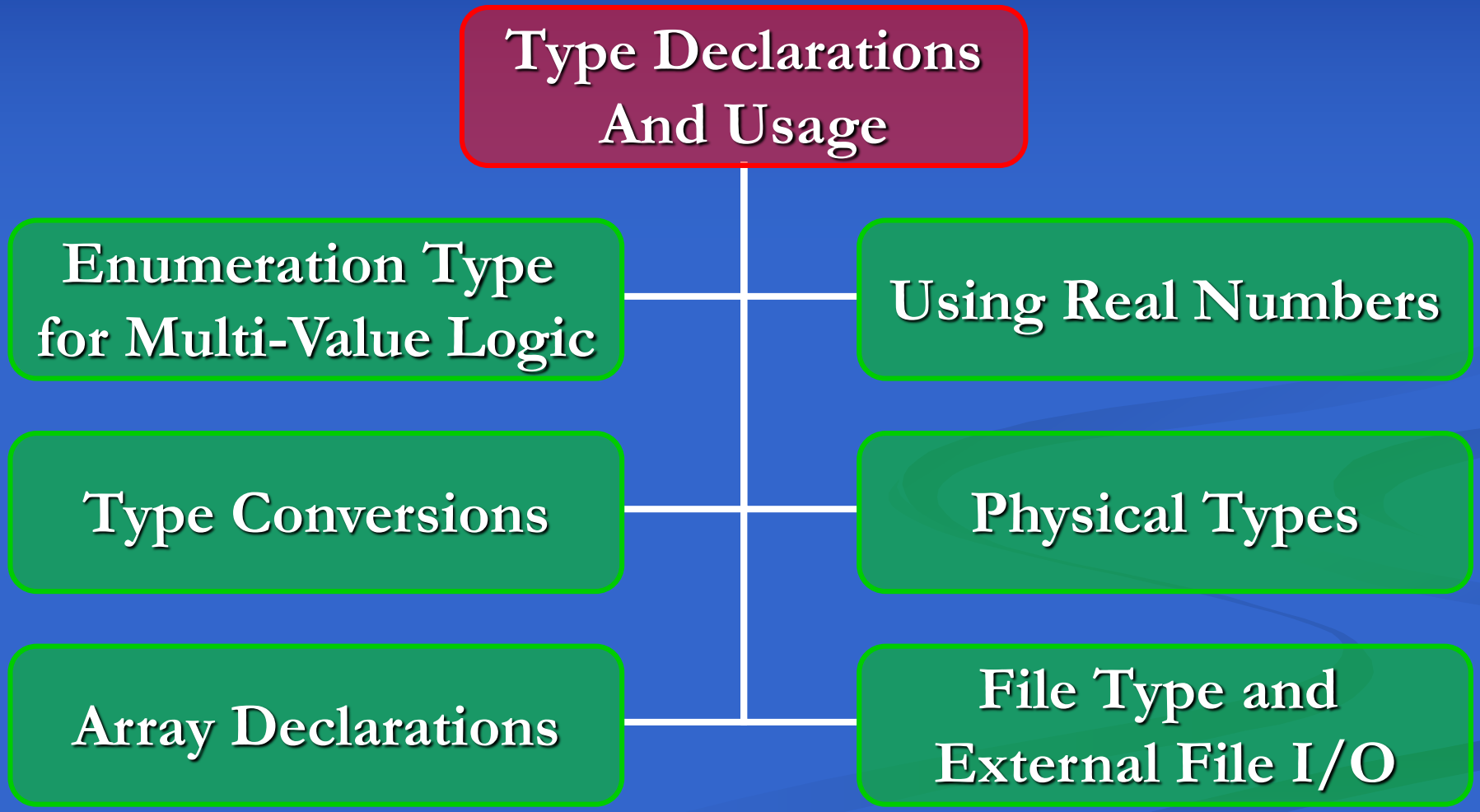
VHDL Language Utilities and Packages

6.5 Predefined Attributes
- 6.5.1 Array Attributes
- 6.5.2 Type Attributes
- 6.5.3 Signal Attributes
- 6.5.4 Entity Attributes
- 6.5.5 User-Defined Attributes

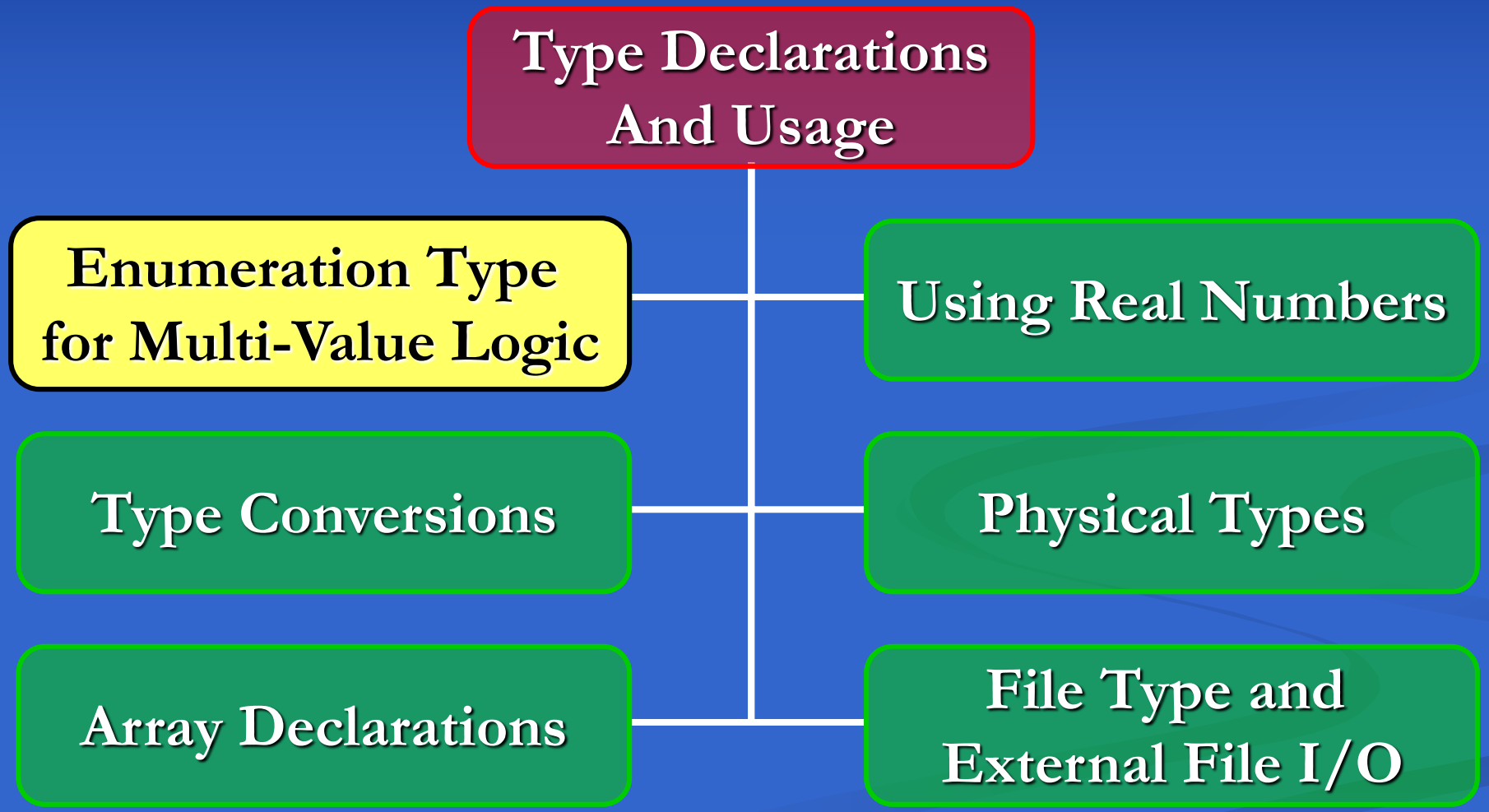
6.6 Standard Libraries and Packages
- 6.6.1 STANDARD Package
- 6.6.2 TEXTIO Package and ASCII I/O
- 6.6.3 Std_logic_1164 Package
- 6.6.4 Std_logic_arith Package

6.7 Summary

Type Declarations And Usage



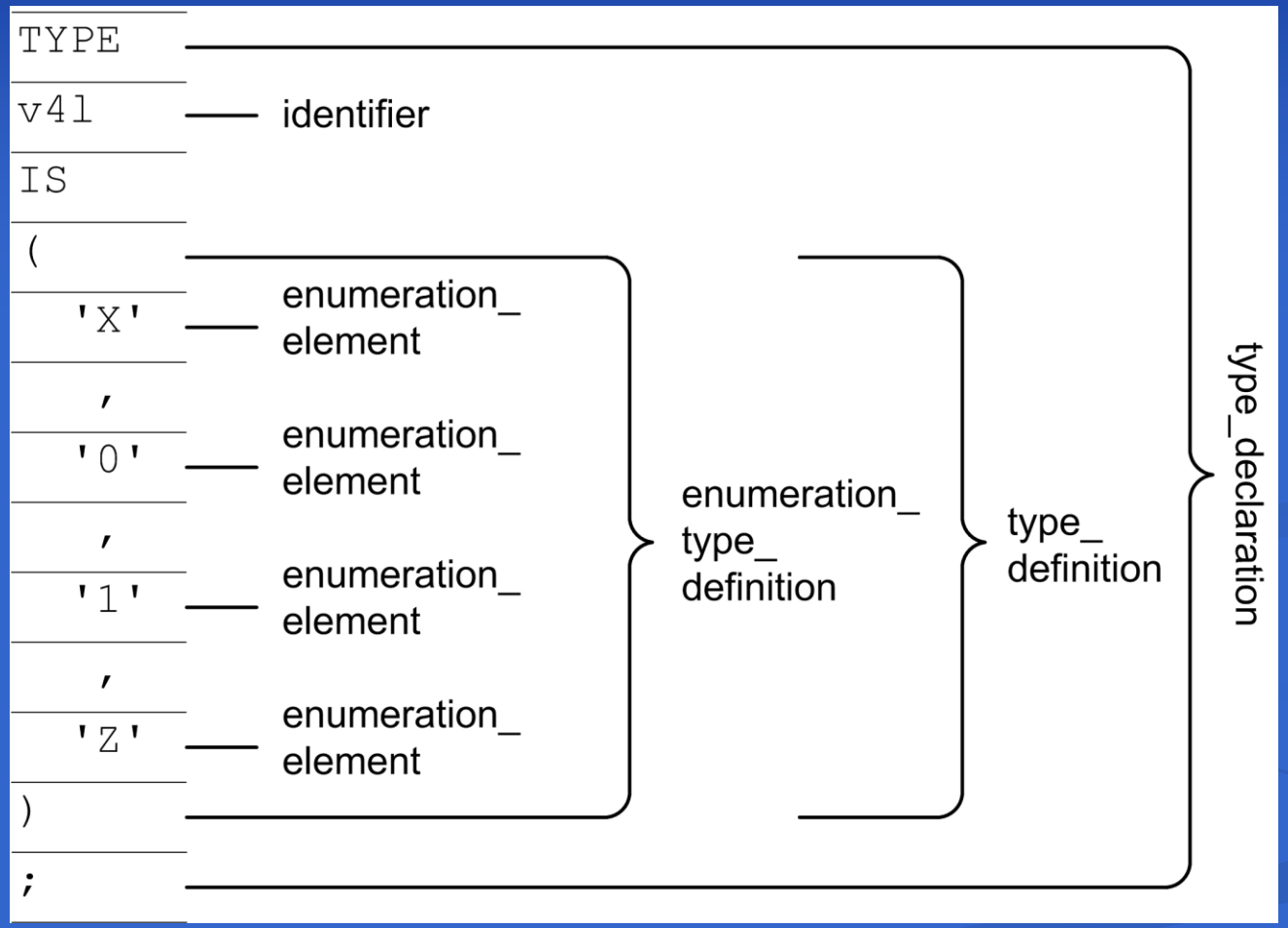
Enumeration Type for Multi-Value Logic



Enumeration Type for Multi-Value Logic

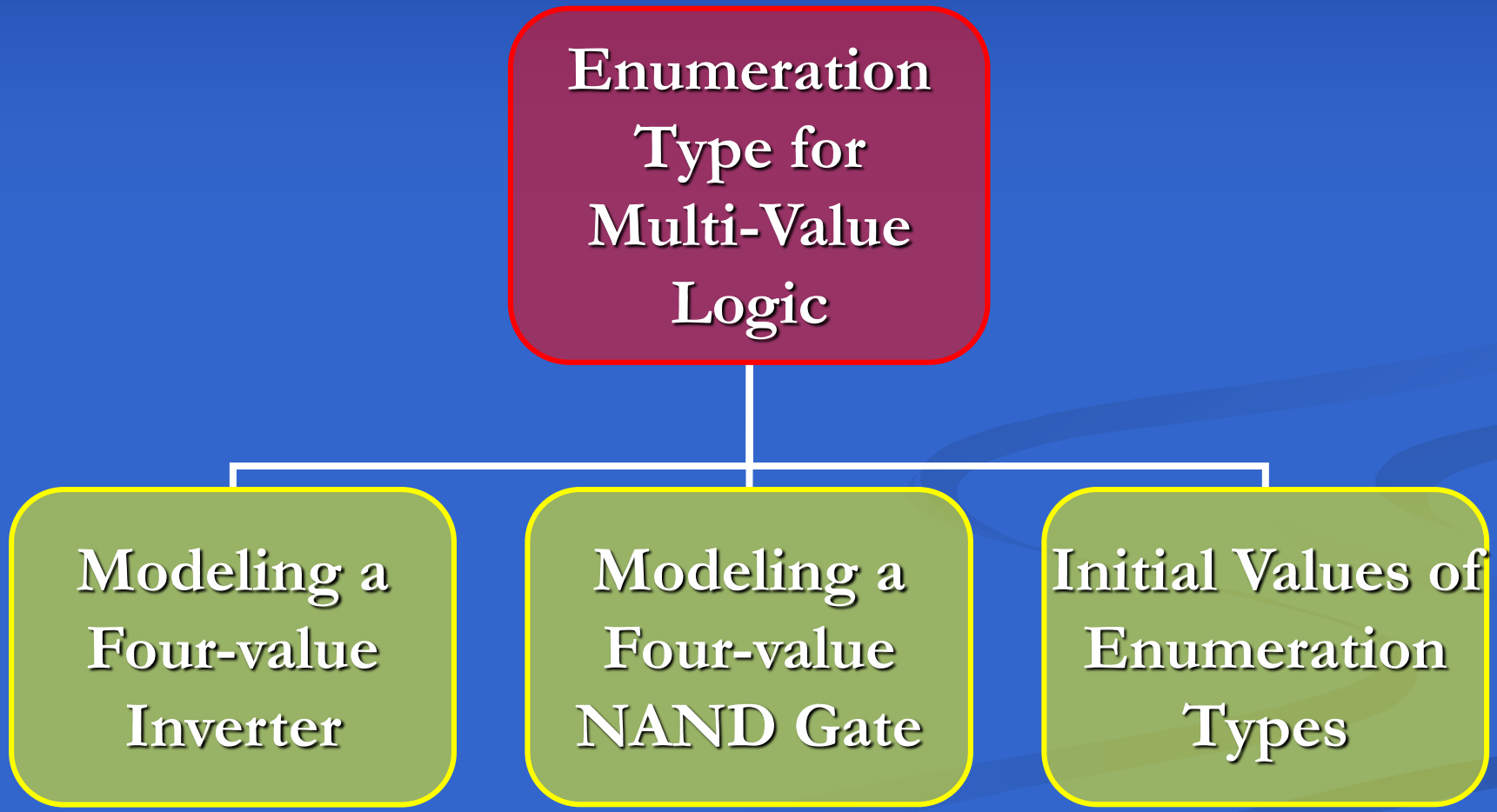
```
TYPE std_logic IS
      ('U','X','0','1','Z','W','L','H','-');

TYPE v4l IS ('X','0','1','Z');
```

Enumeration Type for Multi-Value Logic

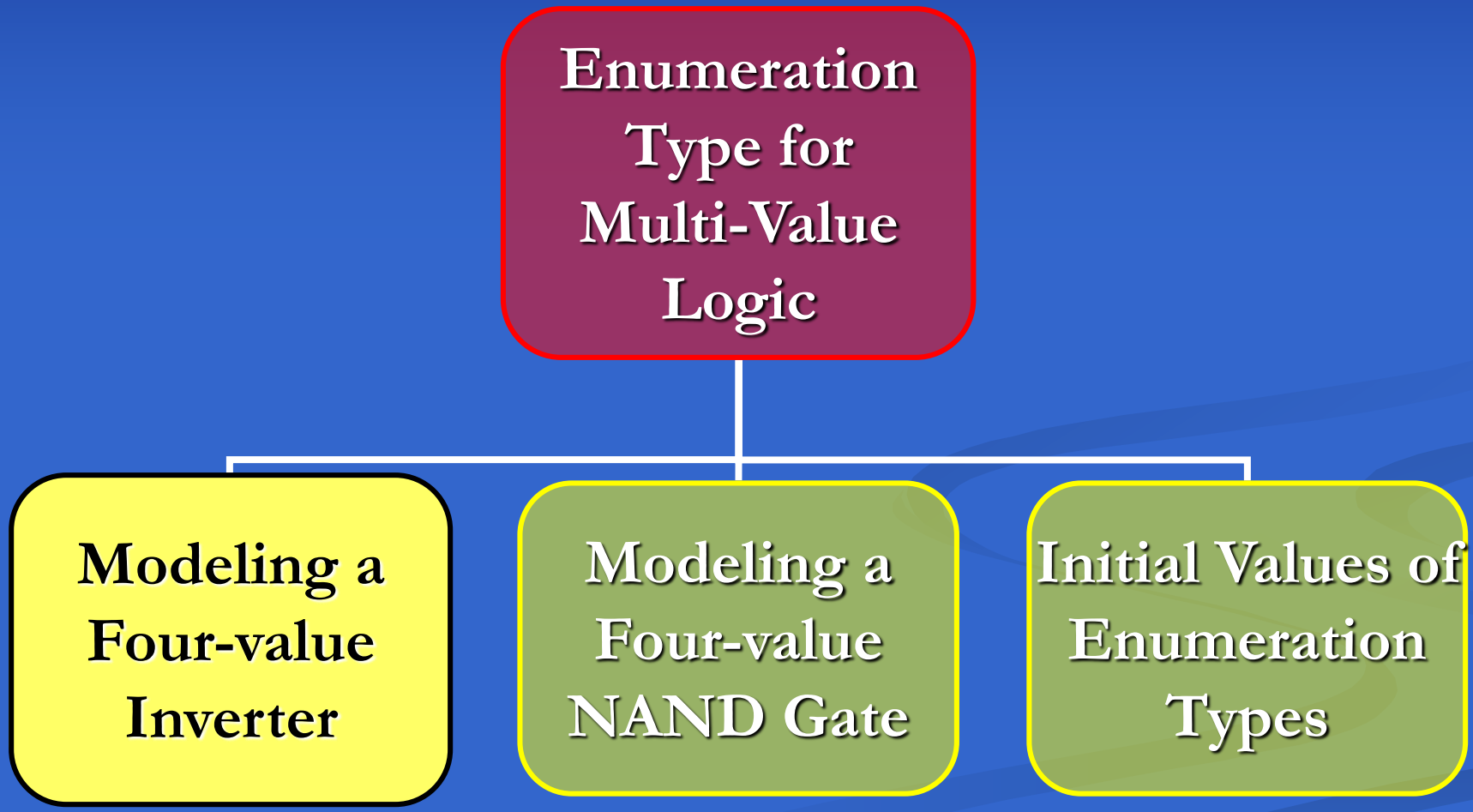


- **Syntax Details of a Type Declaration**

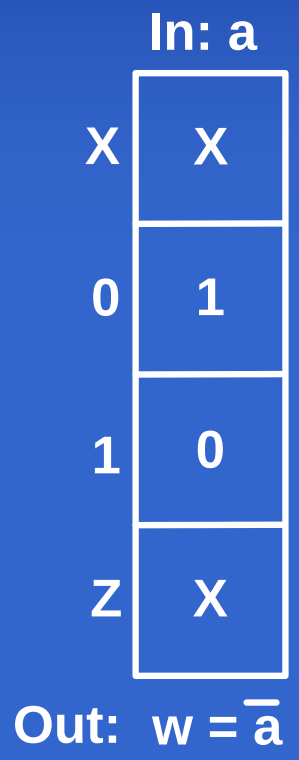
Enumeration Type for Multi-Value Logic



Modeling a Four-value Inverter



Modeling a Four-value Inverter



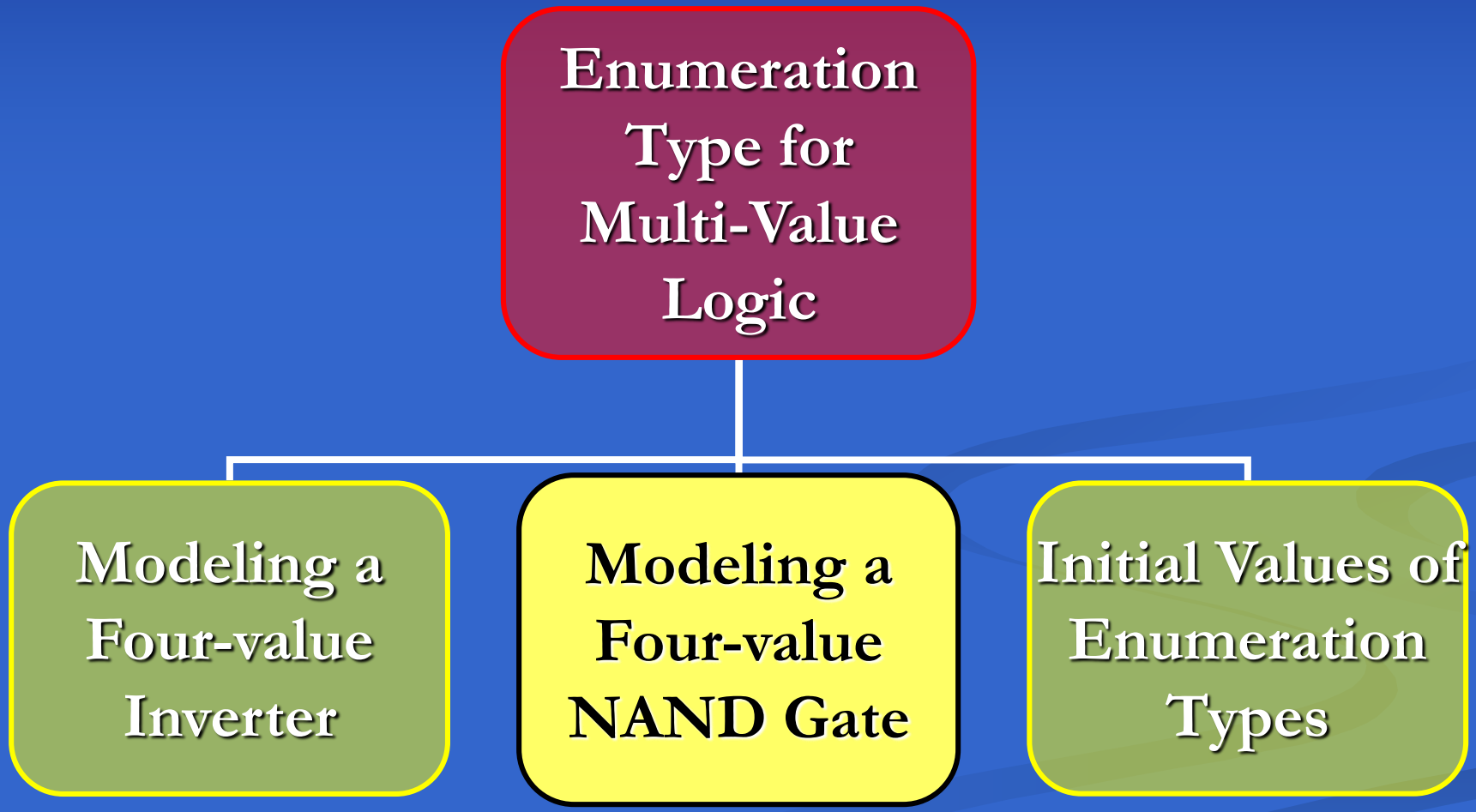
- Input-Output Mapping of an Inverter in *v41* Logic Value System

Modeling a Four-value Inverter

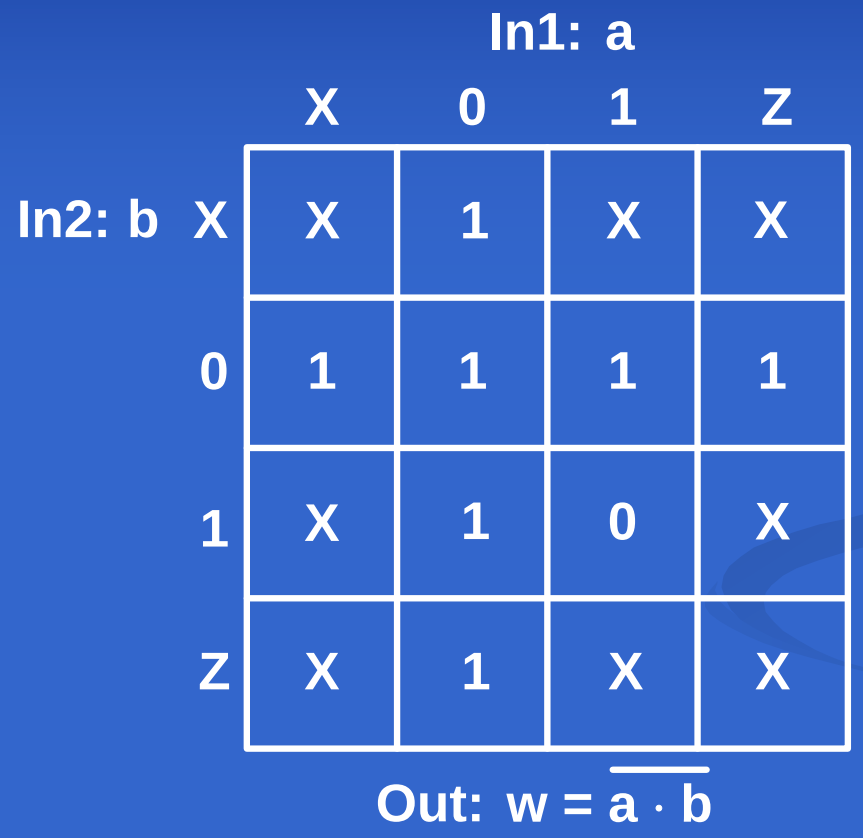
```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;
ENTITY vlog_inv IS
   GENERIC (tplh, tphl : TIME := 0 NS);
   PORT (w : OUT v4l; a : IN v4l);
END ENTITY vlog_inv;
--
ARCHITECTURE conditional OF vlog_inv IS
BEGIN
   w <= '1' AFTER tplh WHEN a = '0' ELSE
        '0' AFTER tphl WHEN a = '1' ELSE
        'X' AFTER tplh;
END ARCHITECTURE conditional;
```

- VHDL Description of an Inverter in *v4l* Logic Value System

Modeling a Four-value NAND Gate



Modeling a Four-value NAND Gate



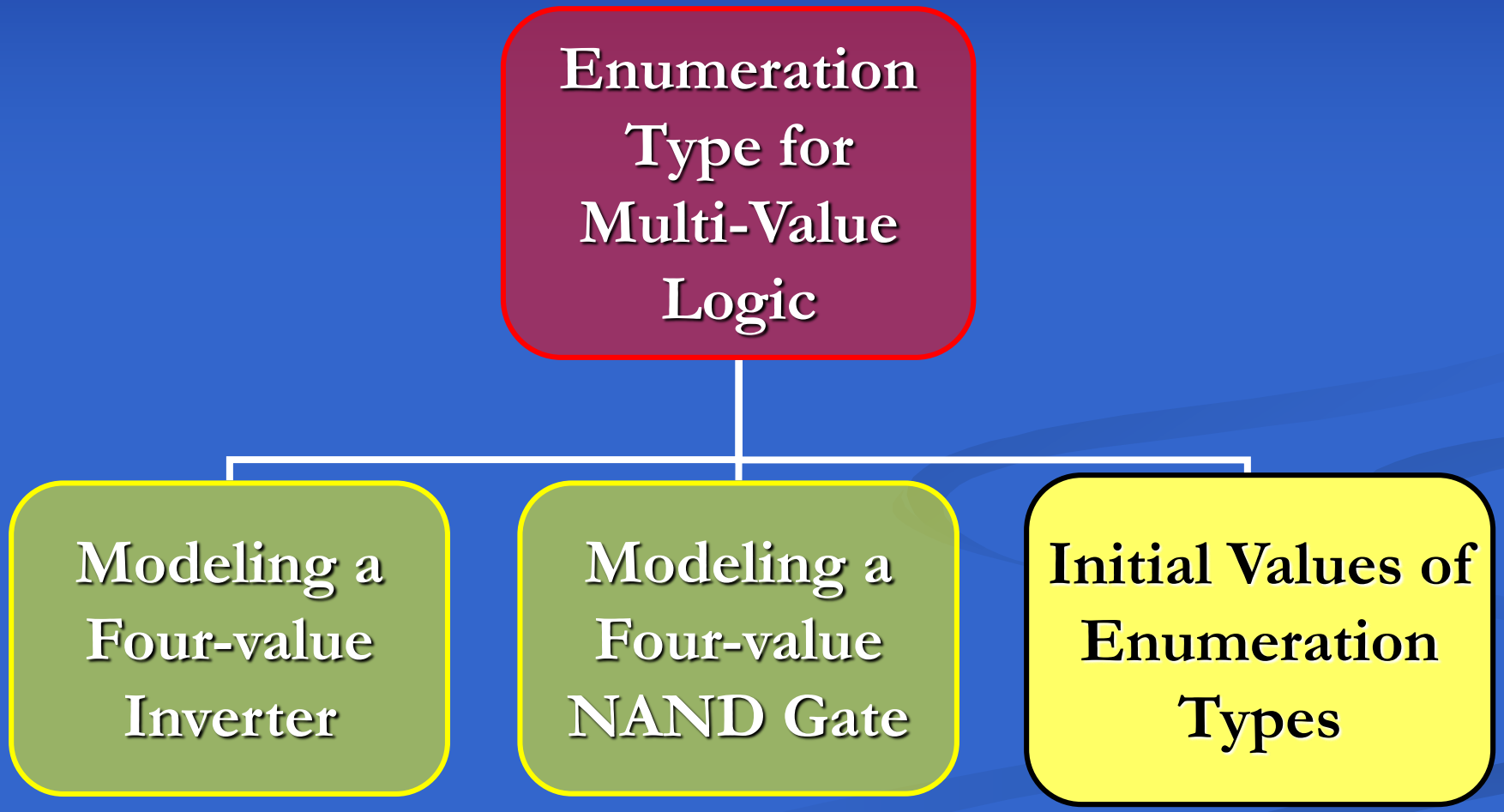
- Input-Output Mapping of a NAND Gate in *v4l* Logic Value System

Modeling a Four-value NAND Gate

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;
ENTITY vlog_nand2 IS
   GENERIC (tplh, tphl : TIME := 0 NS);
   PORT (w : OUT v4l; a, b : IN v4l);
END ENTITY vlog_nand2;
--
ARCHITECTURE conditional OF vlog_nand2 IS
BEGIN
   w <= '1' AFTER tplh WHEN (a='1') NAND (b='1') ELSE
        '0' AFTER tphl WHEN (a='1') AND (b='1') ELSE
        'X' AFTER tplh;
END ARCHITECTURE conditional;
```

- VHDL Description of a NAND Gate in *v4l* Logic Value System

Initial Values of Enumeration Types



Initial Values of Enumeration Types

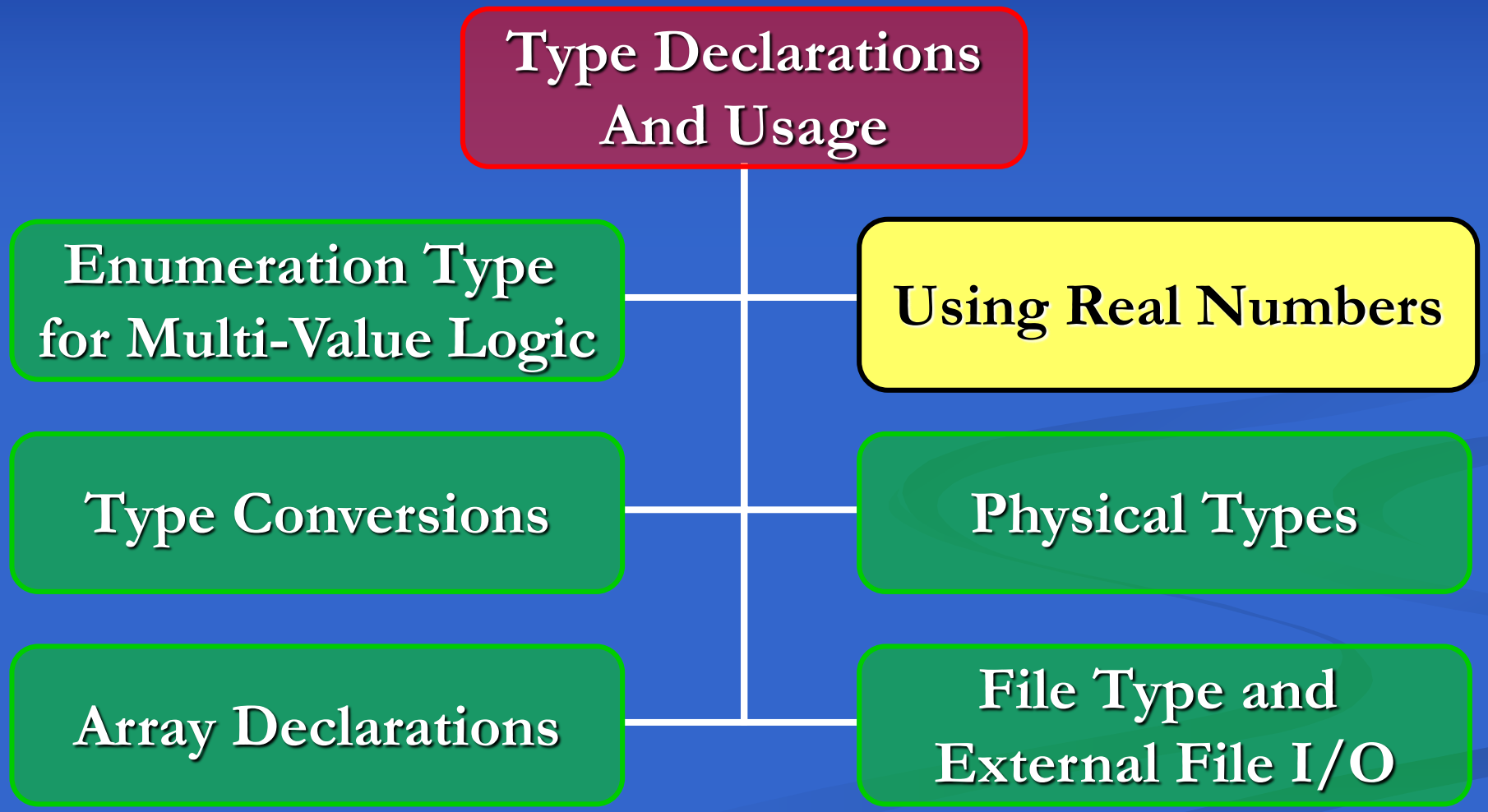
```
TYPE v41 IS ('X','0','1','Z');
```

The left-most Element of *v41* Type

```
TYPE v411 IS ('Z','0','1','X');
```

The left-most Element of *v411* Type

Using Real Numbers



Using Real Numbers

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;

ENTITY cmos_not IS
    GENERIC (c_load : REAL := 0.066E-12); --Farads
    PORT (w : OUT v41; a : IN v41);
    CONSTANT rpu : REAL := 3000.0; --Ohms
    CONSTANT rpd : REAL := 2100.0; --Ohms
END ENTITY cmos_not;
. . . . . .
```

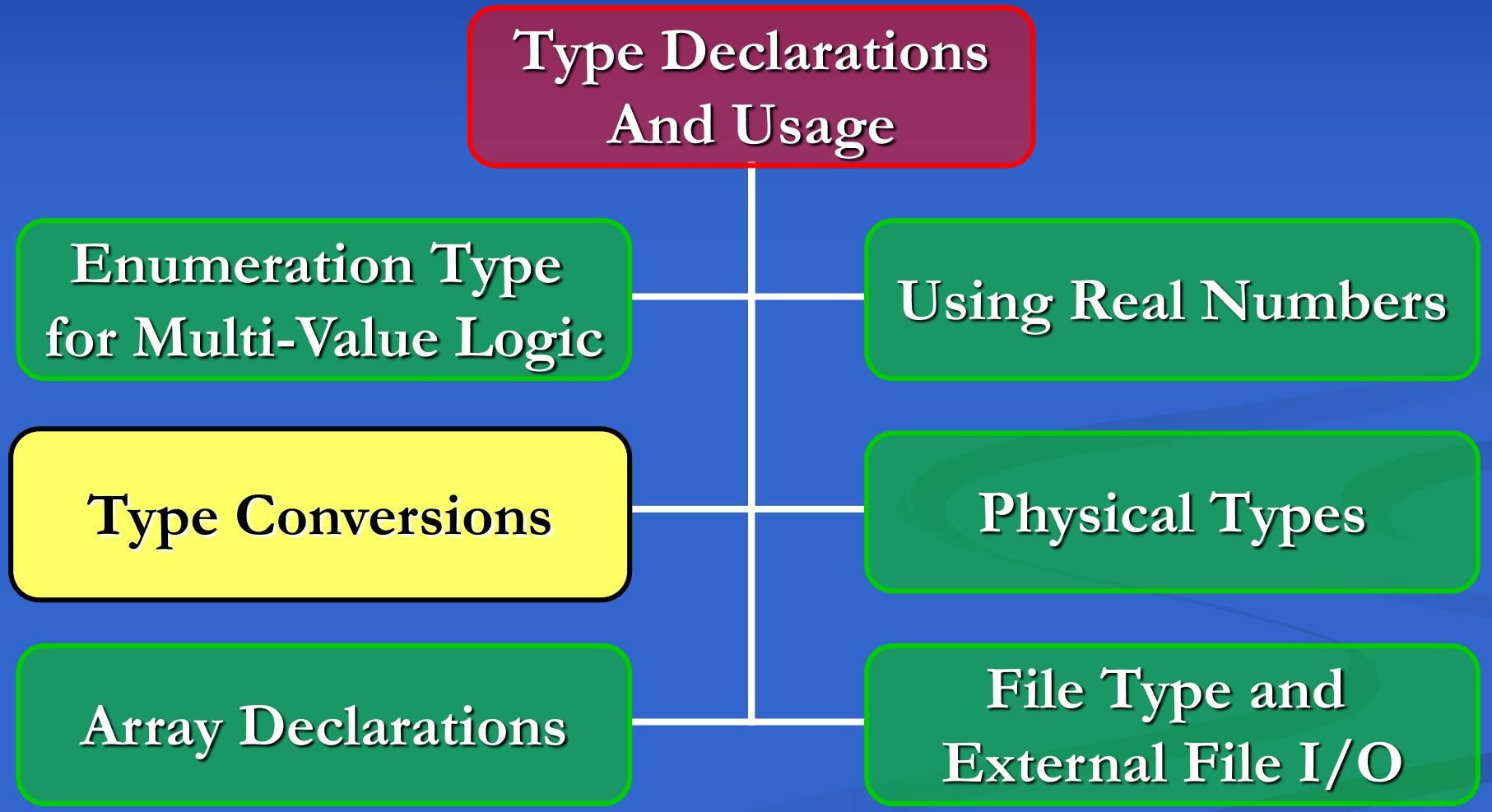
- An Inverter Model with *RC* Timing Parameters

Using Real Numbers

```
. . . . . .
ARCHITECTURE rc_timed OF cmos_not IS
   CONSTANT tplh :
      TIME := INTEGER (rpu * c_load *1.0E15) * 3 FS;
   CONSTANT tphl :
      TIME := INTEGER (rpd * c_load *1.0E15) * 3 FS;
BEGIN
   w <= '1' AFTER tplh WHEN a = '0' ELSE
        '0' AFTER tphl WHEN a = '1' ELSE
        'X' AFTER tplh;
END ARCHITECTURE rc_timed;
```

- An Inverter Model with *RC* Timing Parameters (Continued)

Type Conversions



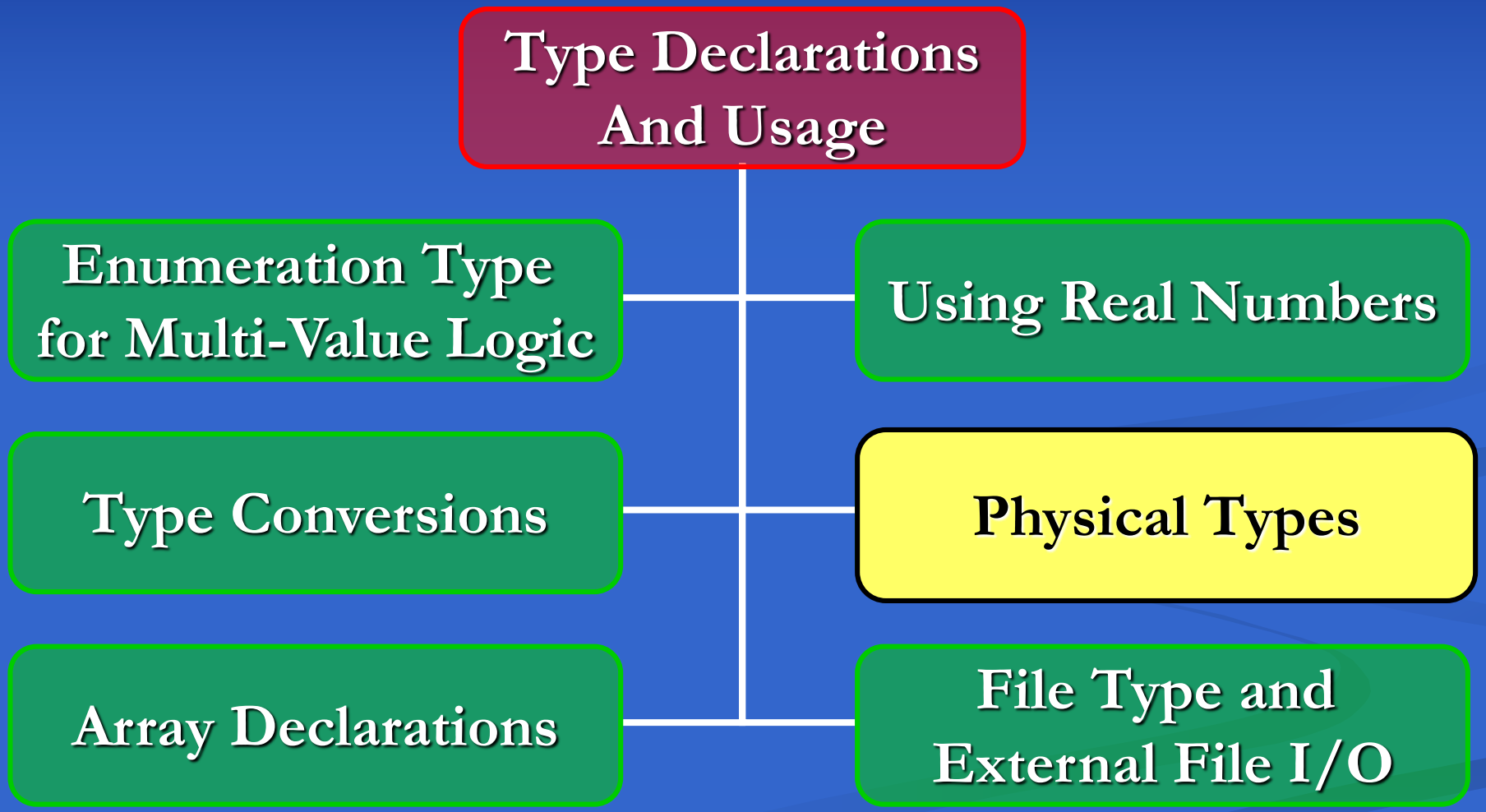
Type Conversions

Explicit Conversion

```
. . . . . .
ARCHITECTURE rc_timed OF cmos_not IS
   CONSTANT tplh :
      TIME := INTEGER (rpu * c_load *1.0E15) * 3 FS;
   CONSTANT tphl :
      TIME := INTEGER (rpd * c_load *1.0E15) * 3 FS;
BEGIN
   w <= '1' AFTER tplh WHEN a = '0' ELSE
        '0' AFTER tphl WHEN a = '1' ELSE
        'X' AFTER tplh;
END ARCHITECTURE rc_timed;
```

- An Inverter Model with *RC* Timing Parameters (Continued)

Physical Types



Physical Types

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

- Type Definition for Defining the C*apacitance* Physical Type

Physical Types

```
TYPE resistance IS RANGE 0 TO INTEGER'HIGH
   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;
```

- Type Definition for Defining the *Resistance* Physical Type

Physical Types

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;
USE utilities.BasicUtilities.ALL;

ENTITY cmos_not IS
    GENERIC (c_load : capacitance := 66 ffr);
    PORT (w : OUT v4l; a : IN v4l);
    CONSTANT rpu : resistance := 3 k_o;
    CONSTANT rpd : resistance := 2.1 k_o;
END ENTITY cmos_not;
. . . . . .
```

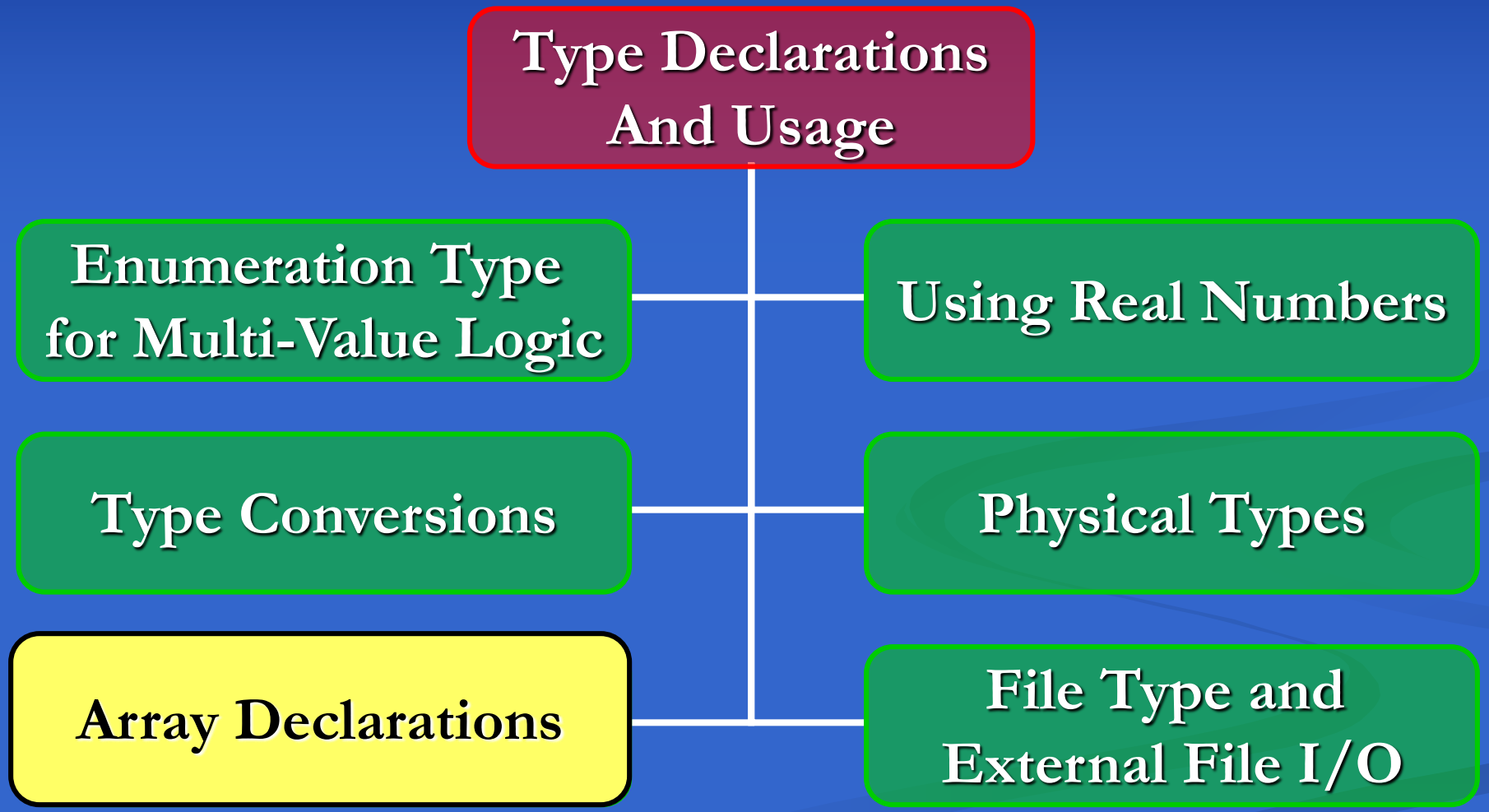
- Using *Resistance* and *Capacitance* Physical Types

Physical Types

```
. . . . . .
ARCHITECTURE rc_timed OF cmos_not 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
   w <= '1' AFTER tplh WHEN a = '0' ELSE
        '0' AFTER tphl WHEN a = '1' ELSE
        'X' AFTER tplh;
END ARCHITECTURE rc_timed;
```

- Using *Resistance* and *Capacitance* Physical Types (Continued)

Array Declarations

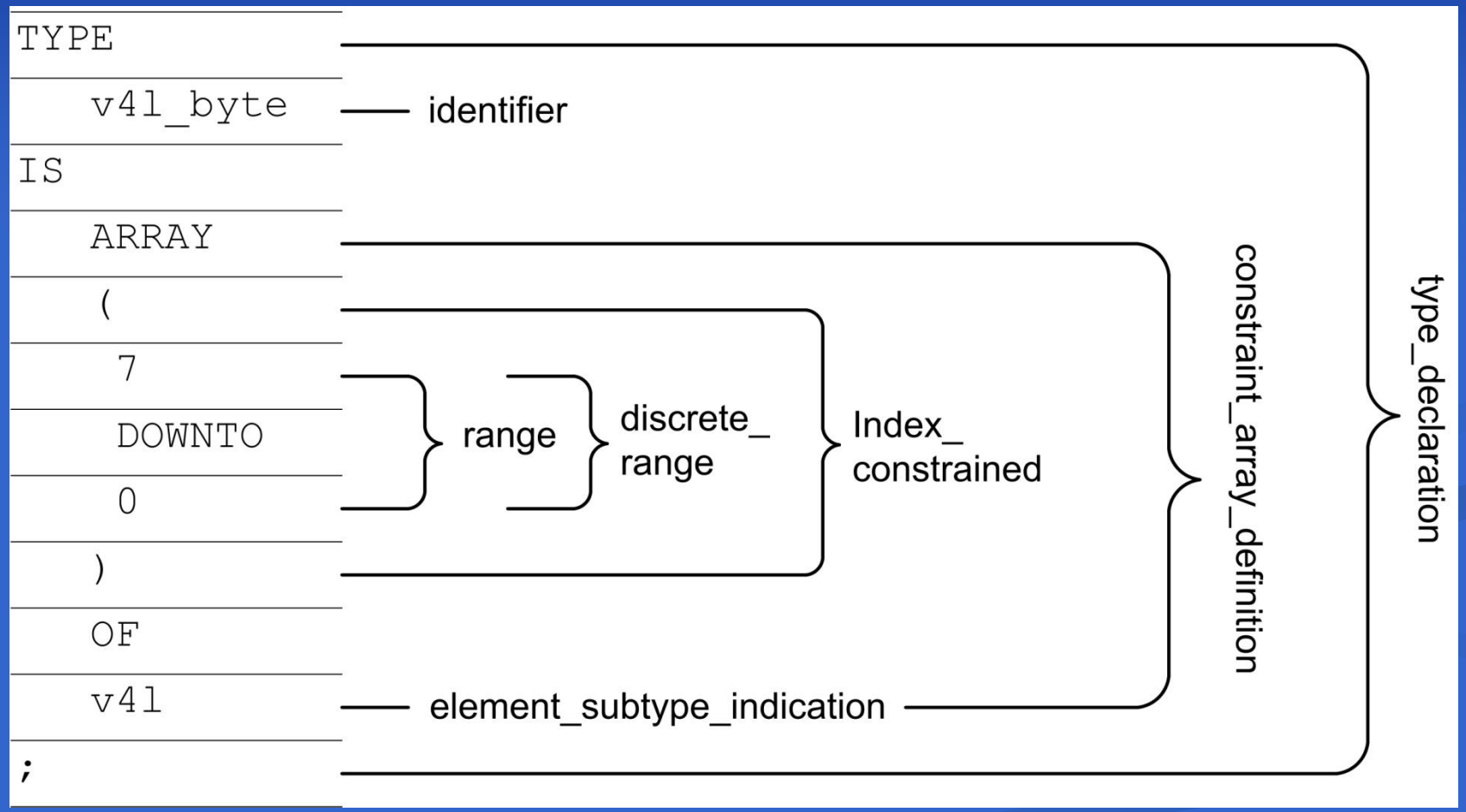


Array Declarations

```
TYPE v41_byte IS ARRAY (7 DOWNTO 0) of v41;
TYPE v41_word IS ARRAY (15 DOWNTO 0) of v41;
TYPE v41_4by8 IS ARRAY (3 DOWNTO 0, 0 TO 7) of v41;
TYPE v41_1kbyte IS ARRAY (0 to 1023) OF v41_byte;
TYPE v41_8cube IS ARRAY (0 TO 7, 0 TO 7, 0 TO 7) of v41;
```

- Declaring Array Types

Array Declarations



- **Syntax Details of an Array Type Declaration**

Array Declarations

```
ARCHITECTURE assign OF array_test IS
    SIGNAL s : v41;
    SIGNAL s_byte : v41_byte;
    SIGNAL s_word : v41_word;
    SIGNAL s_4by8 : v41_4by8;
    SIGNAL s_1kbyte : v41_1kbyte;
    SIGNAL s_8cube : v41_8cube;
BEGIN
    . . . . . .
    . . . . . .
    . . . . . .
END ARCHITECTURE assign;
```

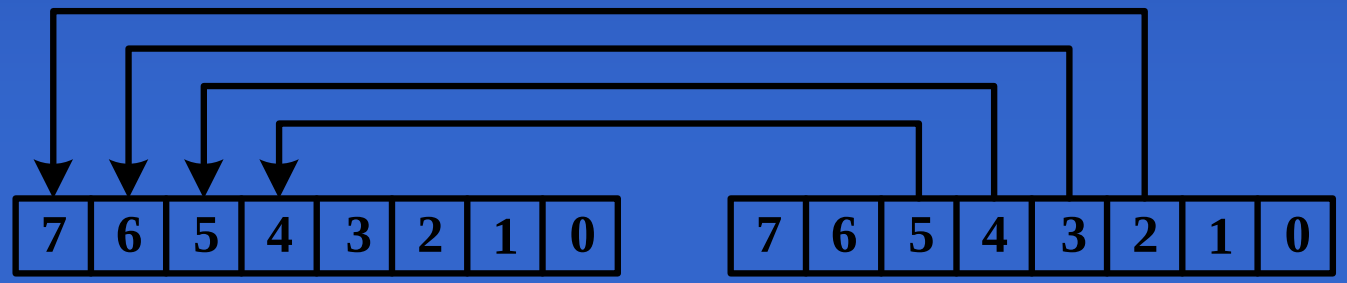
- Signal Assignments Based on Signal Declarations

Array Declarations

```
BEGIN
   SA1: s_byte <= v4l_byte ( s_word (11 DOWNTO 4) );
   SA2: s <= s_4by8 (0, 7);
   SA3: s_byte <= s_1kbyte (27);
   SA4: s <= s_1kbyte (23)(3);
   SA5: s_byte <= s_byte (0) & s_byte (7 DOWNTO 1);
   SA6: s_byte (7 DOWNTO 4) <=
            s_byte(2) & s_byte(3) & s_byte(4) & s_byte(5);
   SA7: s_byte (7 DOWNTO 4) <=
            (s_byte(2), s_byte(3), s_byte(4), s_byte(5));
   SA8:(s_byte(0), s_byte(1), s_byte(2), s_byte(3))
            <= s_byte (5 DOWNTO 2);
END ARCHITECTURE assign;
```

- Signal Assignments Based on Signal Declarations (Continued)

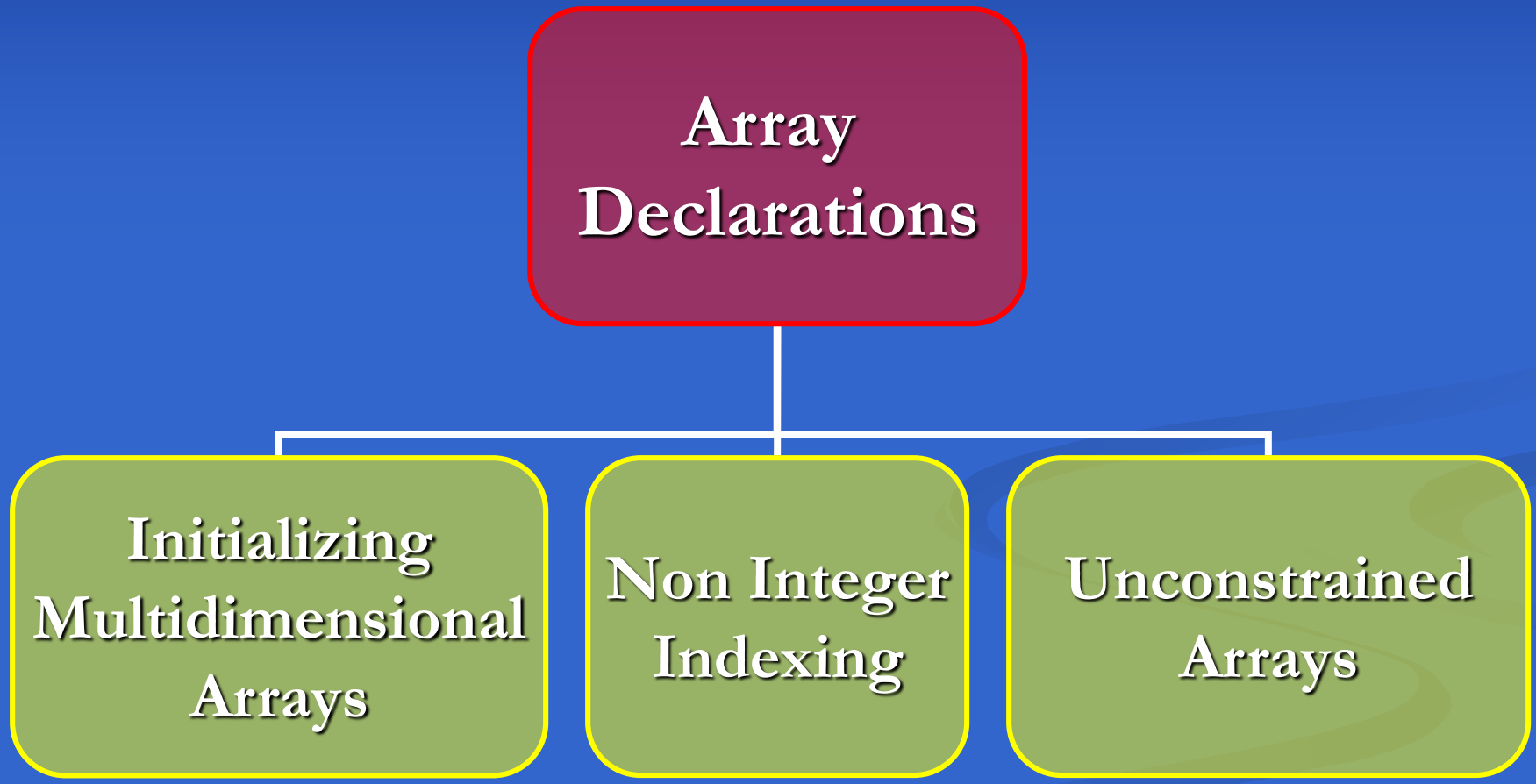
Array Declarations



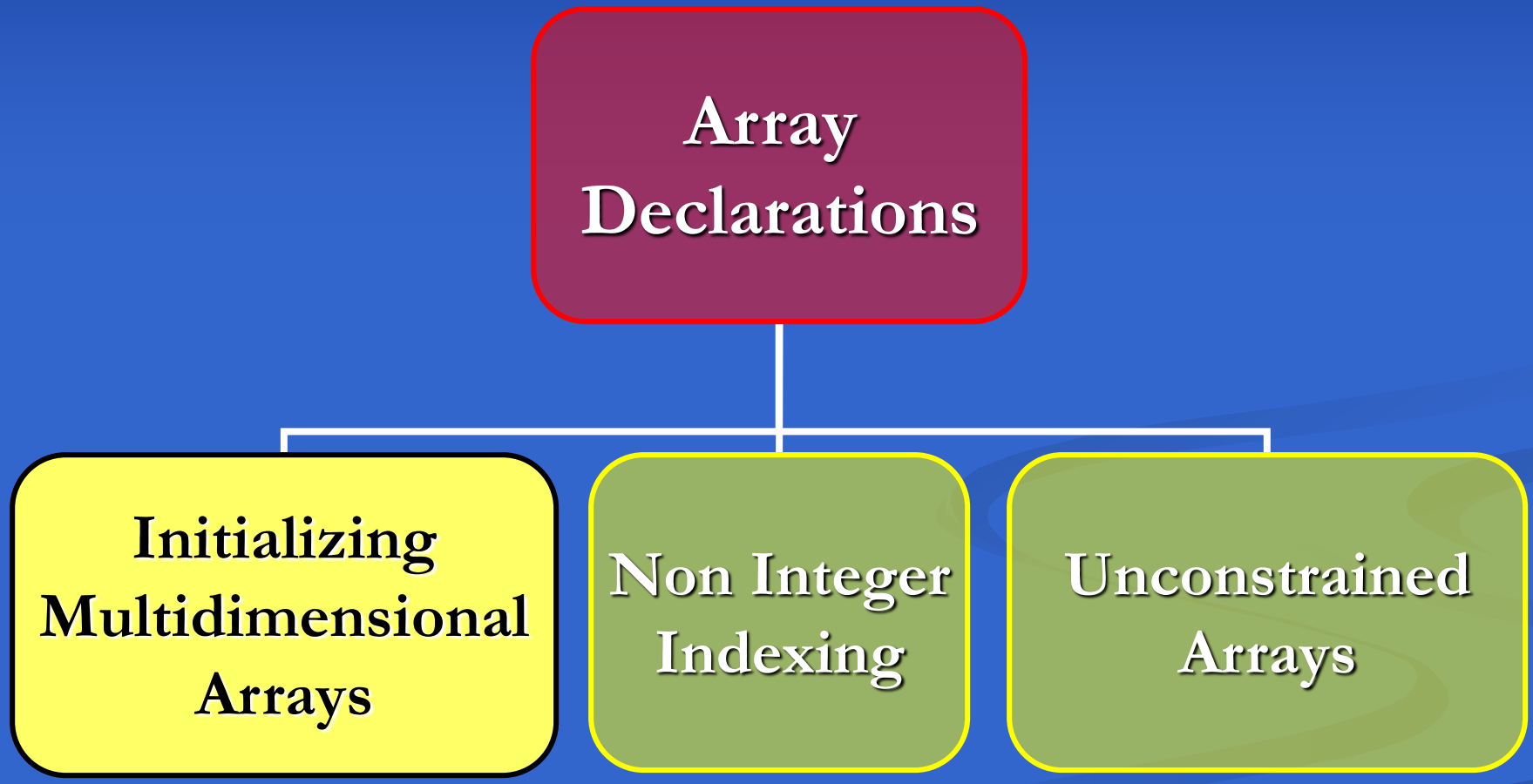

```
SA7: s_byte (7 DOWNTO 4) <=
          (s_byte(2), s_byte(3), s_byte(4), s_byte(5));
```

- Reversing Bits of *s_byte*

Array Declarations



Initializing Multidimensional Arrays

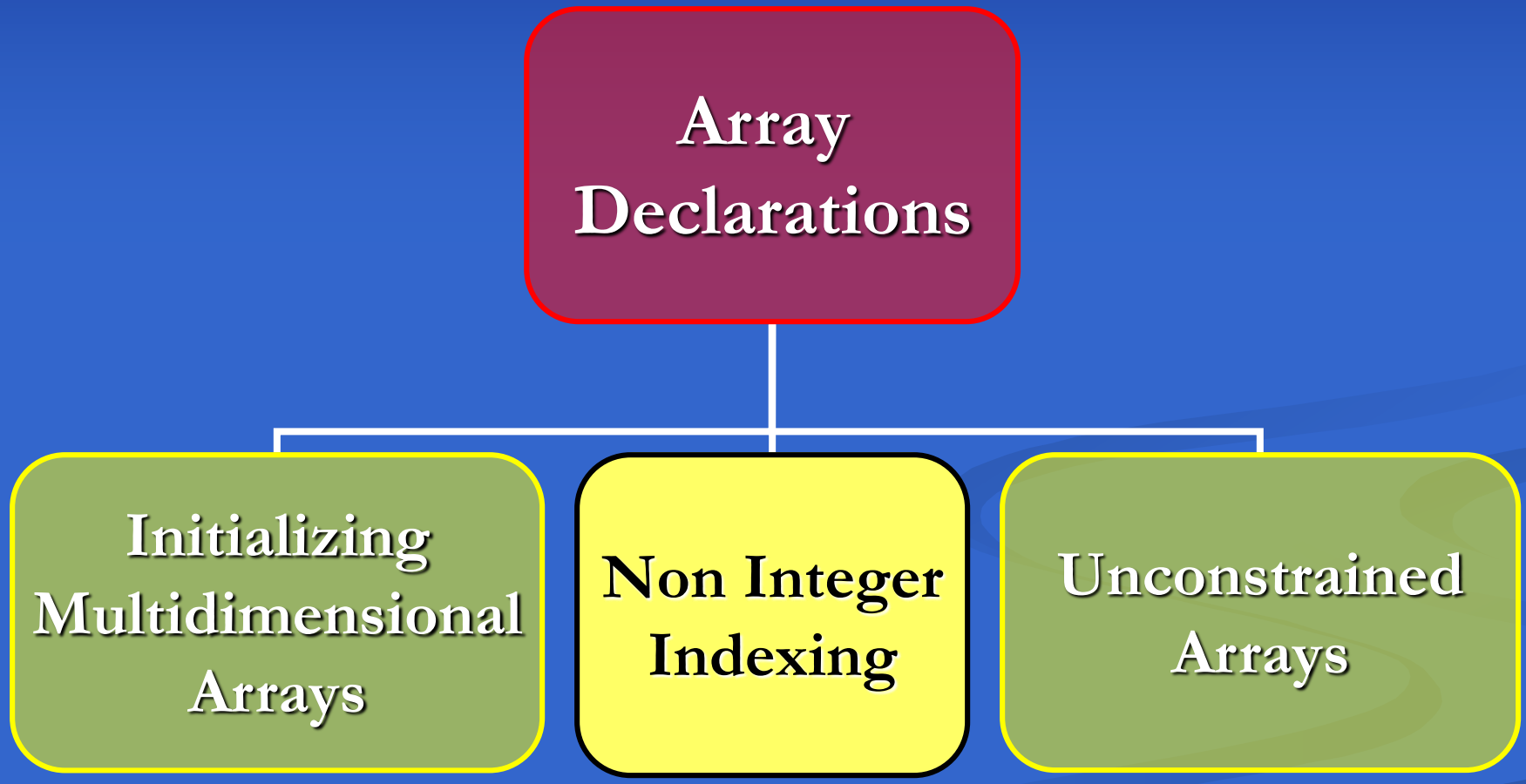


Initializing Multidimensional Arrays

```
SIGNAL s_4by8 : v4l_4by8 :=
     (
          ( '0', '0', '1', '1', 'Z', 'Z', 'X', 'X' ),
          ( 'X', 'X', '0', '0', '1', '1', 'Z', 'Z' ),
          ( 'Z', 'Z', 'X', 'X', '0', '0', '1', '1' ),
          ( '1', '1', 'Z', 'Z', 'X', 'X', '0', '0' )
      );
SIGNAL s_4by8 : v4l_4by8 := (OTHERS => "11000000");
SIGNAL s_4by8 : v4l_4by8 := (OTHERS => (OTHERS =>
                              'Z'));
SIGNAL s_4by8 : v4l_4by8 :=(OTHERS => (0 TO 1 =>
                               '1', OTHERS =>'0'));
```

- Initializing a Two Dimensional Array

Non Integer Indexing



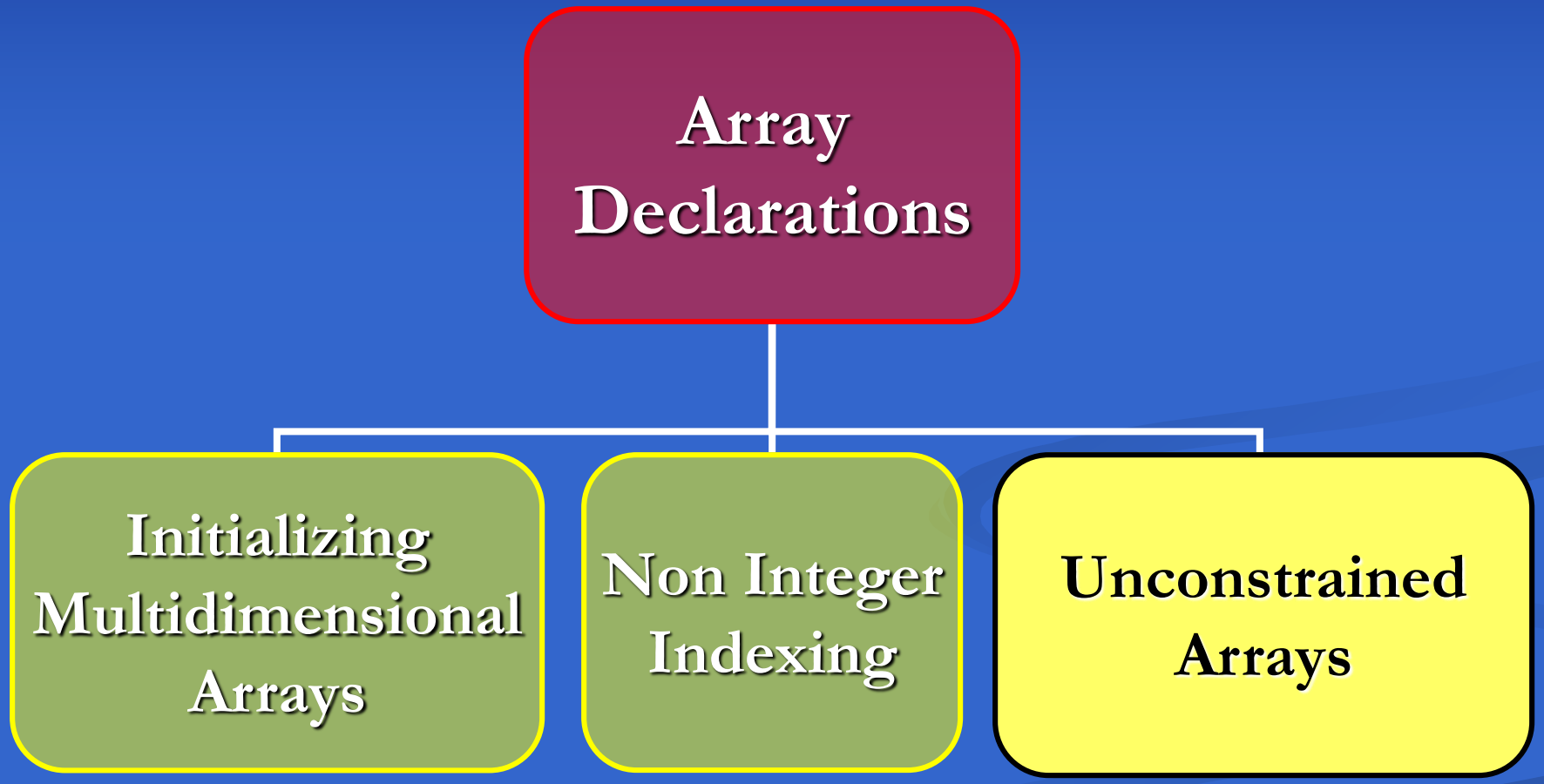
Non Integer Indexing

```
TYPE v4l_2d IS ARRAY (v4l, v4l) OF v4l;
```

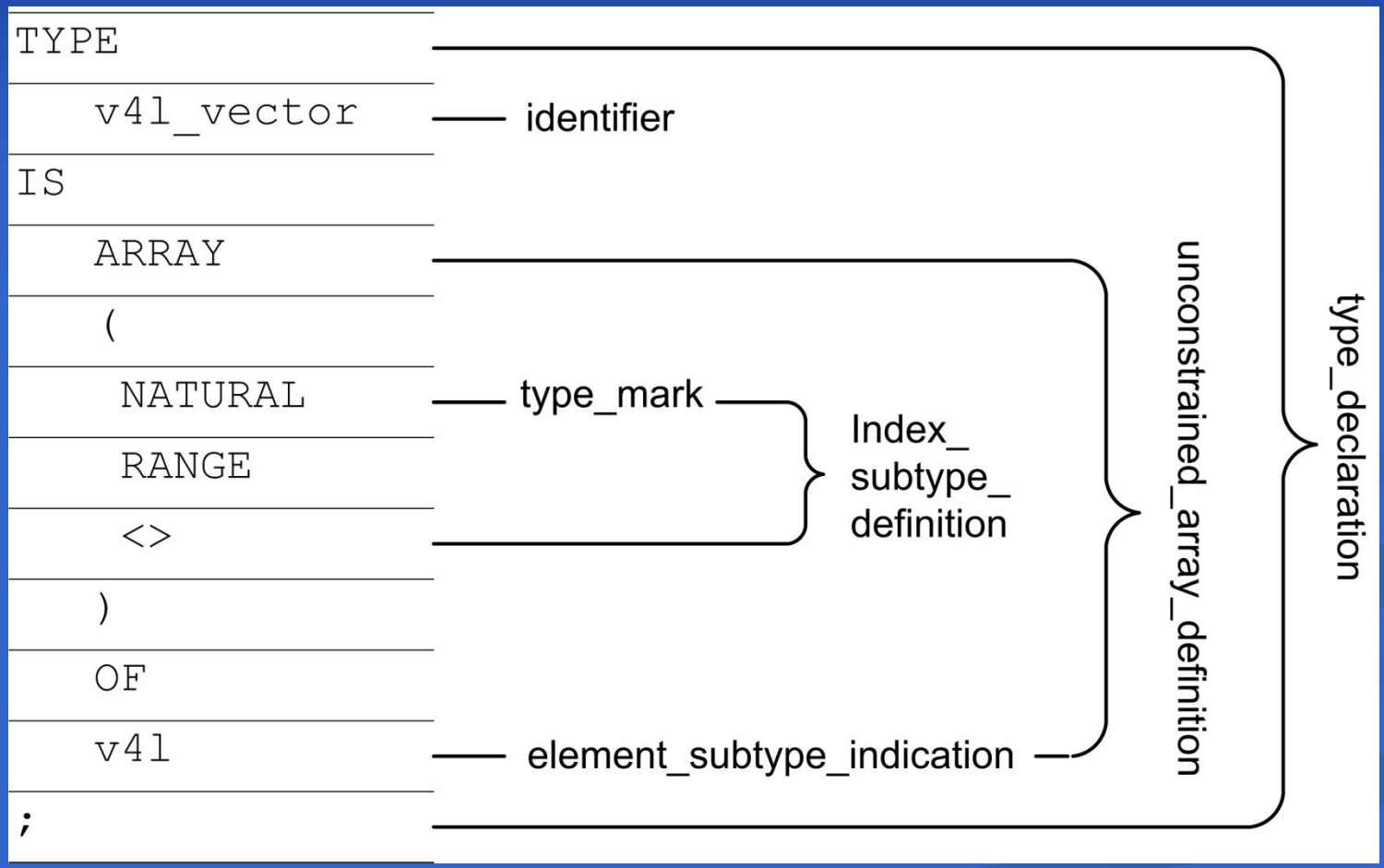
```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;
USE utilities.BasicUtilities.ALL;
ARCHITECTURE tabular OF vlog_nand2 IS
   CONSTANT v4l_nand2_table : v4l_2d := (
       --  X    0    1    Z
         ('X','1','X','X'), -- X
         ('1','1','1','1'), -- 0
         ('X','1','0','X'), -- 1
         ('X','1','X','X'));-- Z
BEGIN
   w <= v4l_nand2_table (a, b) AFTER (tplh +tphl)/2;
END ARCHITECTURE tabular;
```

- Enumeration Type for Discrete Range of a Two-Dimensional Array

Unconstrained Arrays



Unconstrained Arrays



- **Syntax Details of an Unconstrained Array Declaration**

Unconstrained Arrays

```
PROCEDURE onehot_data
   SIGNAL target : OUT v41_vector;
   CONSTANT ti : TIME; CONSTANT n : INTEGER)
IS
   VARIABLE data : v41_vector (target'RANGE);
   VARIABLE i : INTEGER := 0;
BEGIN
   data (0) := '1';
   WHILE i < n LOOP
      data := data(data'RIGHT) & data(data'LEFT DOWNTO 1);
      target <= TRANSPORT data AFTER ti * i;
      i := i + 1;
   END LOOP;
END PROCEDURE onehot_data;
```

- A Generic Version of the *onehot_data* Procedure

Unconstrained Arrays

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;
ENTITY vlog_ram IS
   PORT (address : IN v4l_vector;
         datain : IN v4l_vector; dataout : OUT v4l_vector;
         cs, rwbar : IN v4l; opr : IN BOOLEAN);
END ENTITY vlog_ram;
--
. . . . . .
. . . . . .
```

- A Generic Memory Model

Unconstrained Arrays

```
ARCHITECTURE behavioral OF vlog_ram IS
   TYPE mem IS ARRAY
        (NATURAL RANGE <>, NATURAL RANGE <>) of v41;
BEGIN
   PROCESS
      CONSTANT memsize : INTEGER := 2**address'LENGTH;
      VARIABLE memory : mem (0 TO memsize-1,datain'RANGE);
      BEGIN
         id: IF opr'EVENT THEN
             IF opr=TRUE THEN
                init_mem (memory, "memdata.dat");
             ELSE
                dump_mem (memory, "memdump.dat");
             END IF;
         END IF;
```

- A Generic Memory Model (Continued)

Unconstrained Arrays

```
        . . . . . . . . . . . . . .
        wr: IF cs = '1' THEN
              IF rwbar = '0' THEN          -- Writing
                 FOR i IN dataout'RANGE LOOP
                    memory (int(address), i) := datain (i);
                 END LOOP;
              ELSE                         -- Reading
                 FOR i IN datain'RANGE LOOP
                    dataout (i) <= memory (int(address), i);
                 END LOOP;
              END IF;
        END IF;
        WAIT ON cs, rwbar, address, datain, opr;
    END PROCESS;
END ARCHITECTURE behavioral;
```

- A Generic Memory Model (Continued)

Unconstrained Arrays

```
FUNCTION int (invec : v41_vector) RETURN INTEGER IS
   VARIABLE tmp : INTEGER := 0;
BEGIN
   FOR i IN invec'LENGTH - 1 DOWNTO 0 LOOP
      IF invec (i) = '1' THEN
         tmp := tmp + 2**i;
      ELSIF invec (i) = '0' THEN
         tmp := tmp;
      ELSE
         tmp := 0;
      END IF;
   END LOOP;
   RETURN tmp;
END FUNCTION int;
```

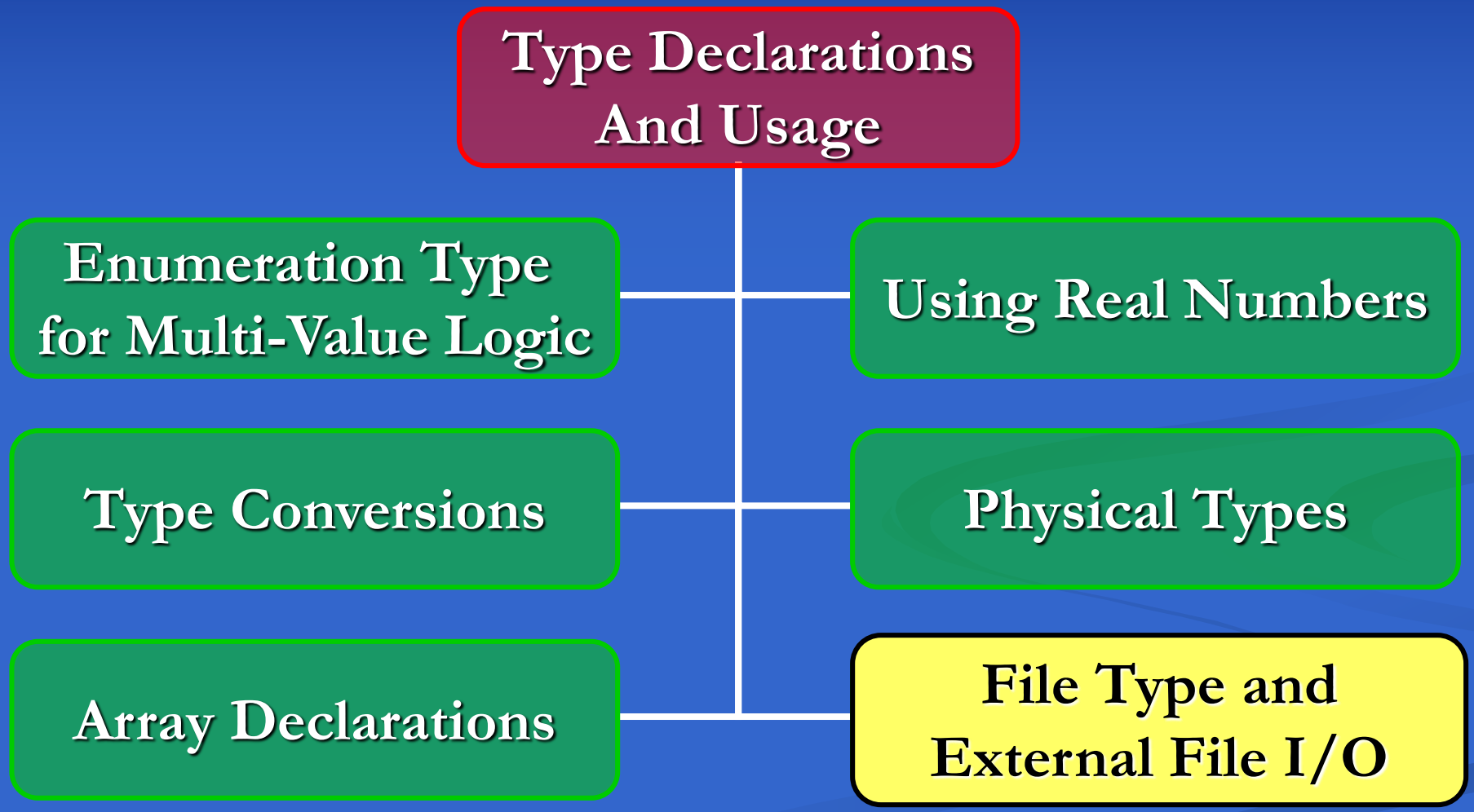
- Unconstrained Function *int*

Unconstrained Arrays

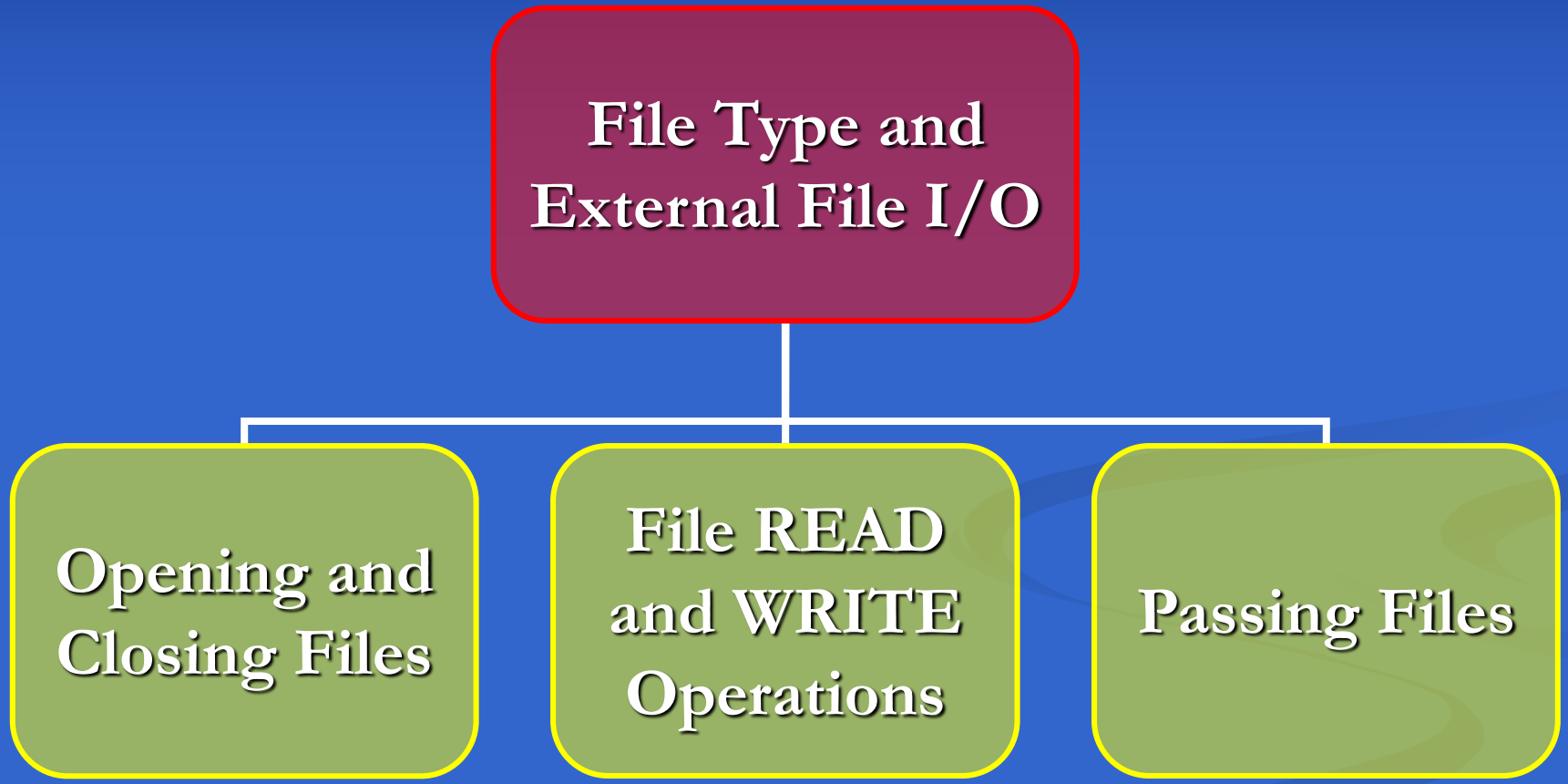
```
ENTITY vlog_ram_tester IS END ENTITY vlog_ram_tester;
ARCHITECTURE timed OF vlog_ram_tester IS
   SIGNAL ramin, ramout : v41_vector (7 DOWNTO 0);
   SIGNAL addr : v41_vector (5 DOWNTO 0);
   SIGNAL cs, rwbar : v41;
   SIGNAL operate : BOOLEAN;
BEGIN
   UU1: ENTITY WORK.vlog_ram (behavioral)
   PORT MAP (addr, ramin, ramout, cs, rwbar, operate);
   operate <= TRUE AFTER 5 NS, FALSE AFTER 400 NS;
   cs <= '0', '1' AFTER 15 NS, '0' AFTER 337 NS;
   rwbar <= '1', '1' AFTER 190 NS;
   addr <= "101100" AFTER 020 NS, "101110" AFTER 040 NS
   ramin <= "11110001" AFTER 010 NS, . . .
END ARCHITECTURE timed;
```

- Testbench Instantiating an Unconstrained Memory

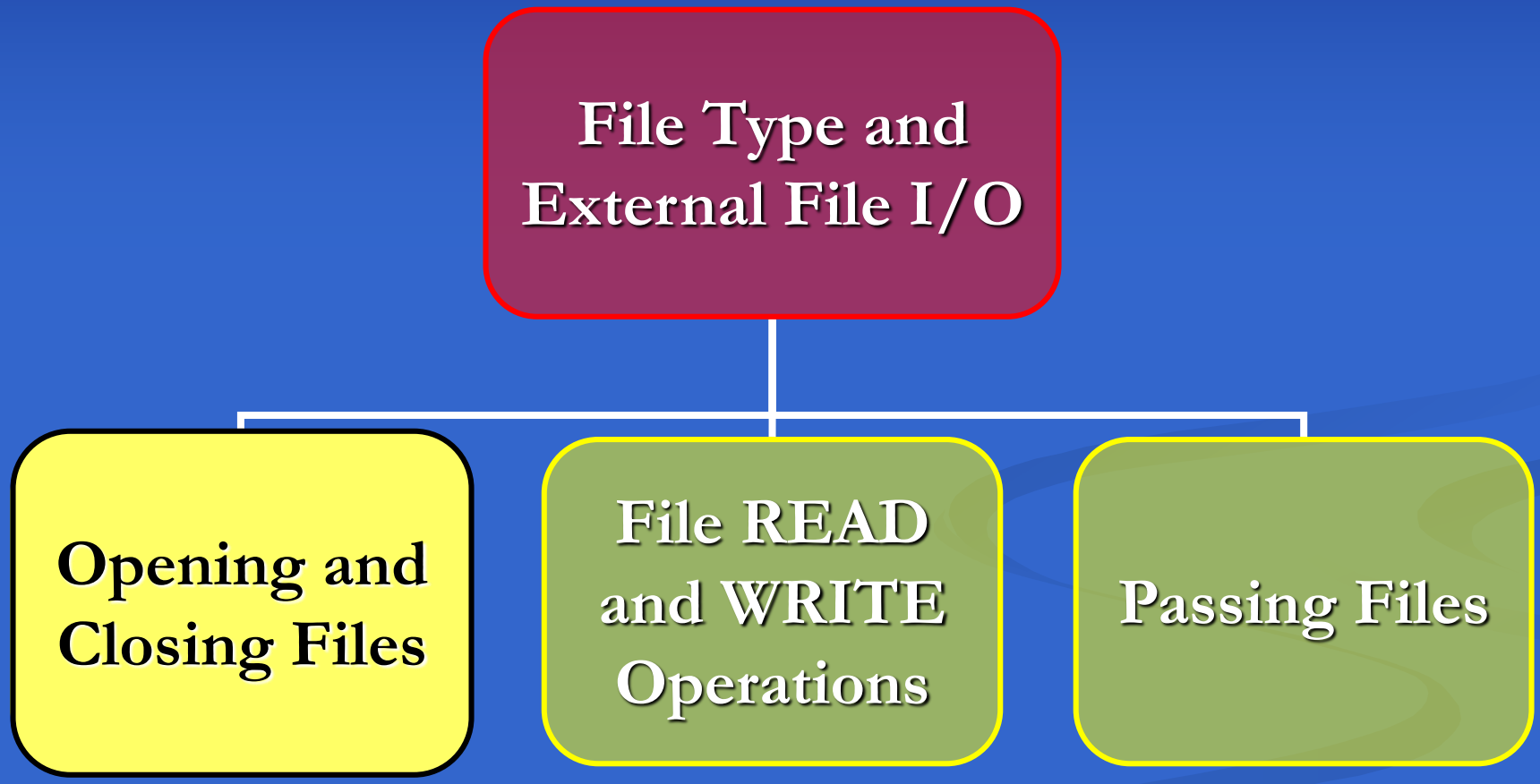
File Type and External File I/O



File Type and External File I/O



Opening and Closing Files



Opening and Closing Files

```
TYPE logic_data IS FILE OF CHARACTER;

FILE input_logic_value_file1 :
                              logic_data;
FILE input_logic_value_file2 :
                              logic_data IS “input.dat”;
FILE input_logic_value_file3 :
                 logic_data OPEN READ_MODE IS “input.dat”;

FILE output_logic_value_file1 :
                               logic_data;
FILE output_logic_value_file2 :
                logic_data OPEN WRITE_MODE IS “input.dat”;
```

Opening and Closing Files

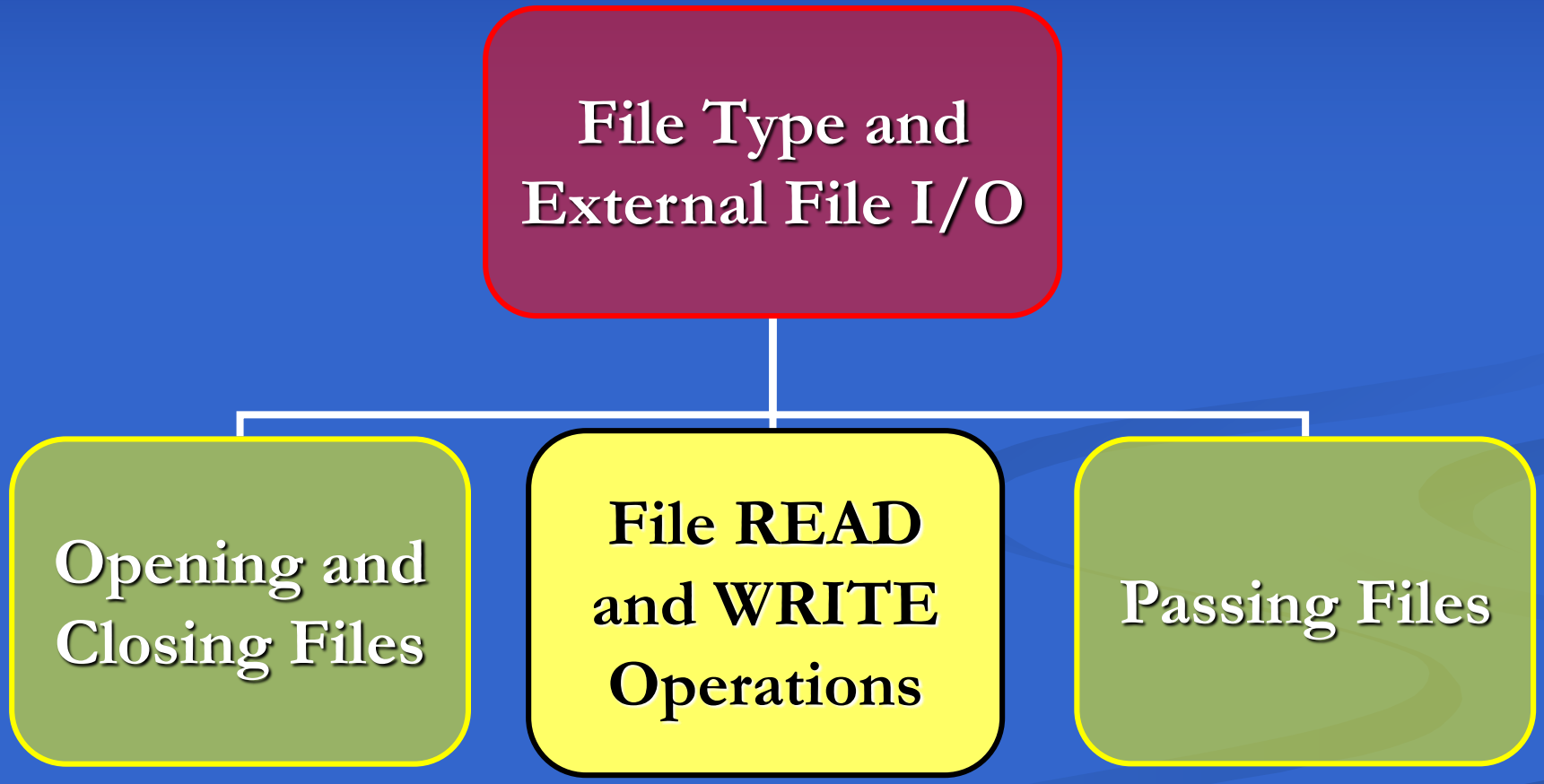
```
FILE_OPEN (input_logic_value_file1,
           "input.dat", READ_MODE);
FILE_OPEN (output_logic_value_file1,
           "output.dat", WRITE_MODE);
```

FILE_OPEN_STATUS type may be included as the first parameter of the FILE_OPEN statement:

- OPEN_OK
- STATUS_ERROR
- NAME_ERROR
- MODE_ERROR

```
FILE_CLOSE (input_logic_value_file1);
FILE_CLOSE (output_logic_value_file1);
```

File READ and WRITE Operations



File READ and WRITE Operations

```
TYPE v4lfiletype IS FILE OF CHARACTER;
PROCEDURE init_mem
    (VARIABLE memory: OUT mem;
     CONSTANT datafile: STRING)
IS
    FILE v4ldata : v4lfiletype;
    VARIABLE v4lvalue : v4l;
    VARIABLE char : CHARACTER;
BEGIN
    . . . . . .
    . . . . . .
END PROCEDURE init_mem;
```

- Reading an External File

File READ and WRITE Operations

```
  . . . . . . . . . . . . . .
BEGIN
    FILE_OPEN (v4ldata, datafile, READ_MODE);
    FOR i IN memory'RANGE(1) LOOP
        FOR j IN memory'REVERSE_RANGE(2) LOOP
            READ (v4ldata, char);
            v4lvalue := chartov4l (char);
            memory (i,j) := chartov4l (char);
        END LOOP;
        READ (v4ldata, char);
        READ (v4ldata, char); -- read cr lf
    END LOOP;
END PROCEDURE init_mem;
```

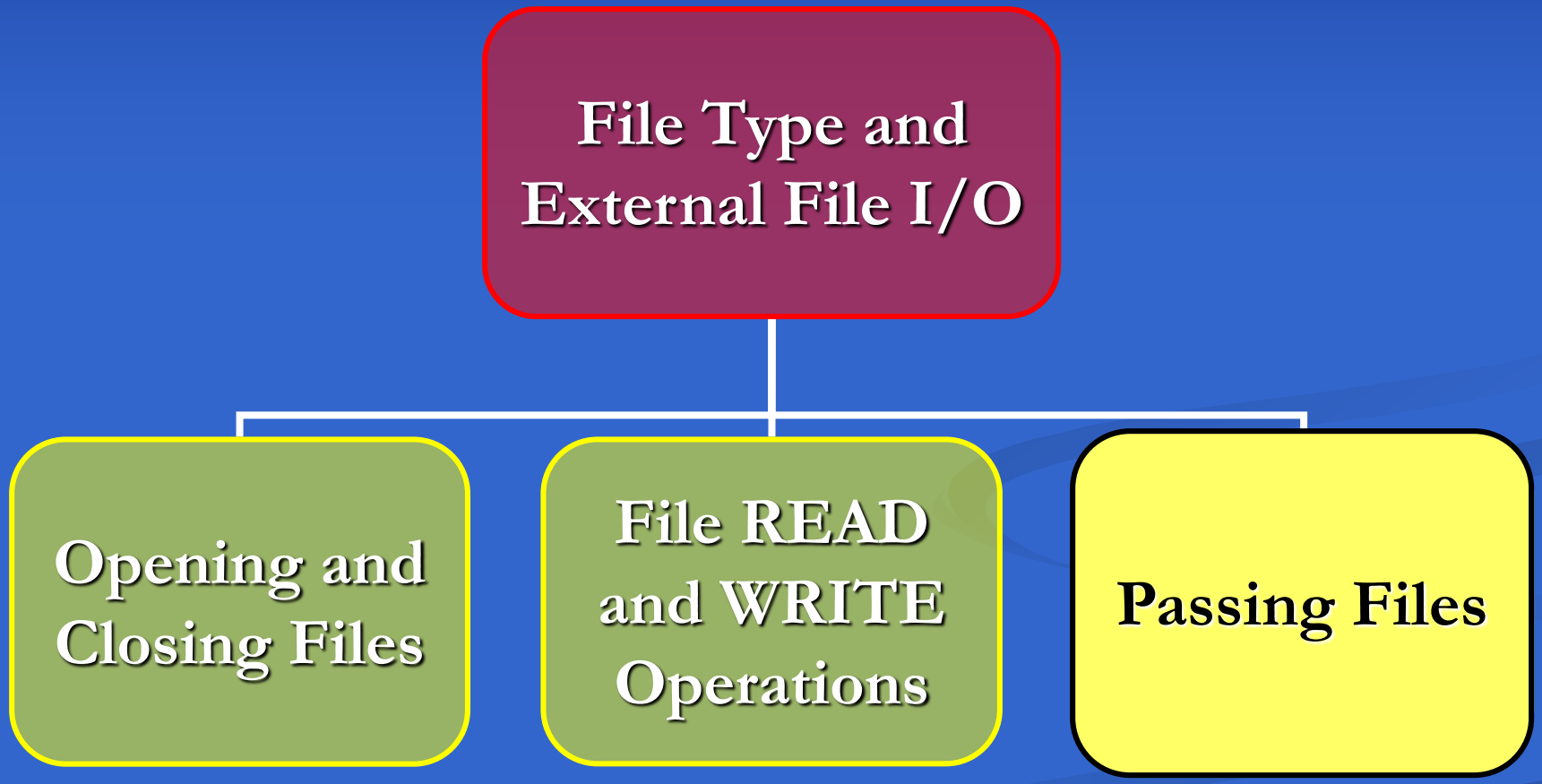
- Reading an External File (Continued)

File READ and WRITE Operations

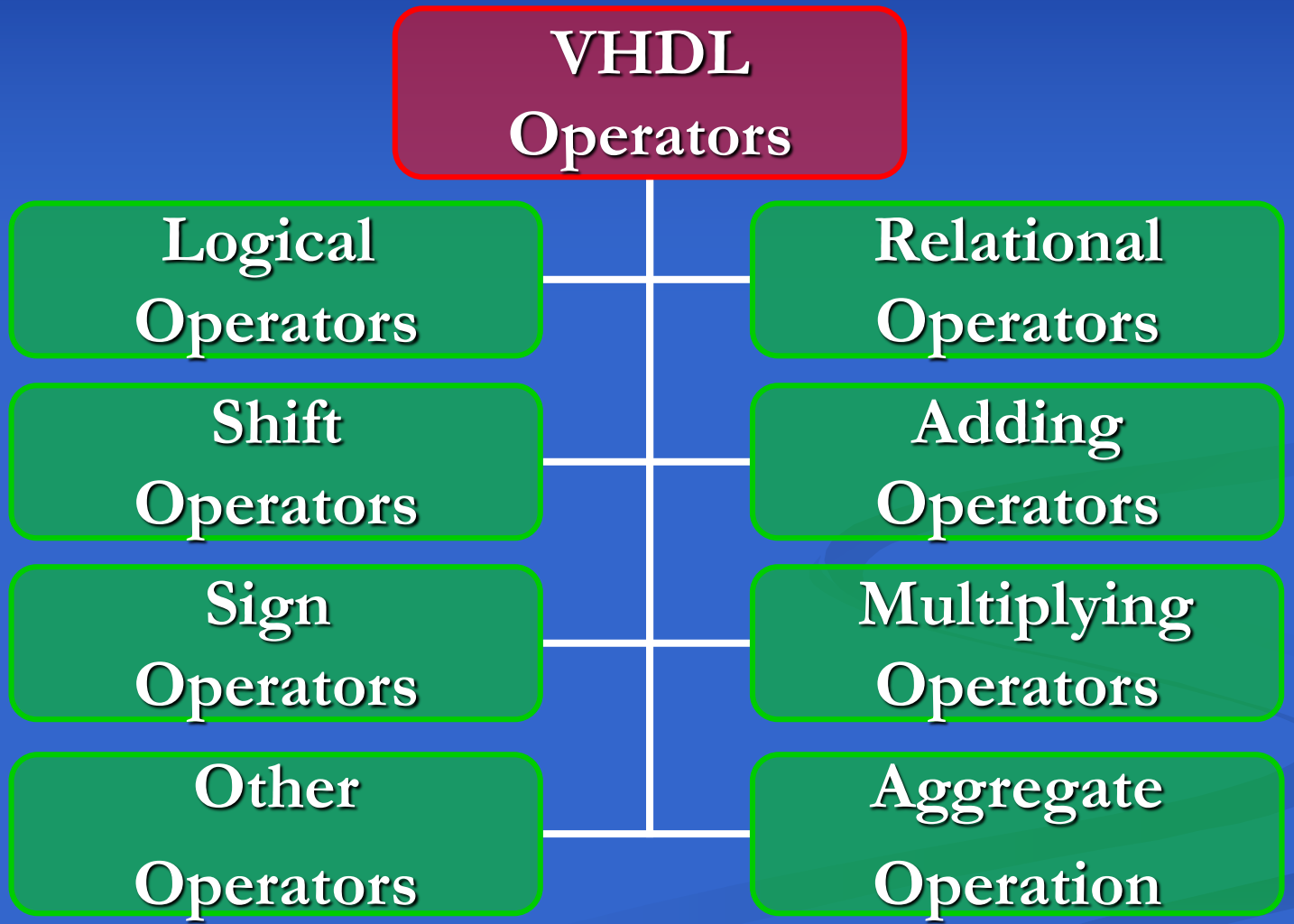
```
PROCEDURE dump_mem
    (VARIABLE memory: IN mem; CONSTANT datafile: STRING)
IS
     FILE v4ldata : v4lfiletype;
     VARIABLE v4lvalue : v4l; VARIABLE char : CHARACTER;
BEGIN
     FILE_OPEN (v4ldata, datafile, WRITE_MODE);
     FOR i IN memory'RANGE(1) LOOP
        FOR j IN memory'REVERSE_RANGE(2) LOOP
           v4lvalue := memory (i, j);
           WRITE (v4ldata, v4ltochar (v4lvalue));
        END LOOP;
        WRITE (v4ldata, cr);
      END LOOP;
END PROCEDURE dump_mem;
```

- Writing into an External File

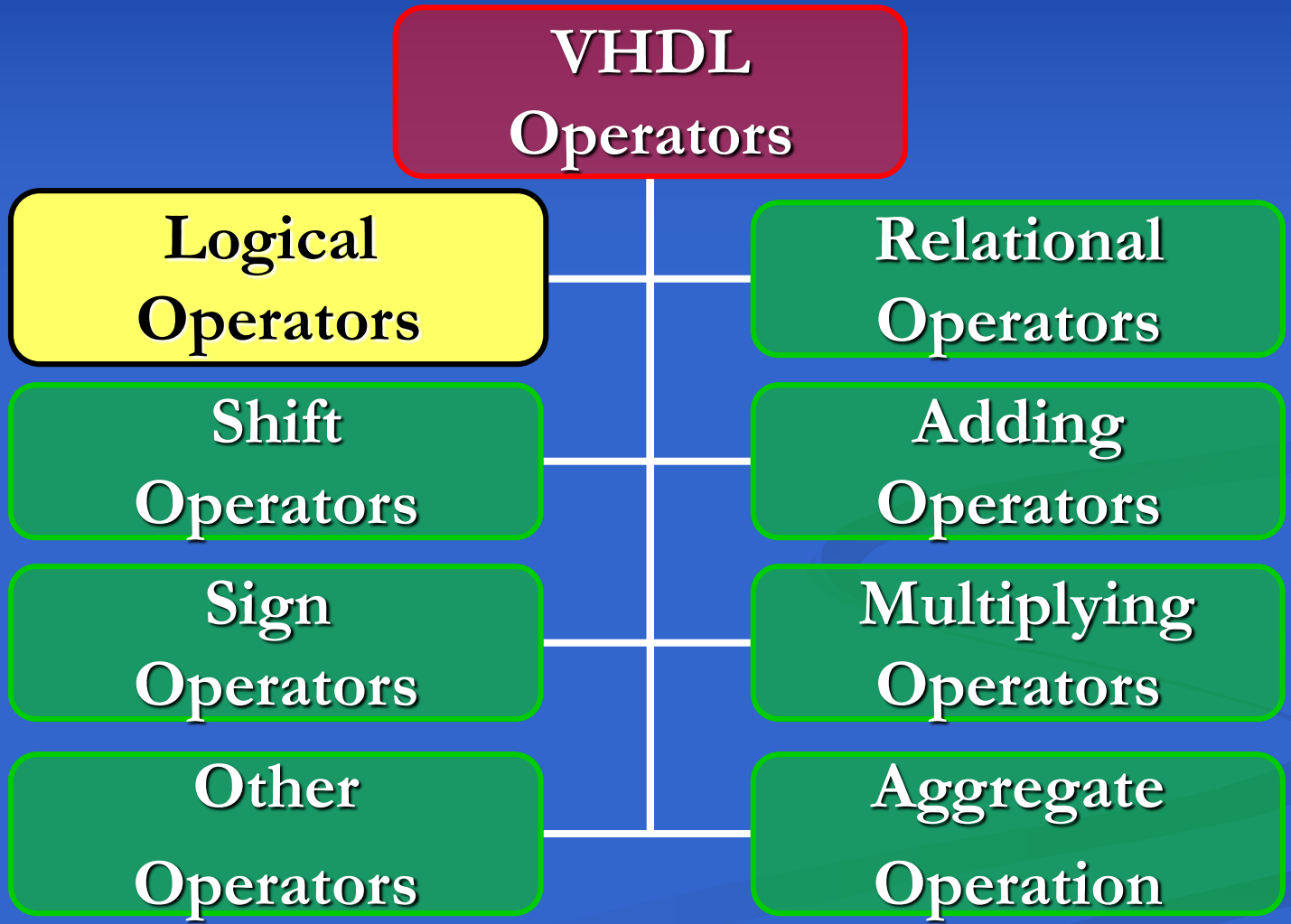
Passing Files



VHDL Operators



Logical Operators



Logical Operators

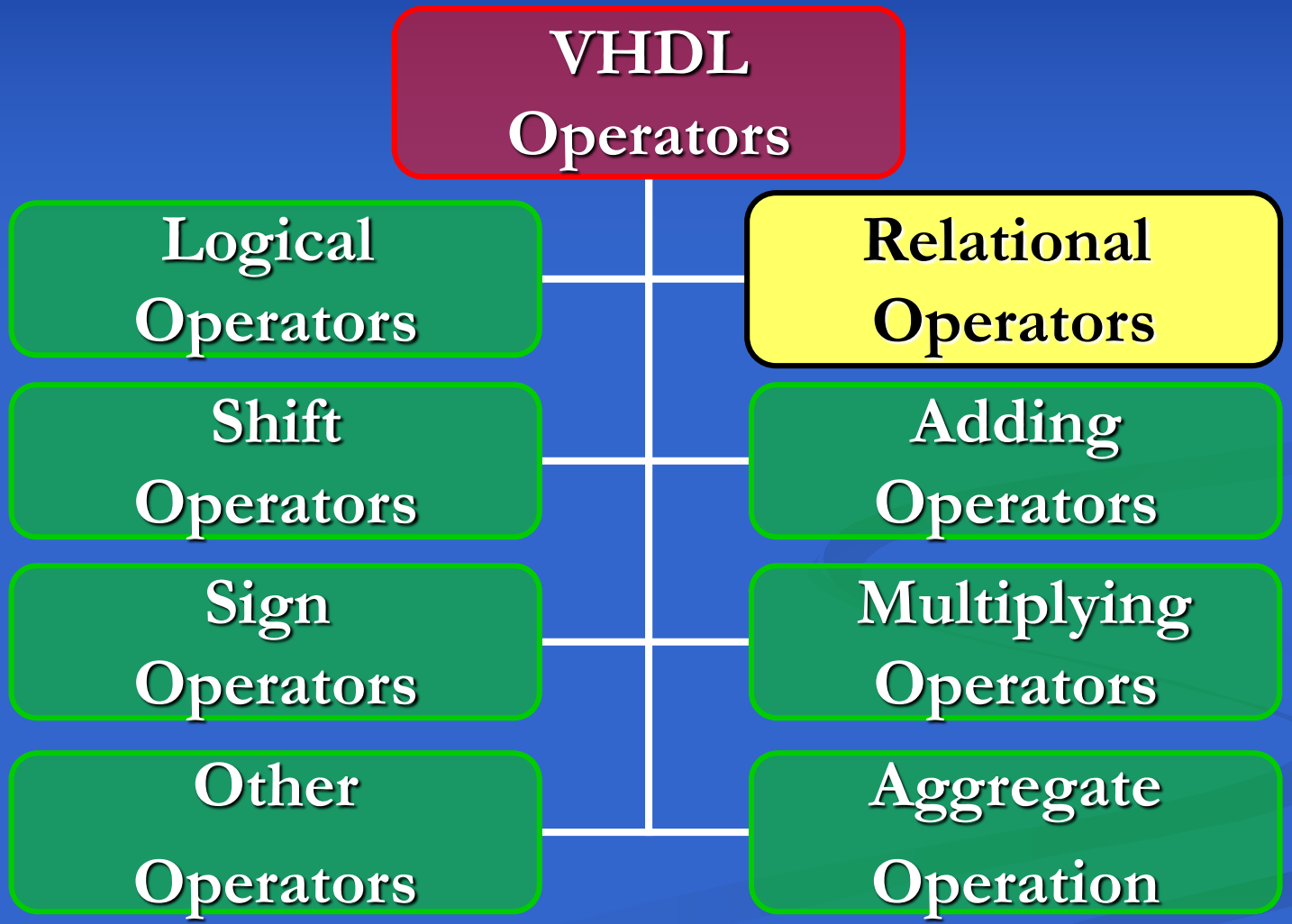
- Logical Operators:
  - AND, OR, NAND, NOR, XOR, XNOR, and NOT
  - Example : `x <= a XNOR b;`

- Logical operators perform on predefined types **BIT**, **BOOLEAN** and **BIT_VECTOR**.

- Strings representing operator symbols can be used as function names for performing the same function as the operator they are representing:

  - Example:
```
x <= "XOR" (a, b);
x_vector <= "AND" (a_vector, b_vector);
```

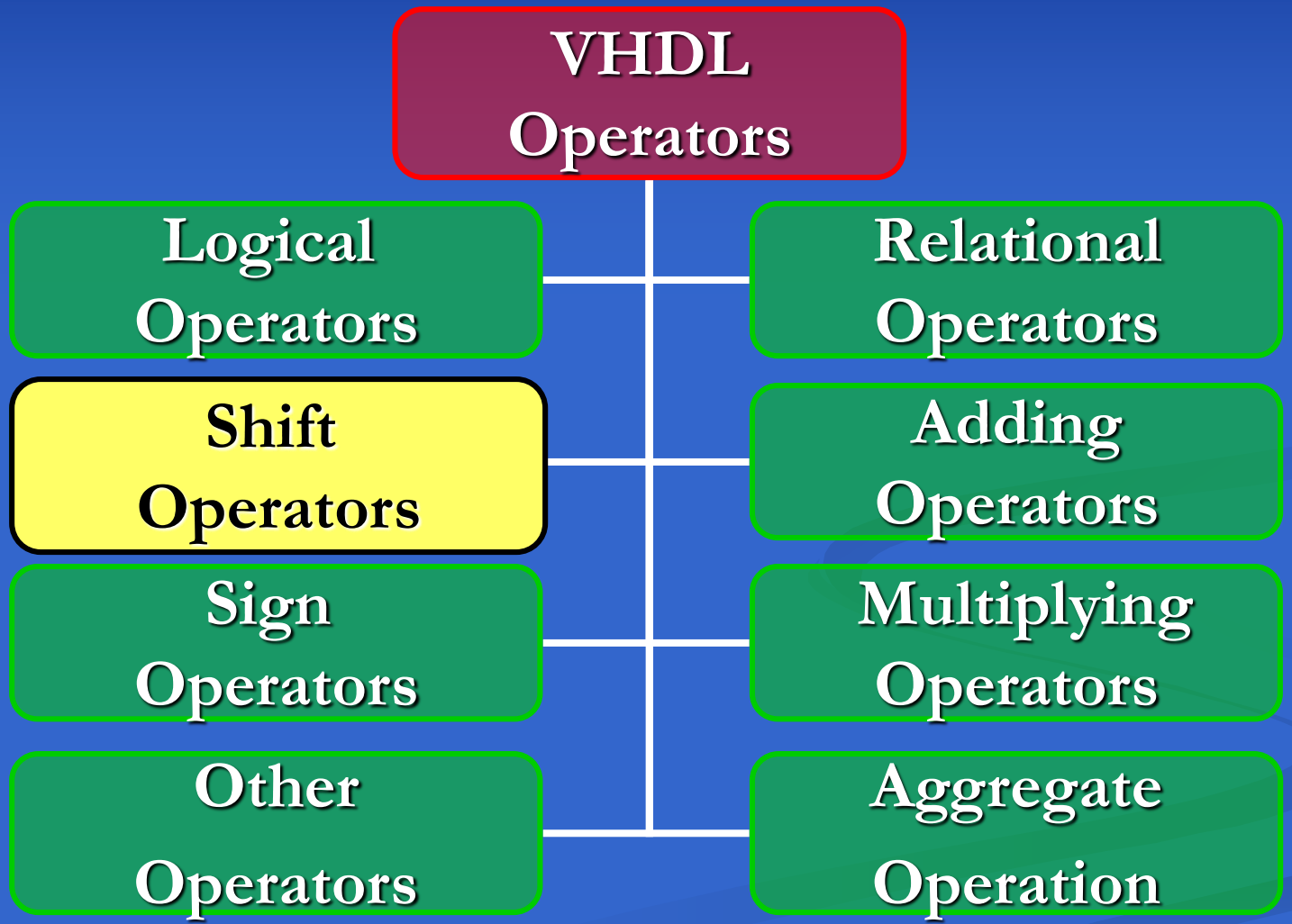
Relational Operators



Relational Operators

- Relational operators operate on operands of the same type and return a BOOLEAN TRUE or FALSE value.
- Operators in this group are
  - =, /=, <, <=, >, and >= with equal, not equal, less than, less than or equal, greater than, and greater than or equal functionalities.
- The = and /= operators operate on operands of any type. The other relational operators perform their normal functions when used with scalar operands.
- When array operands are used with these operators (<, <=, >, and >=), they perform ordering operations and return TRUE or FALSE based on values of array elements starting from the left.

Shift Operators



Shift Operators

	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

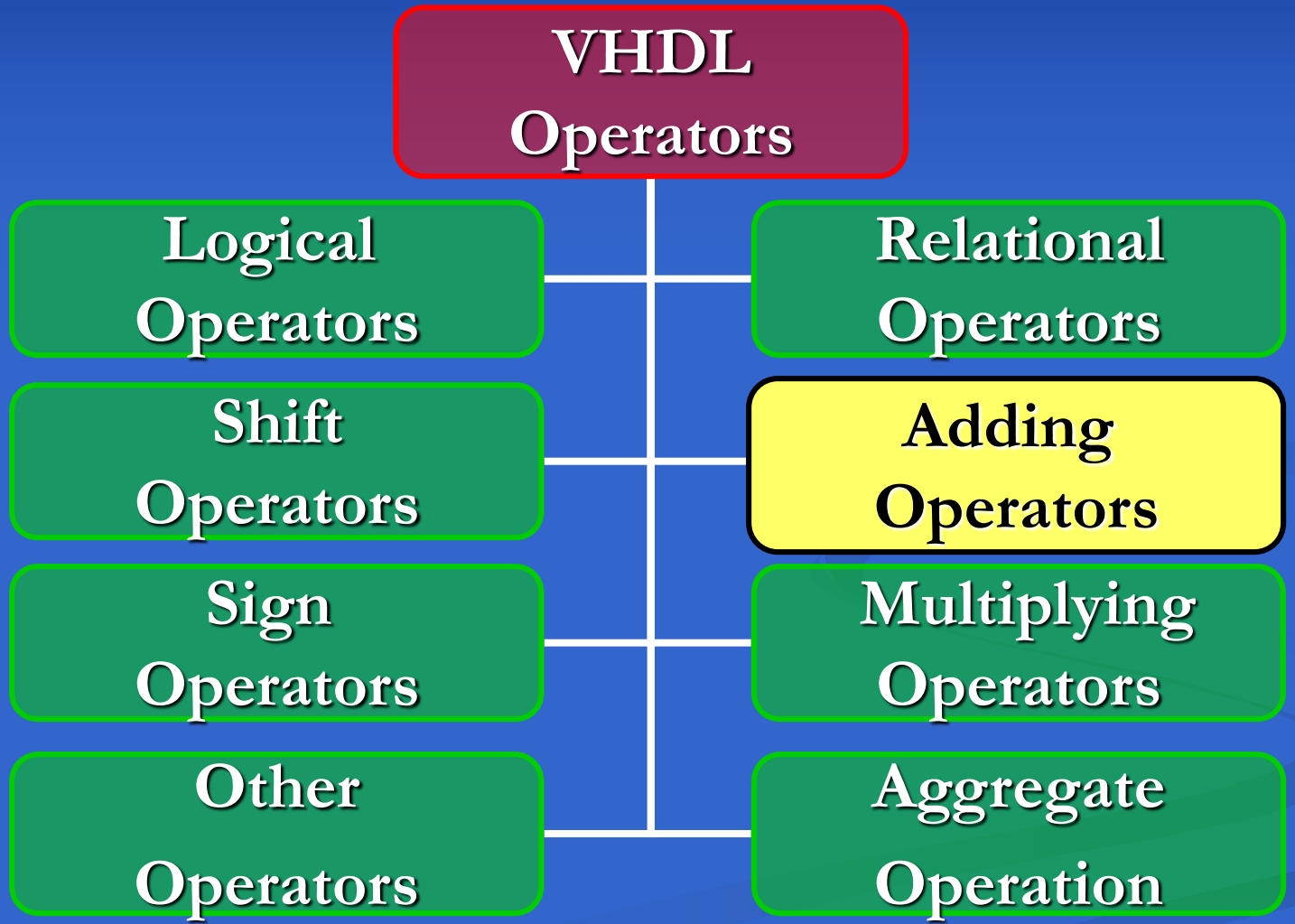
- **Shift Operators**

Shift Operators

Start with av=	Z	0	1	X	Z	1	0	1
av SLL 1	0	1	X	Z	1	0	1	X
av SLA 1	0	1	X	Z	1	0	1	1
av SRL 1	X	Z	0	1	X	Z	1	0
av SRA 1	Z	Z	0	1	X	Z	1	0
av ROL 1	0	1	X	Z	1	0	1	Z
av ROR 1	1	Z	0	1	X	Z	1	0

- Application of Shift Operators

Adding Operators



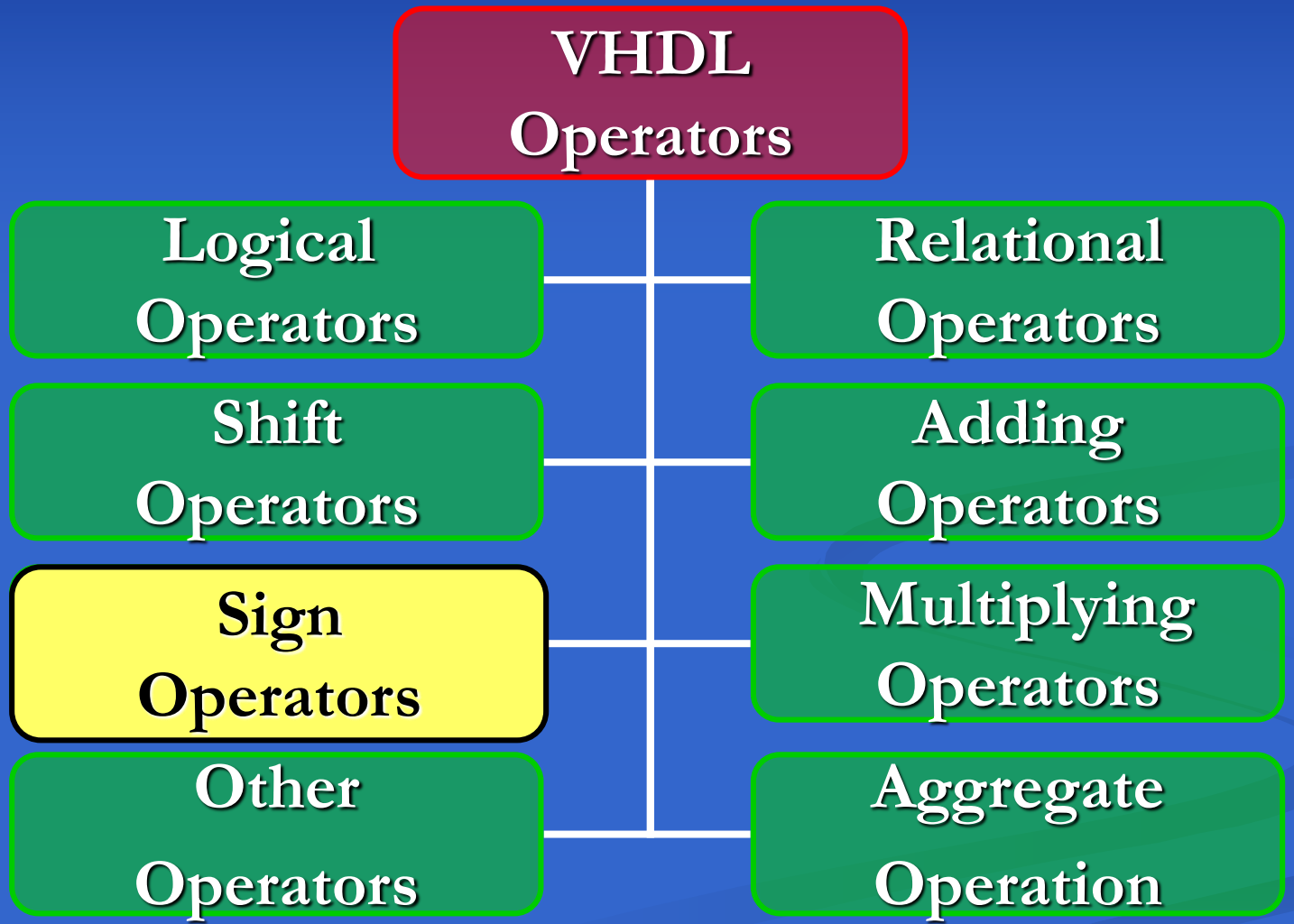
Adding Operators

- Addition, subtraction, and concatenation form the adding group of operators.

- Add and subtract are defined for numeric types of **INTEGER** and **REAL**.
- Both operands of an adding operator must have the same type.
- Add and subtract **are not defined** for **BIT** or **BIT_VECTOR** types, but VHDL packages for defining such operations are available.
- As with other operators, an adding operator can be used in the following two formats:
  - a + b
  - "+" (a, b)

- Operands of a concatenation operator must be arrays or elements of the same type. Concatenating two scalars of the same type forms an array of size 2.

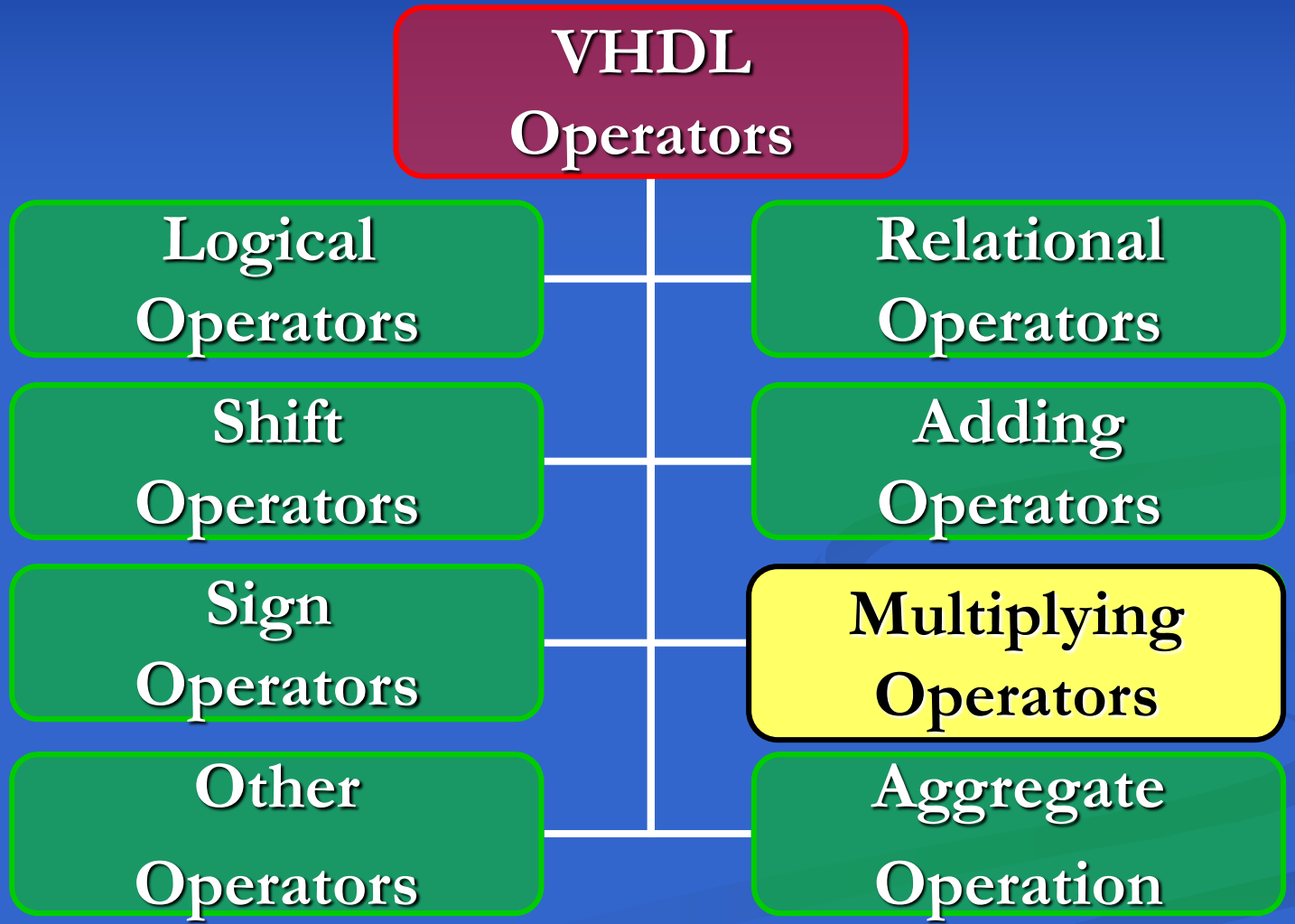
Sign Operators



Sign Operators

- Sign operators + and – are unary operators that apply to numeric types.

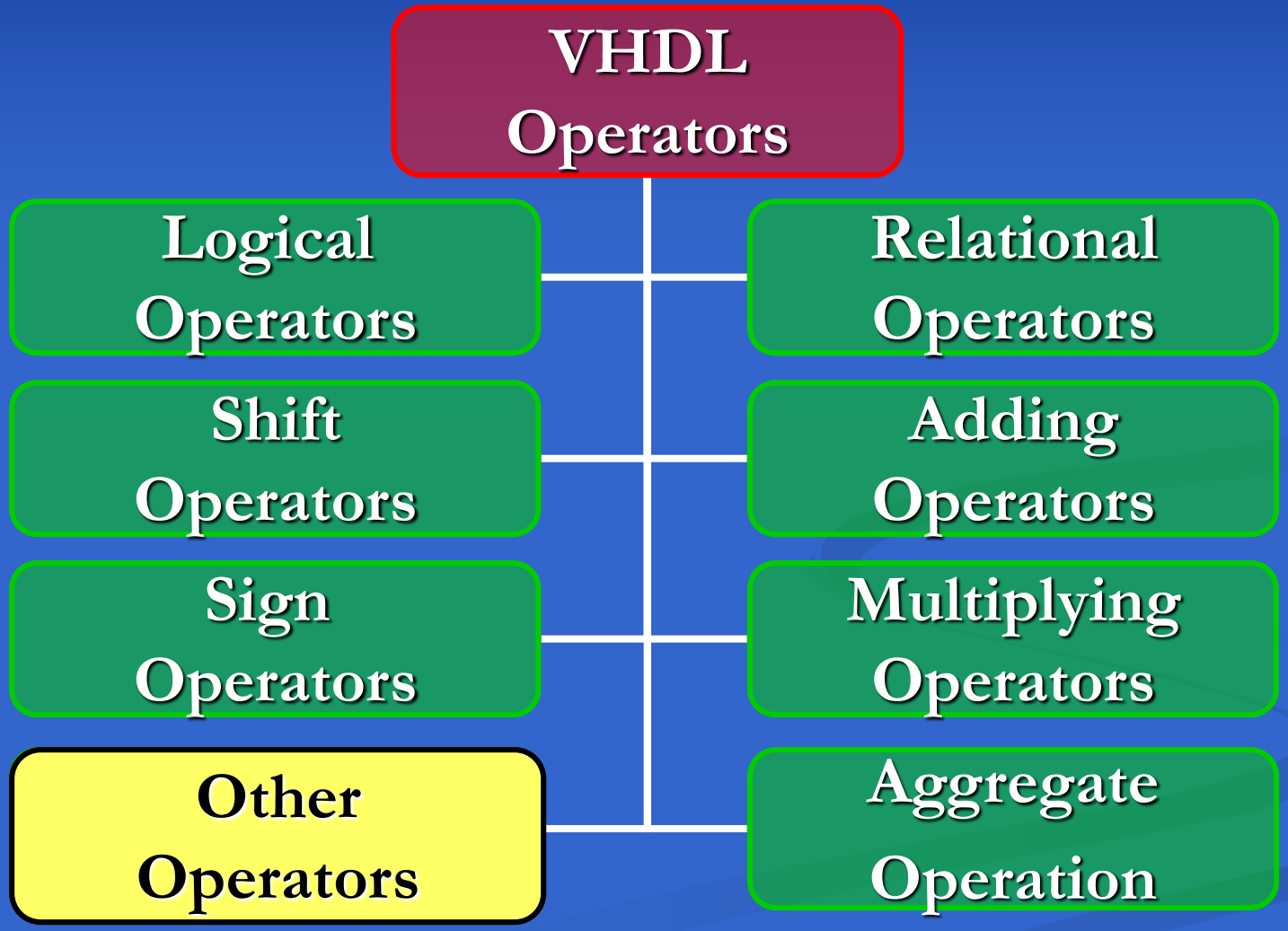
Multiplying Operators



Multiplying Operators

- The four multiplying operators are *, /, MOD, and REM.
- Multiplication and division have their conventional mathematical meanings and are defined for operands of the same type of INTEGER or REAL.
- Both operands of MOD and REM operators must be of the INTEGER type.
- The remainder, REM, operator returns the remainder of integer division of the absolute value of its left operand by the absolute value of its right operand. The sign of the result is the same as that of the left operand.
- The modulus, MOD, operator calculates the modulus of its left and right operands. The sign of the result is the same as that of the right operand.

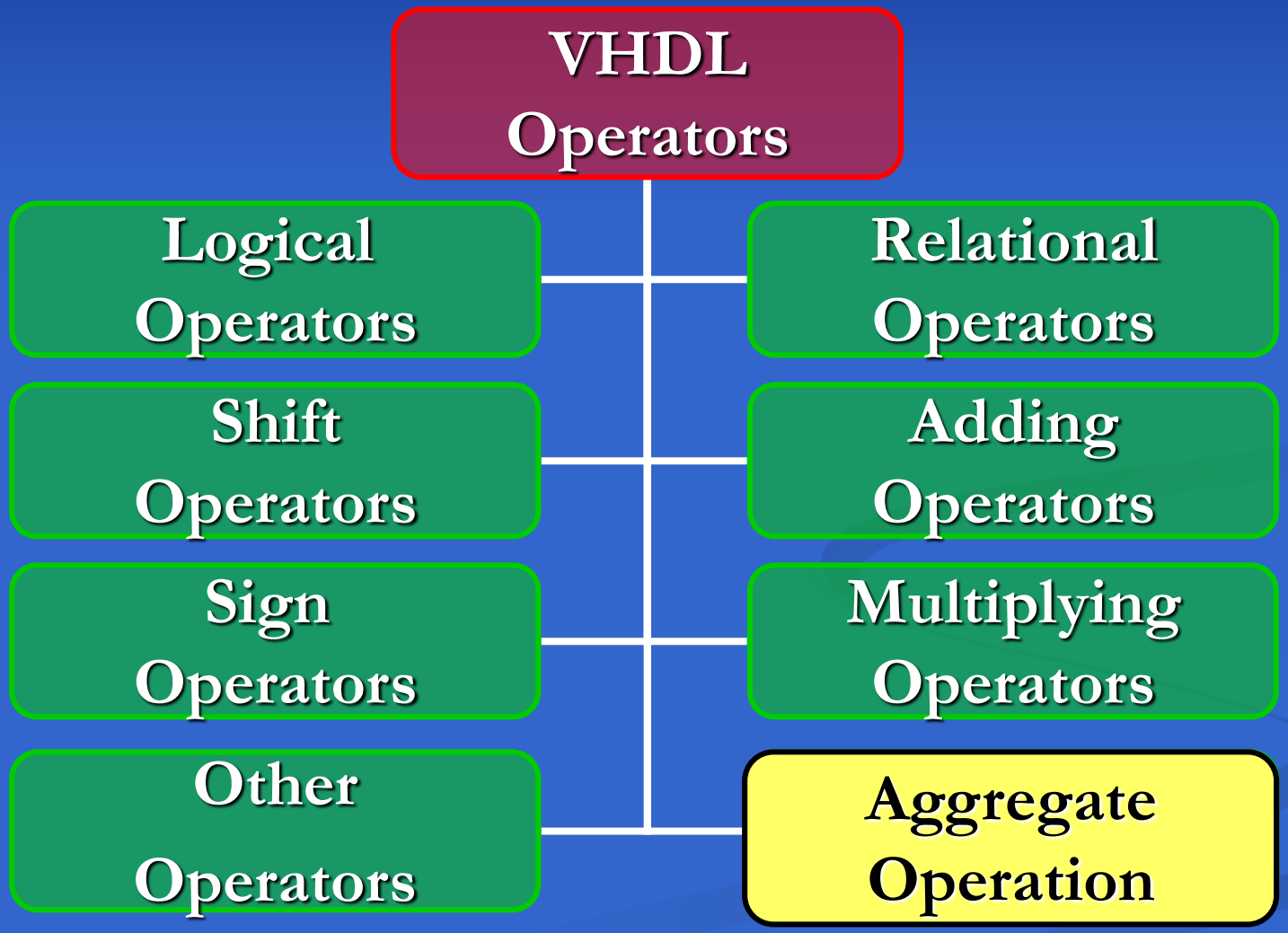
Other Operators



Other Operators

- ** (exponential)
- ABS (absolute value)

Aggregate Operation

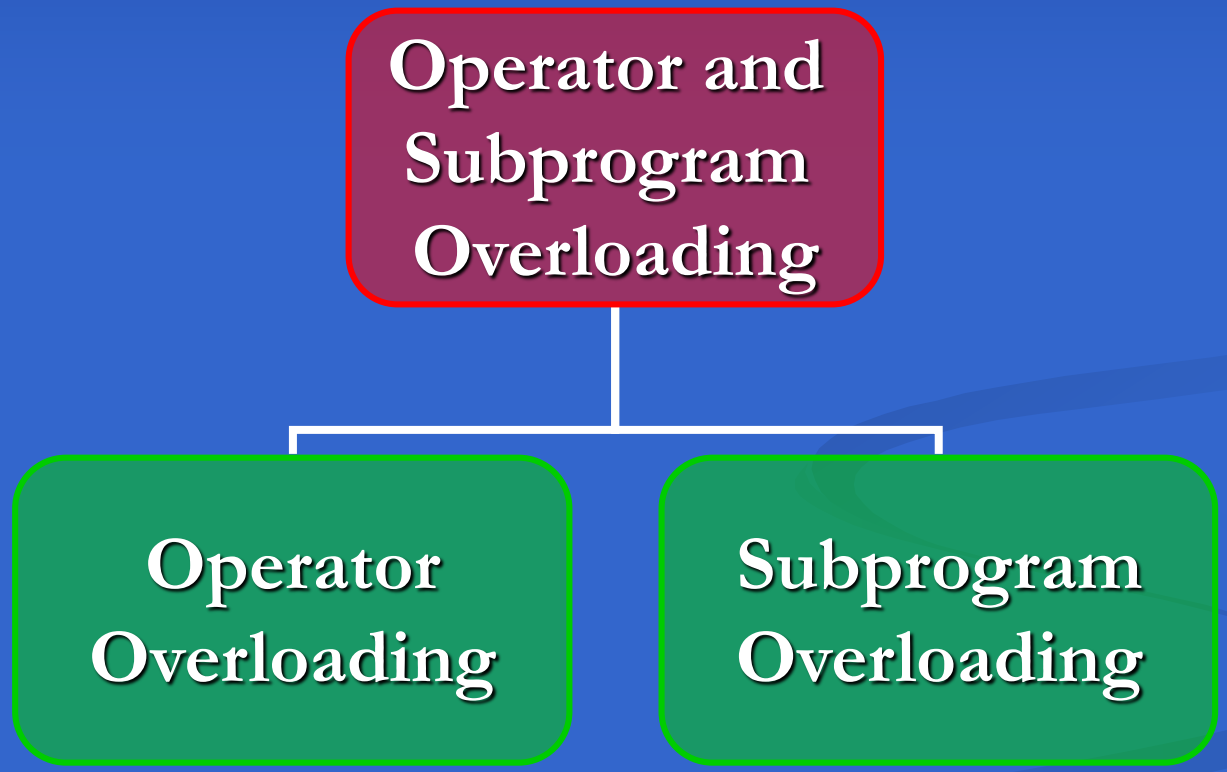


Aggregate Operation

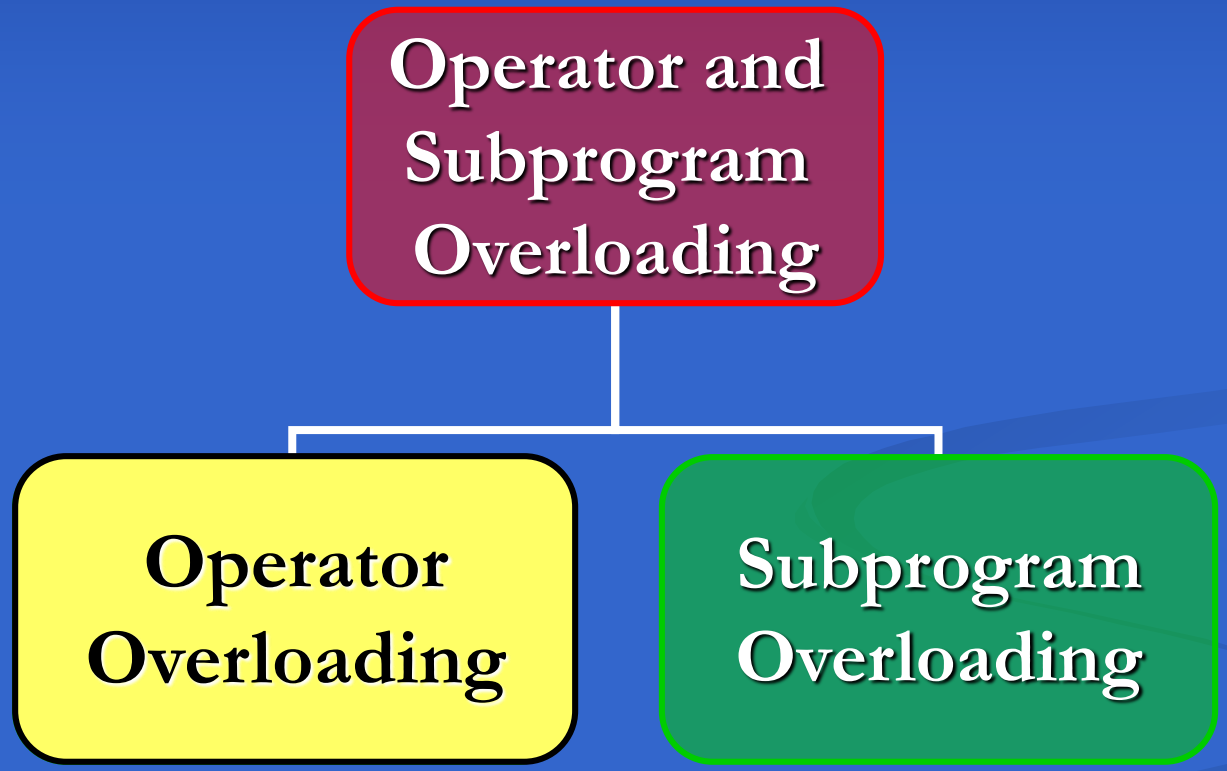
- An aggregate operation combines one or more values into a complex array or record type.
- Assuming *a* and *b* are objects of BIT type:
  - (a, b) , a & b are equivalent
  - The first expression uses an aggregate operation to form a 2-bit vector, and the second expression concatenates *a* and *b* together. A

- Aggregate operation can only be applied to elements of the same size and type.
- Concatenation, on the other hand, can be used to concatenate different-size arrays of the same element type.
- An aggregate operation applies to records as well as arrays.
- An aggregate can be done on the left-hand side of a signal assignment:
  - (a, b) <= a2;
  - (a, b) <= "10";
  - (a, b) <= ('1', '0');

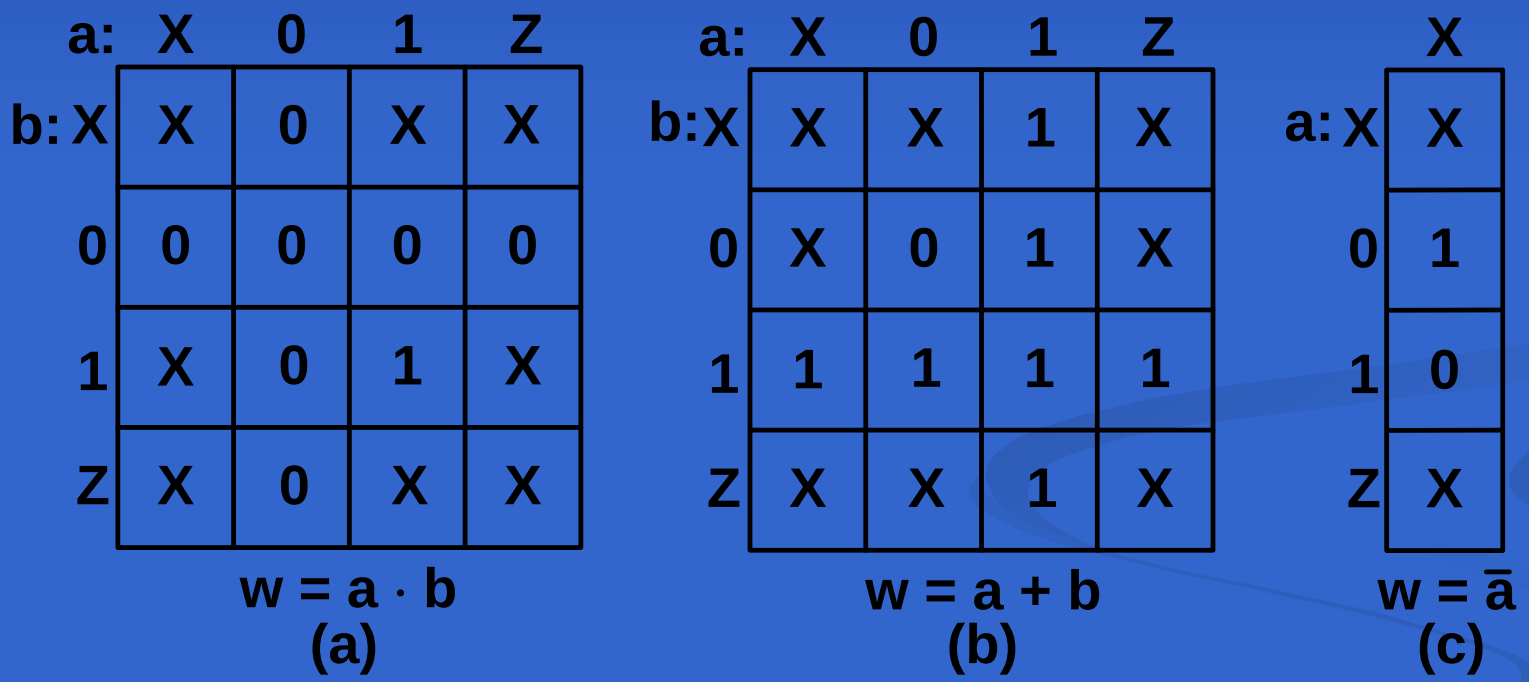
Operator and Subprogram Overloading



Operator Overloading



Operator Overloading



b \ a	X	0	1	Z
X	X	0	X	X
0	0	0	0	0
1	X	0	1	X
Z	X	0	X	X

w = a · b
(a)

b \ a	X	0	1	Z
X	X	X	1	X
0	X	0	1	X
1	1	1	1	1
Z	X	X	1	X

w = a + b
(b)

a	X
X	X
0	1
1	0
Z	X

w = ā
(c)

- Verilog 4-Value Logic Operations Used for *v41*

Operator Overloading

```
FUNCTION "AND" (a, b : v4l) RETURN v4l IS
   CONSTANT v4l_and_table : v4l_2d := (
      'X' =>   ('X','0','X','X'),
      '0' =>   ('0','0','0','0'),
      '1' =>   ('X','0','1','X'),
      'Z' =>   ('X','0','X','X'));
BEGIN
   RETURN v4l_and_table (a, b);
END "AND";
```

- Overloading AND Logical Function for the v4l Four Value Logic System

Operator Overloading

```
LIBRARY utilities;
USE utilities.VerilogLogic.ALL;

ENTITY multiplexer IS
   PORT (a, b, s : IN v4l; w : OUT v4l);
END ENTITY;
--
ARCHITECTURE booloeanlevel OF multiplexer IS
BEGIN
   w <= (a AND NOT s) OR (b AND s);
END ARCHITECTURE booloeanlevel;
```

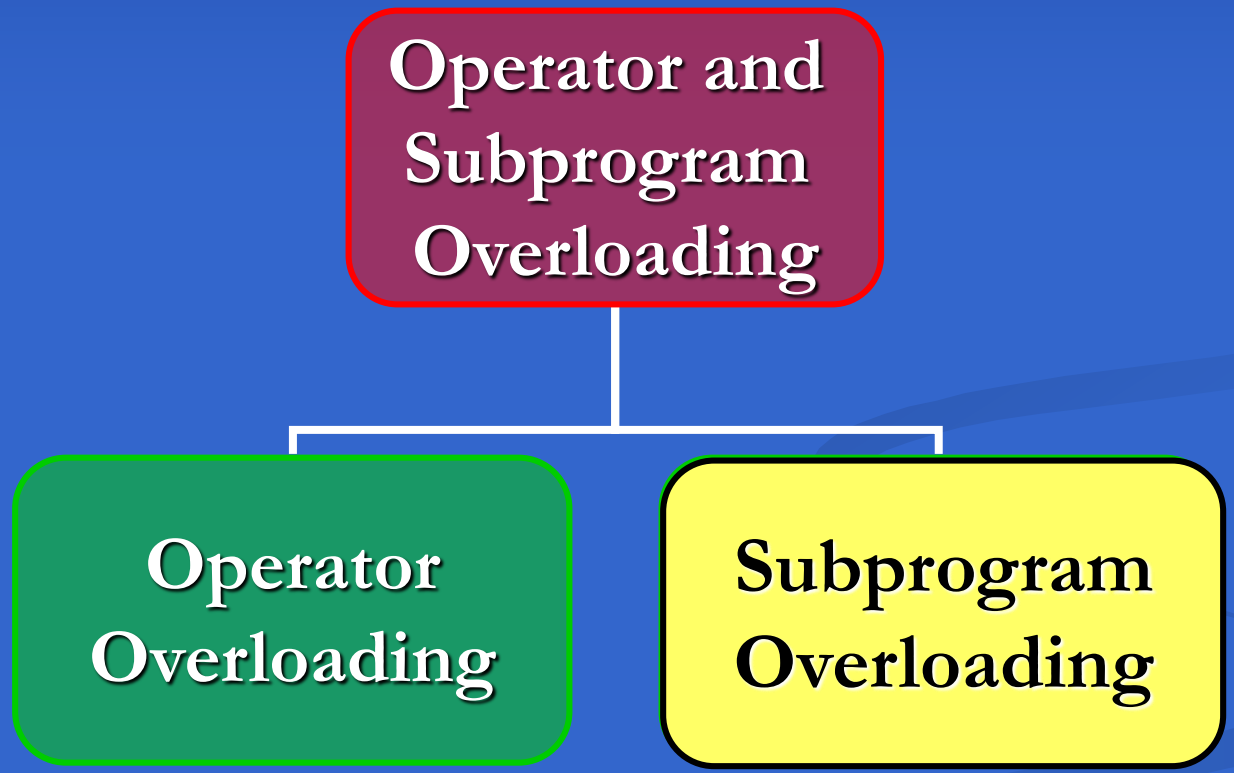
- Using Overloaded Operators

Operator Overloading

```
FUNCTION "*" (a : resistance; b : capacitance)
   RETURN TIME IS
BEGIN
   RETURN  ( ( a / 1 l_o) * ( b / 1 ffr ) * 1 FS ) /
           1000;
END "*";
```

- Overloading: Multiplying *Resistance* and *Capacitance* Resulting TIME

Subprogram Overloading



Subprogram Overloading

```
TYPE mem IS ARRAY (NATURAL RANGE <>,
                   NATURAL RANGE <>) of BIT;
TYPE bit_filetype IS FILE OF CHARACTER;

PROCEDURE dump_mem (VARIABLE memory : IN mem;
                    CONSTANT datafile : STRING) IS
   FILE BIT_data : BIT_filetype;
   VARIABLE BIT_value : BIT;
   TYPE BIT_char IS ARRAY (BIT) OF CHARACTER;
   CONSTANT BIT_tochar : BIT_char := ('0', '1');
BEGIN
   ......................
   ......................
END PROCEDURE dump_mem;
```

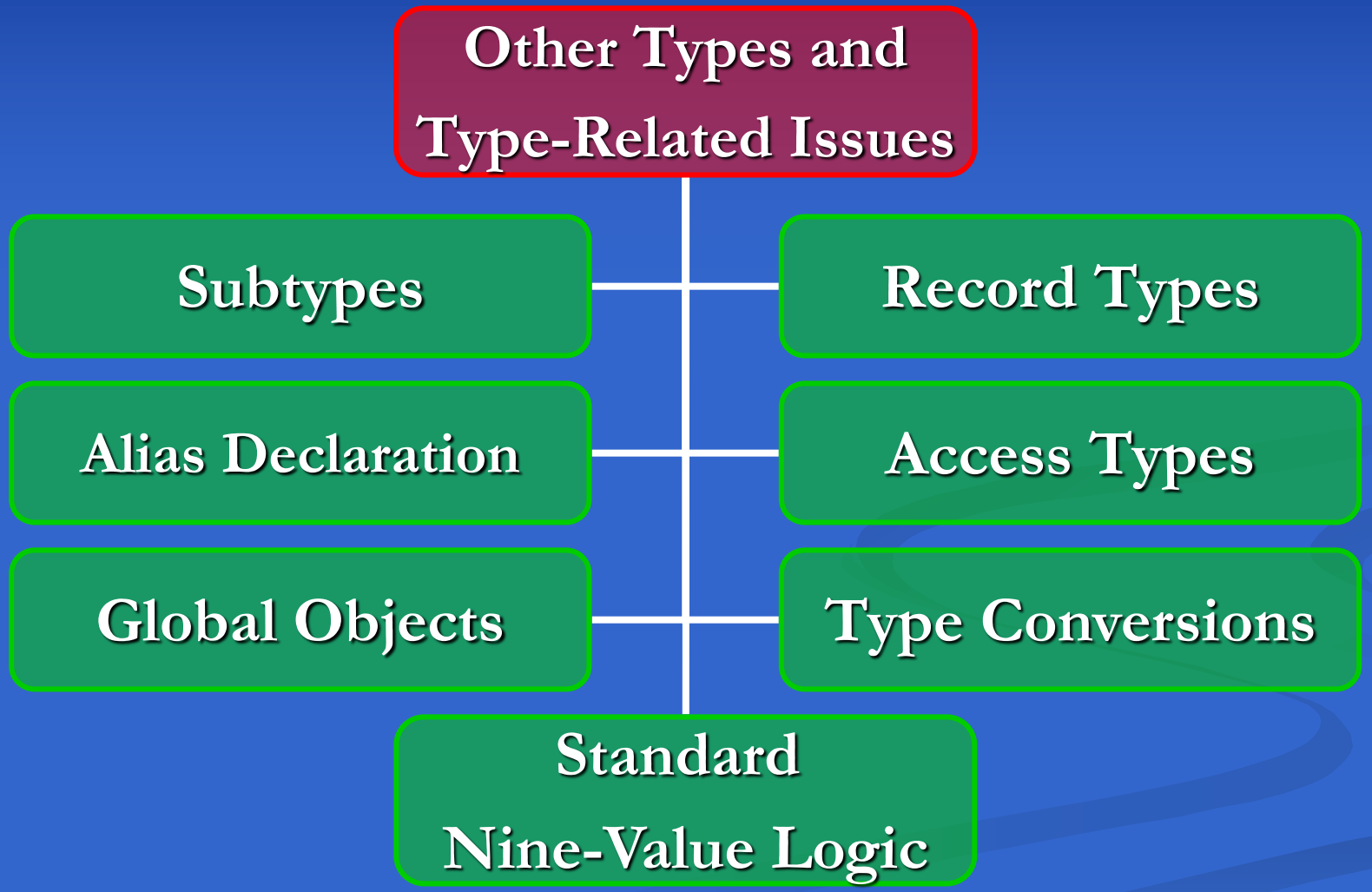
- Overloaded Memory Dump Procedure

Subprogram Overloading

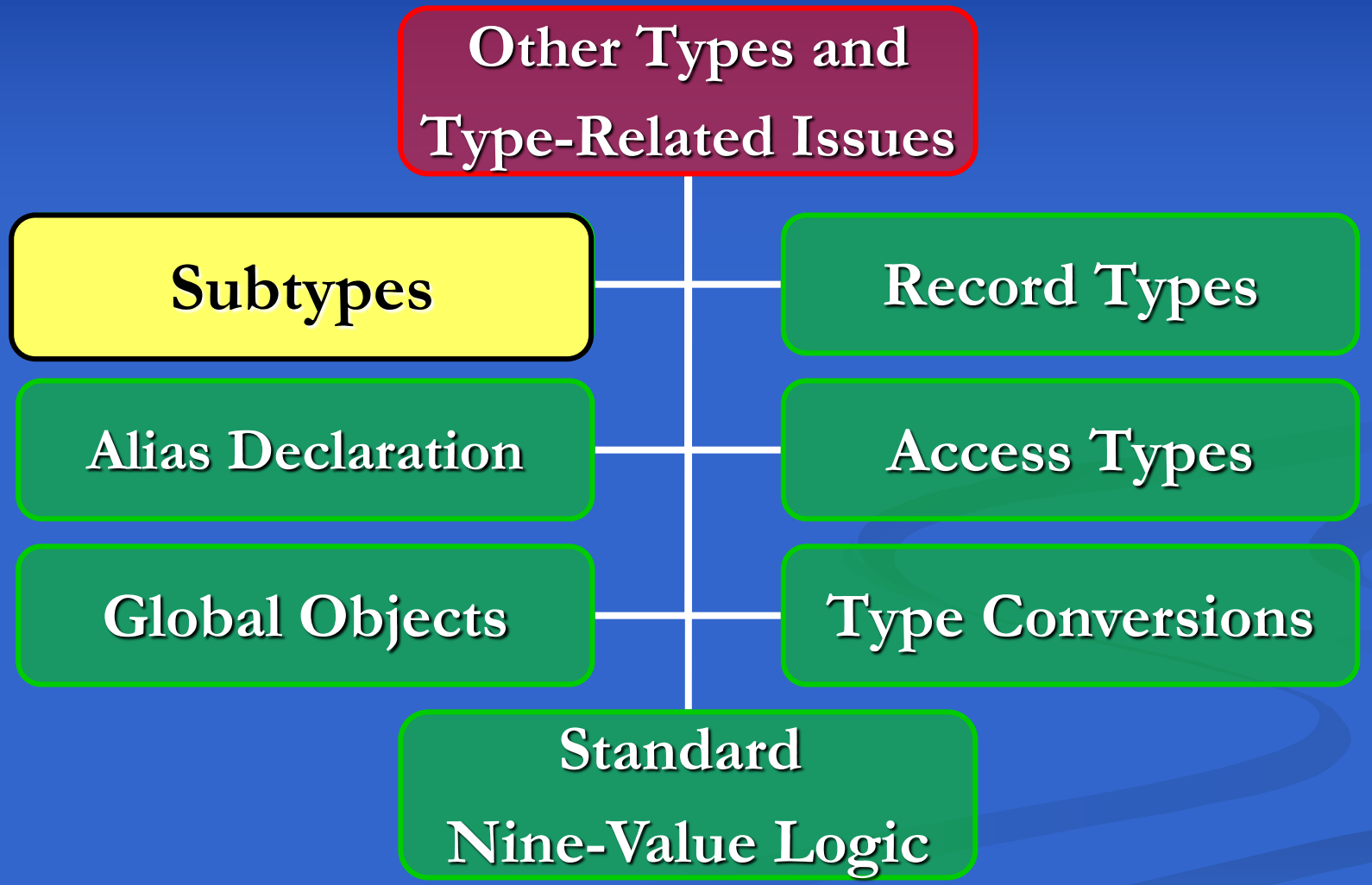
```
.....................
.....................
BEGIN
   FILE_OPEN (BIT_data, datafile, WRITE_MODE);
   FOR i IN memory'RANGE(1) LOOP
      FOR j IN memory'REVERSE_RANGE(2) LOOP
         BIT_value := memory (i, j);
         WRITE (BIT_data, BIT_tochar (BIT_value));
      END LOOP;
      WRITE (BIT_data, cr);
   END LOOP;
END PROCEDURE dump_mem;
```

- Overloaded Memory Dump Procedure (Continued)

Other Types and Type-Related Issues



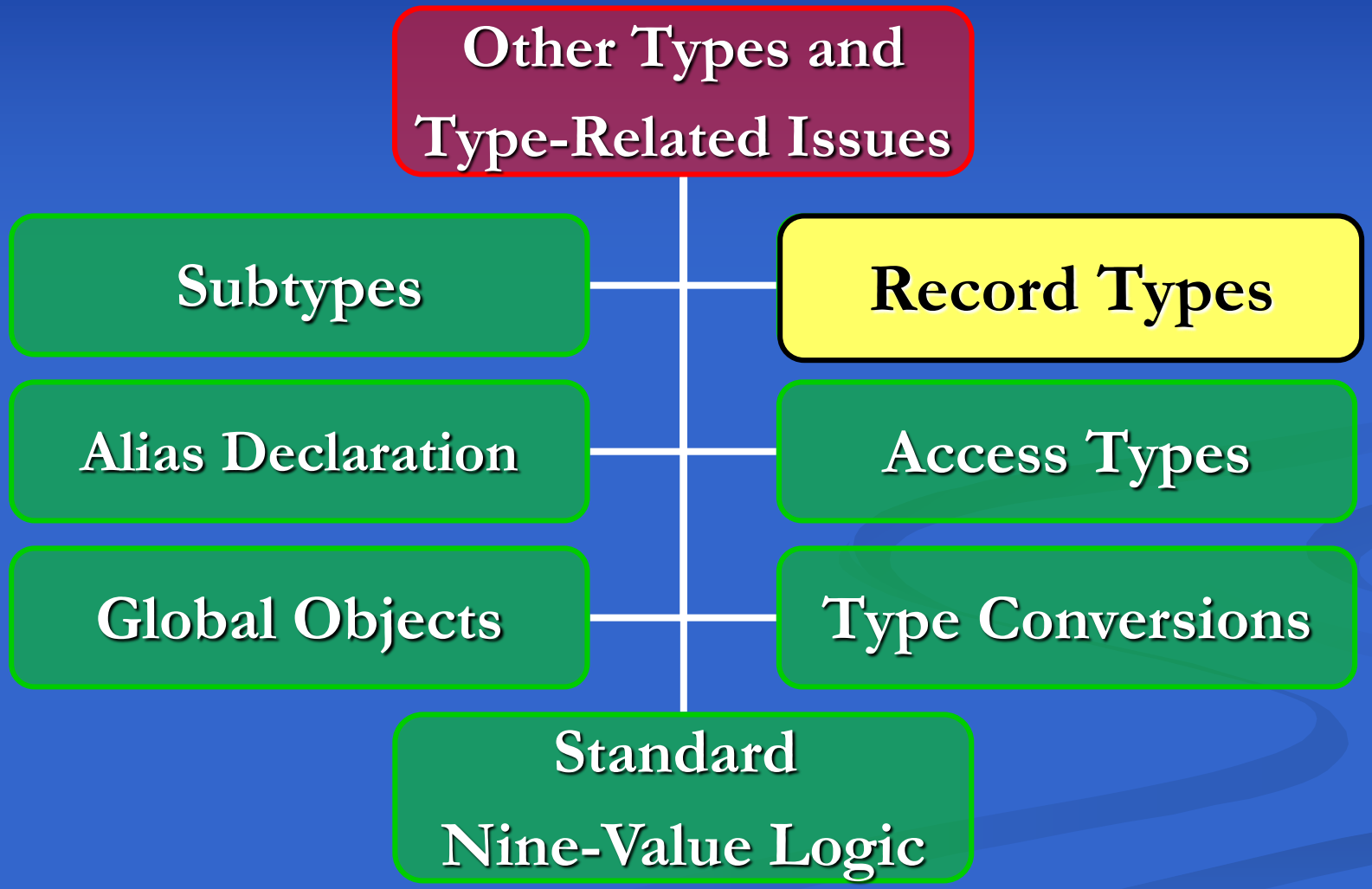
Subtypes


Other Types and
Type-Related Issues
Subtypes
Record Types
Alias Declaration
Access Types
Global Objects
Type Conversions
Standard
Nine-Value Logic

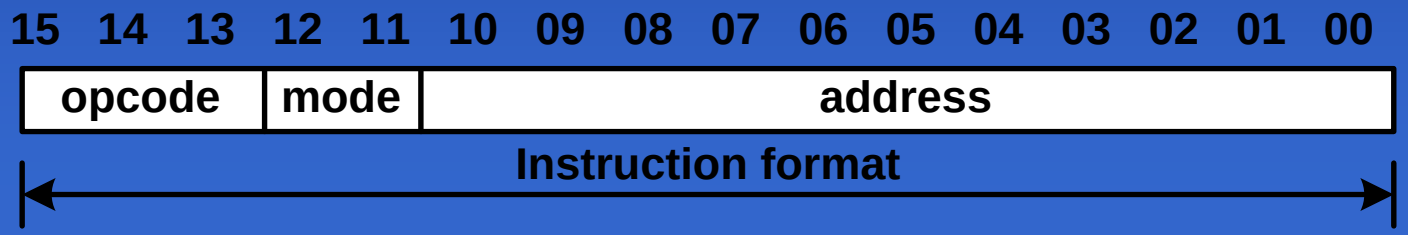
Subtypes

```
SUBTYPE bcd_numbers IS INTEGER RANGE 0 TO 9;

SUBTYPE v31 IS v41 RANGE '0' TO 'Z';

SUBTYPE v21 IS v41 RANGE '0' TO '1';
```

Record Types



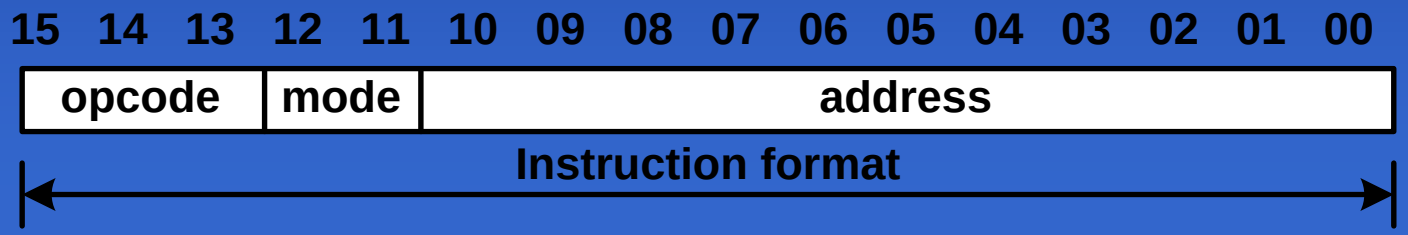
Record Types




```
TYPE opcode IS (sta, lda, add, sub, and, nop, jmp,jsr);
TYPE mode IS RANGE 0 TO 3;
TYPE address IS BIT_VECTOR (10 DOWNTO 0);
```

- Record Type, (a) Three Instruction Fields

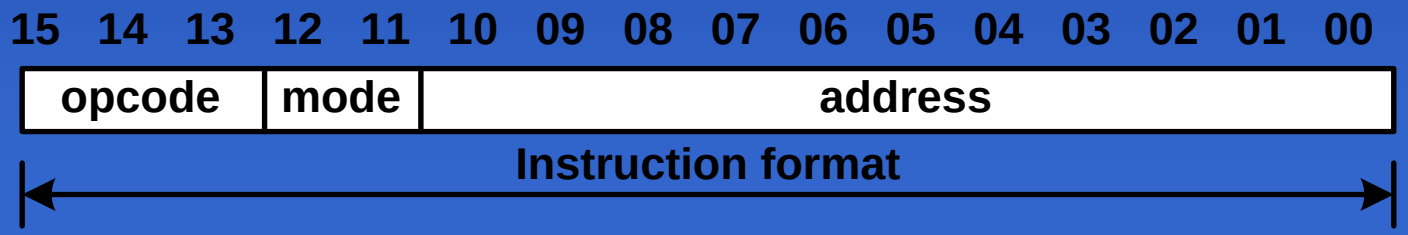
Record Types




```
TYPE instruction_format IS RECORD
       opc : opcode;
       mde : mode;
       adr : address;
END RECORD;
```

- Record Type, (b) Declaration of Instruction Format

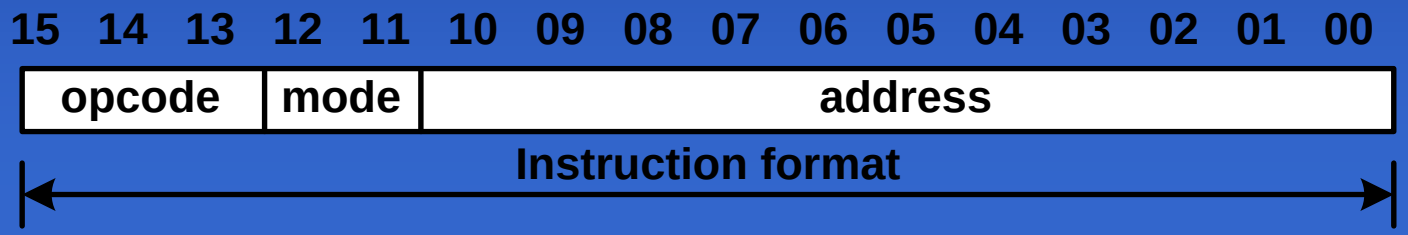
Record Types




```
SIGNAL instr : instruction_format := (nop, 0,
                                      "00000000000");
```

- Record Type, (c) A Signal of Record Type

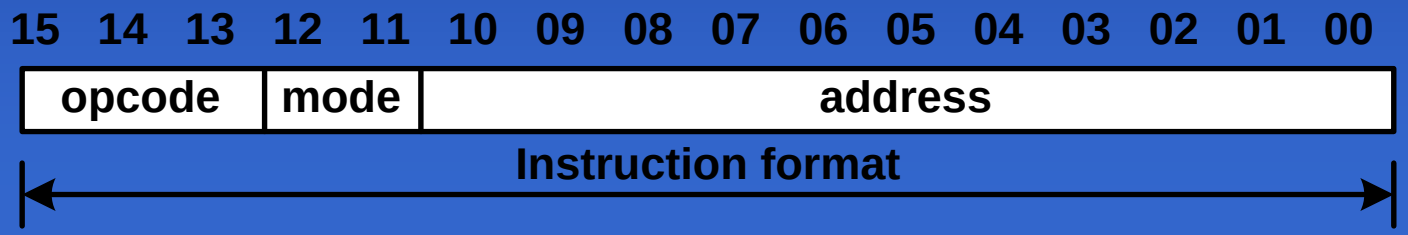
Record Types




```
instr.opc  <= lda;
instr.mde <= 2;
instr.adr   <= "00011110000";
```

- Record Type, (d) Referencing Fields of a Record Type Signal

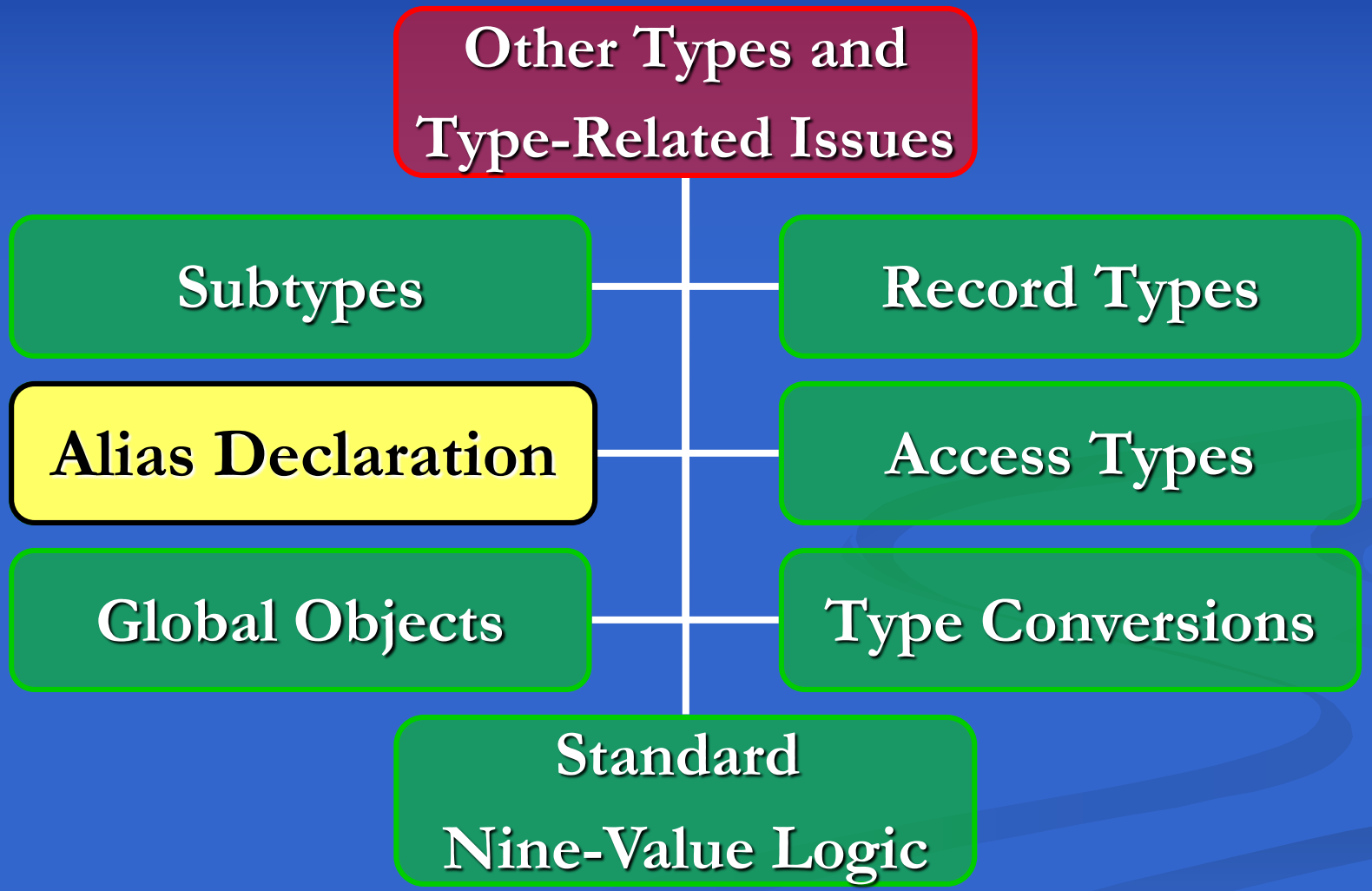
Record Types



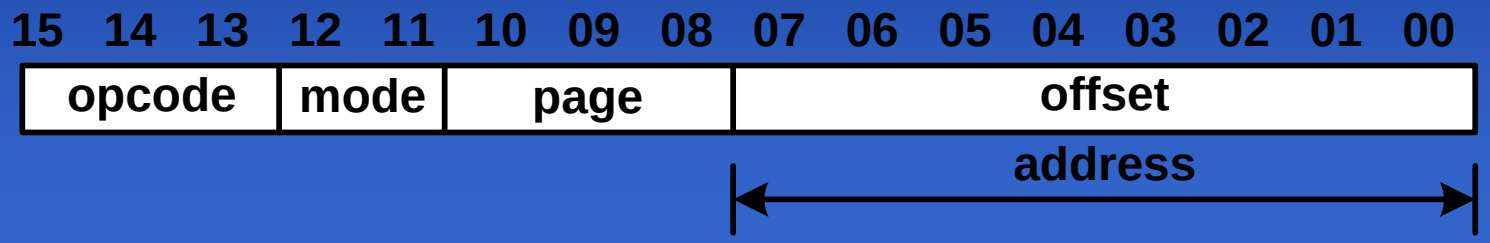

```
instr <= (adr => (OTHERS => '1'), mde => 2,
          opc => sub)
```

- Record Type, (e) Record Aggregate

Alias Declaration



Alias Declaration



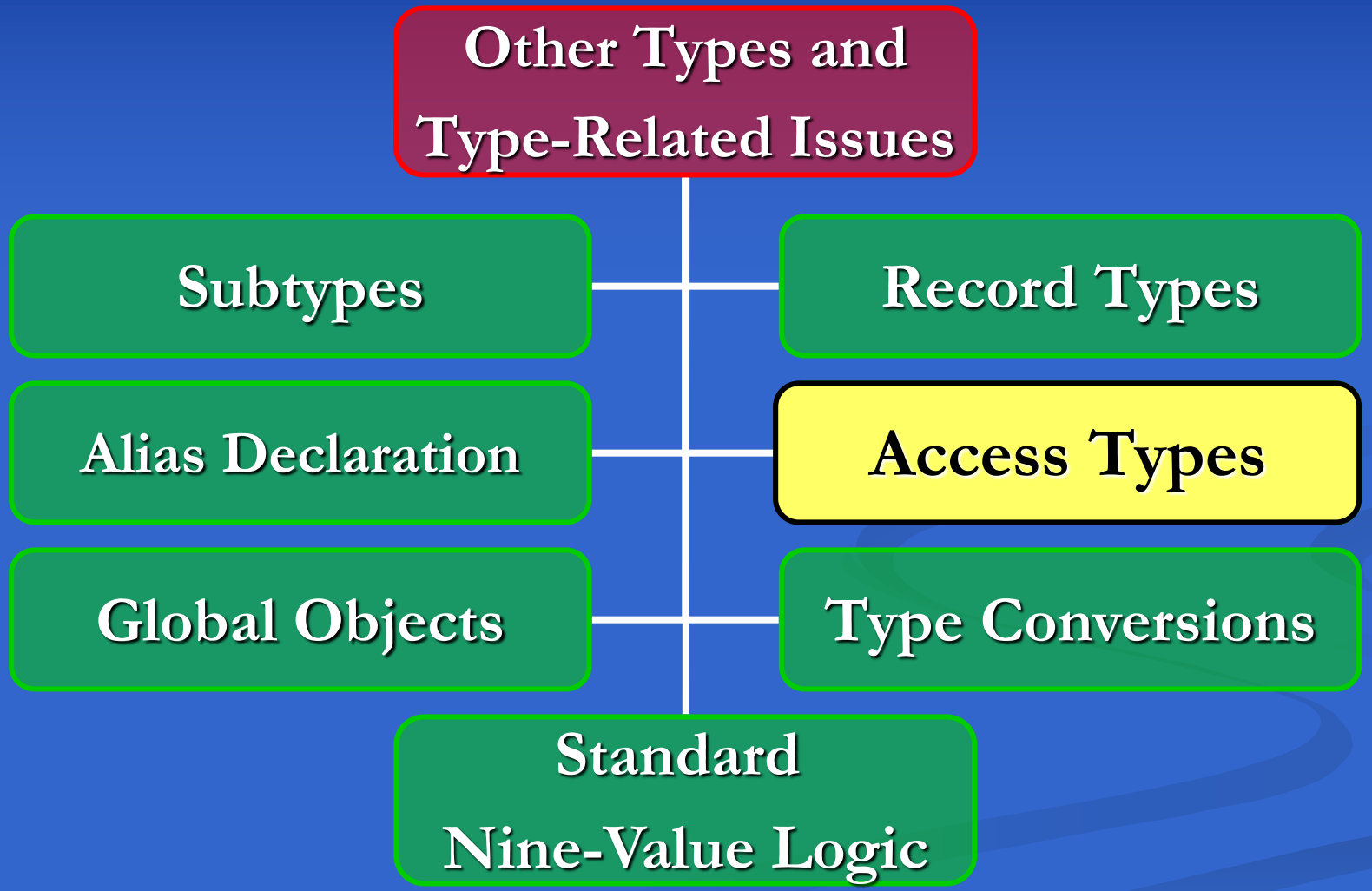

```
ALIAS page :
   BIT_VECTOR (2 DOWNTO 0) IS instr.adr (10 DOWNTO 8);
ALIAS offset :
   BIT_VECTOR (7 DOWNTO 0) IS instr.adr (7 DOWNTO 0);
```

```
page <= "001";
offset <= X"F1";
```

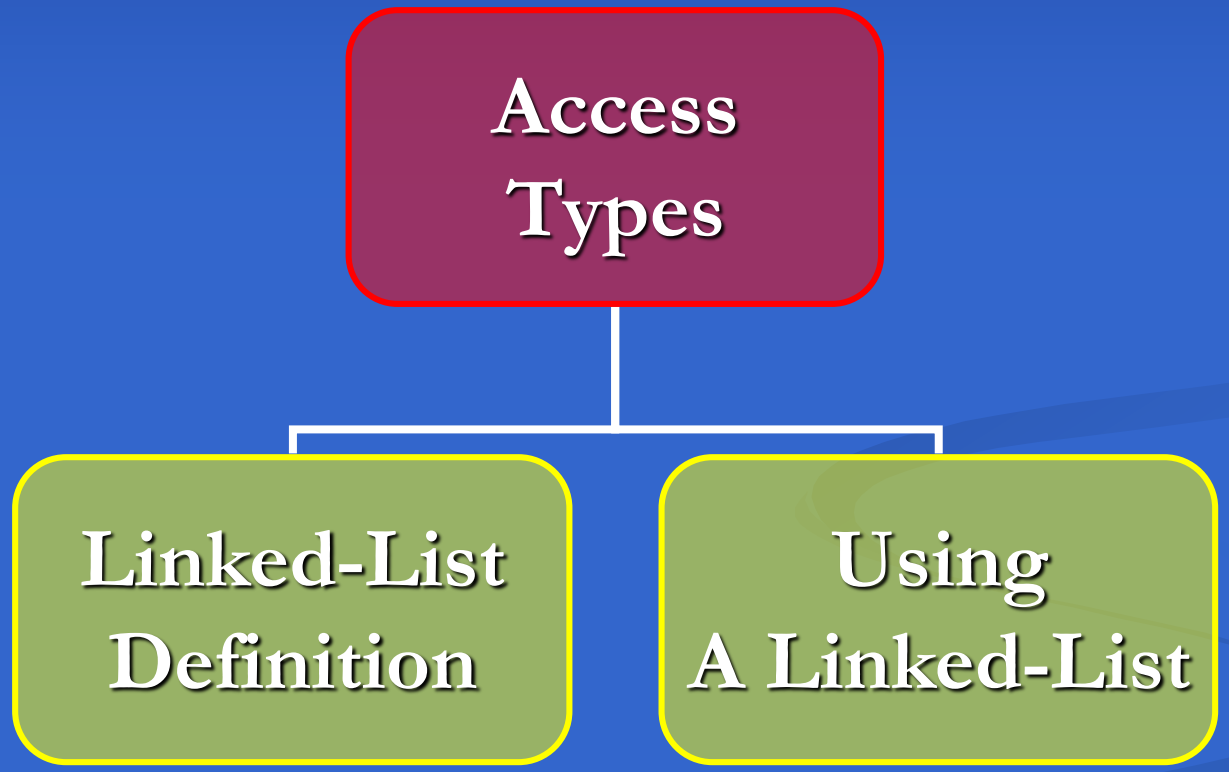
- Alias Declaration, (a) Page and Offset Addresses,
  (b) Alias Declaration for the Page and Offset Parts of the Address,

  (c) Assignments to Page and Offset Parts of Address

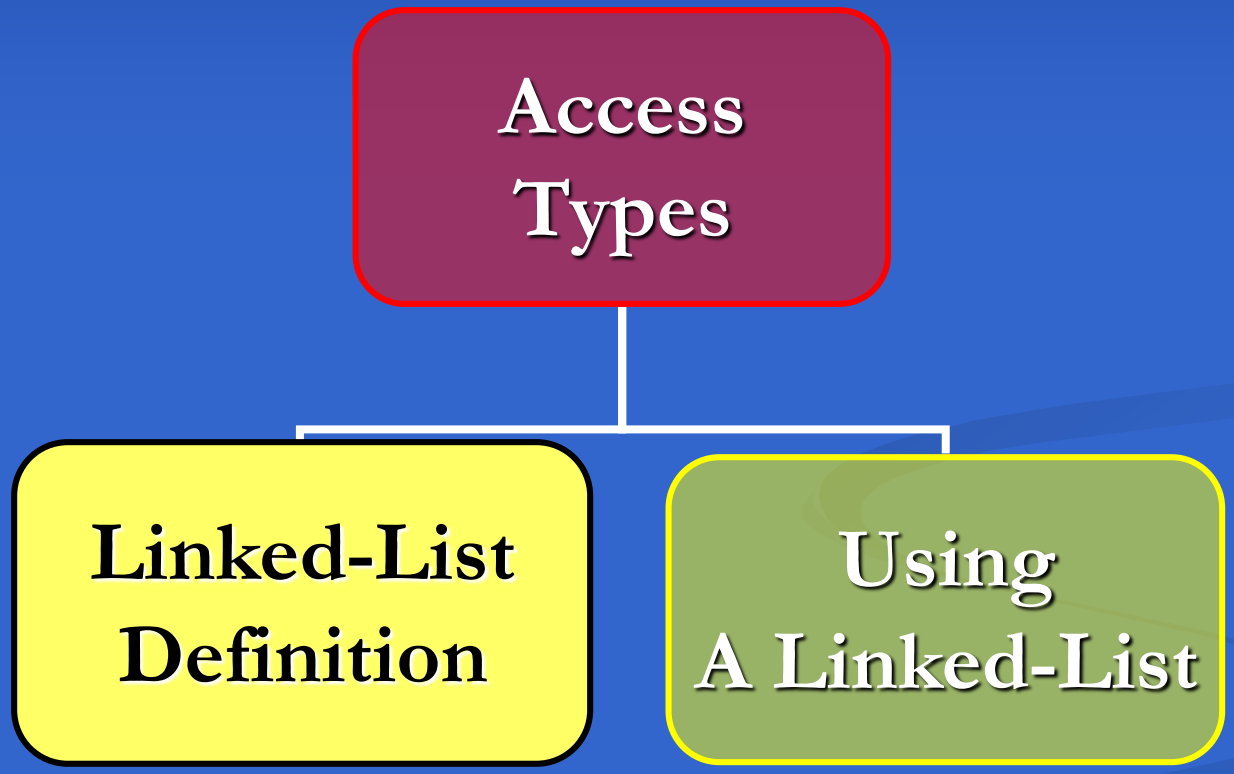
Access Types



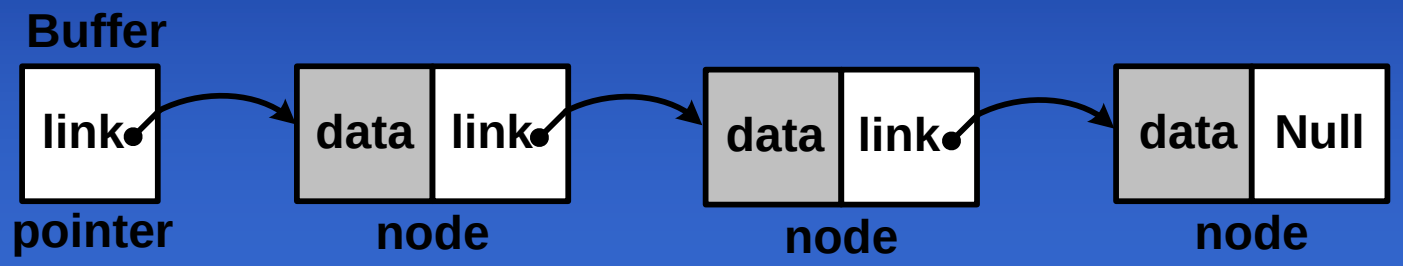
Access Types



Linked-List Definition



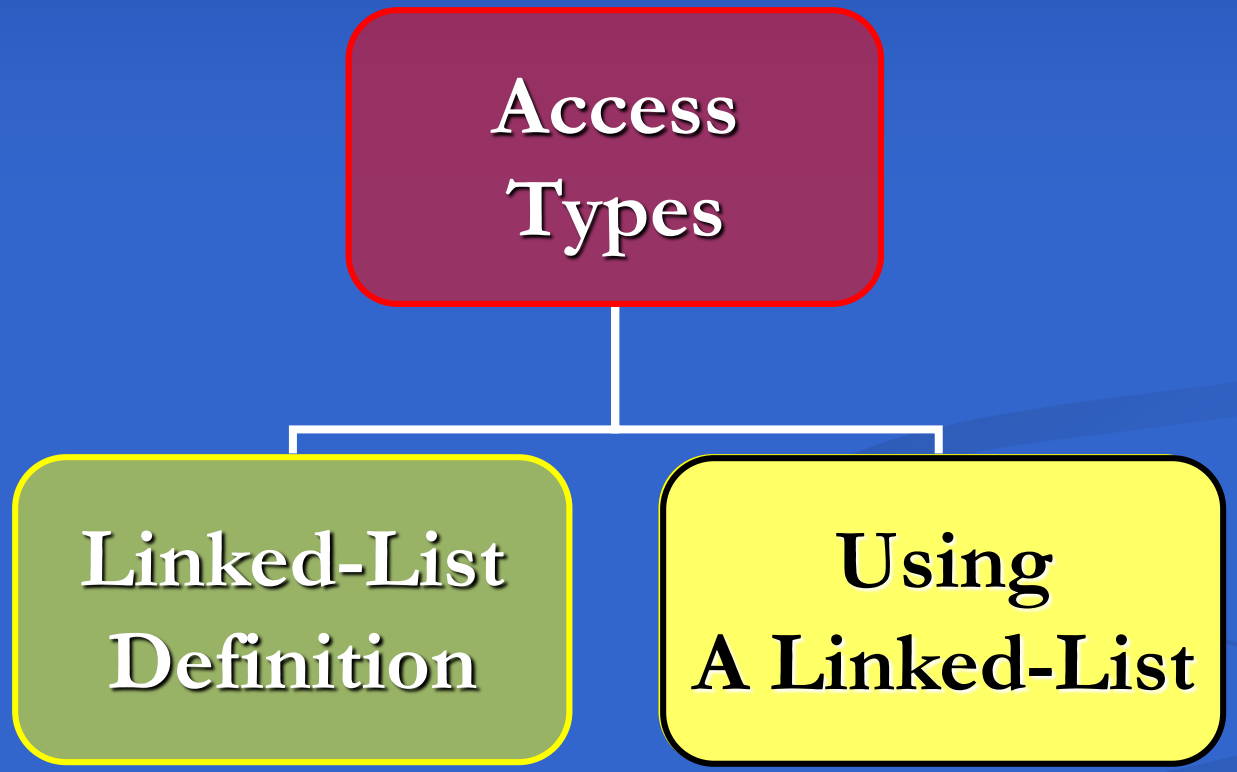
Linked-List Definition




```
TYPE node;
TYPE pointer IS ACCESS node;
TYPE node IS RECORD
   data : INTEGER;
   link : pointer;
END RECORD;
```

- Linked List Graphical Representation and Definition in VHDL

Using A Linked-List



Using A Linked-List

```
PROCEDURE insert
    (VARIABLE head : INOUT pointer; din : INTEGER)
IS
   VARIABLE t1 : pointer;
BEGIN
   -- Insert a node with value din
   IF head=NULL THEN
      head := NEW node;
      head.data := din;
      head.link := NULL;
      REPORT "The List was originally empty!";
   ELSE
      ..........................
```

- Creating a linked list and entering data

Using A Linked-List

```
        ..........................
        t1 := head;
        WHILE t1.link /= NULL LOOP
           t1 := t1.link;
        END LOOP;
        t1.link := NEW node;
        t1 := t1.link;
        t1.data := din;
        t1.link := NULL;
    END IF;
REPORT "Value:"&INTEGER'IMAGE(din)&" inserted!";
END insert;
```

- Creating a linked list and entering data (Continued)

Using A Linked-List

```
PROCEDURE remove
    (VARIABLE head : INOUT pointer; v : IN INTEGER)
IS
   VARIABLE t1, t2 : pointer;
BEGIN
   t1 := head;
   t2 := head;
   IF head /= NULL THEN
      IF head.data = v THEN
         head := head.link;
         REPORT "Value:"&INTEGER'IMAGE(v)&
                " was in the head and removed!";
         ..........................
```

- Removing an Item From a Linked List

Using A Linked-List

```
    ELSE
        WHILE t1 /= NULL LOOP
            IF t1.data = v THEN
                t2.link := t1.link;
                REPORT "Value:"&INTEGER'IMAGE(v)&
                       " removed!";
                EXIT;
            ELSE
                t2 := t1;
            END IF;
            t1 := t1.link;
        END LOOP;
     END IF;
```

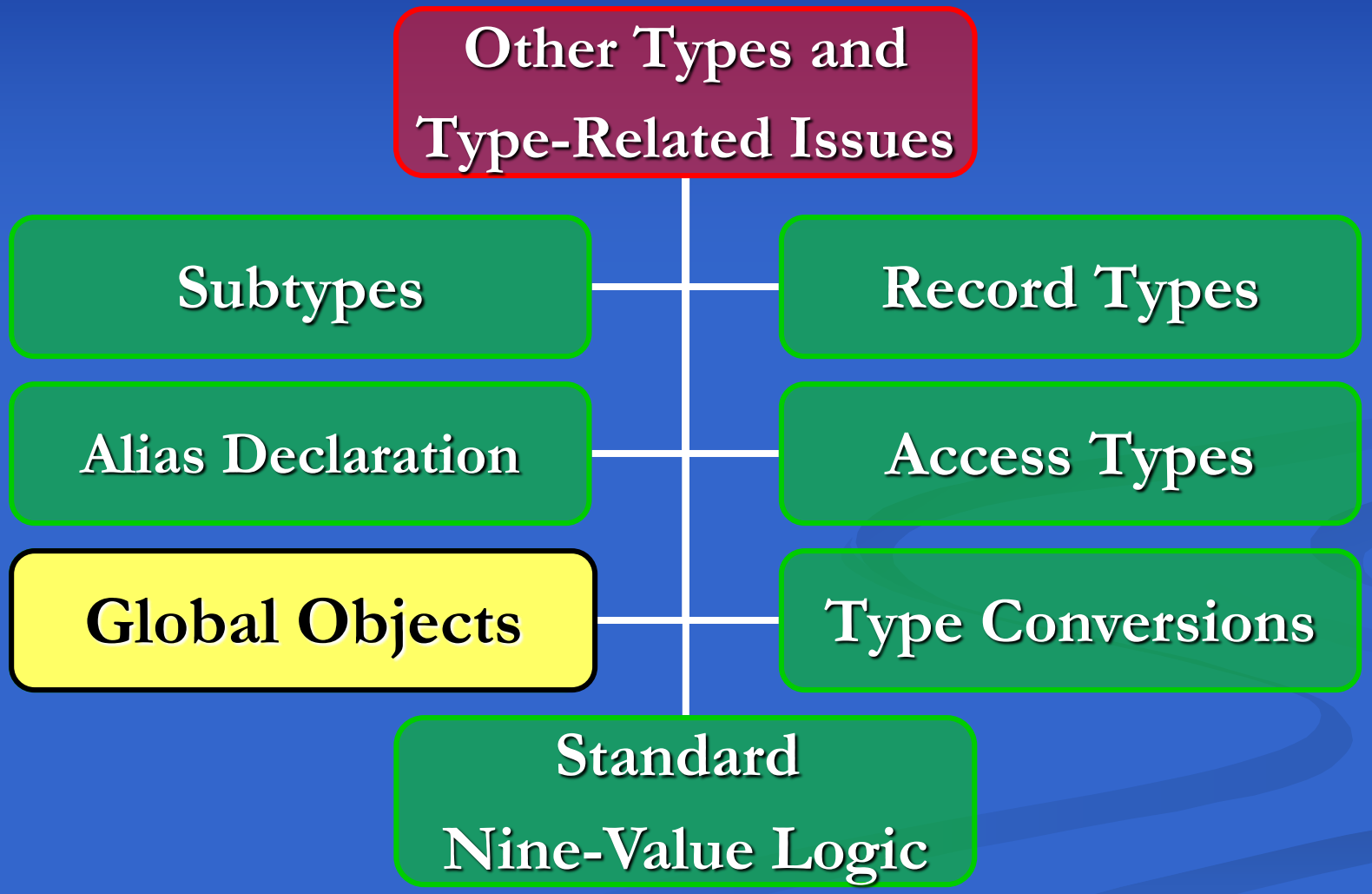
- Removing an Item From a Linked List (Continued)

Using A Linked-List

```
PROCEDURE clear (VARIABLE head : INOUT pointer) IS
   VARIABLE t1, t2 : pointer;
BEGIN
   -- Free all the linked list
   t1 := NEW node;
   t1 := head;
   head := NULL;
   WHILE t1 /= NULL LOOP
      t2 := t1;
      t1 := t1.link;
      DEALLOCATE (t2);
   END LOOP;
   REPORT "The List cleared successfully!";
END clear;
```

- Freeing a Linked List

Global Objects



Global Objects

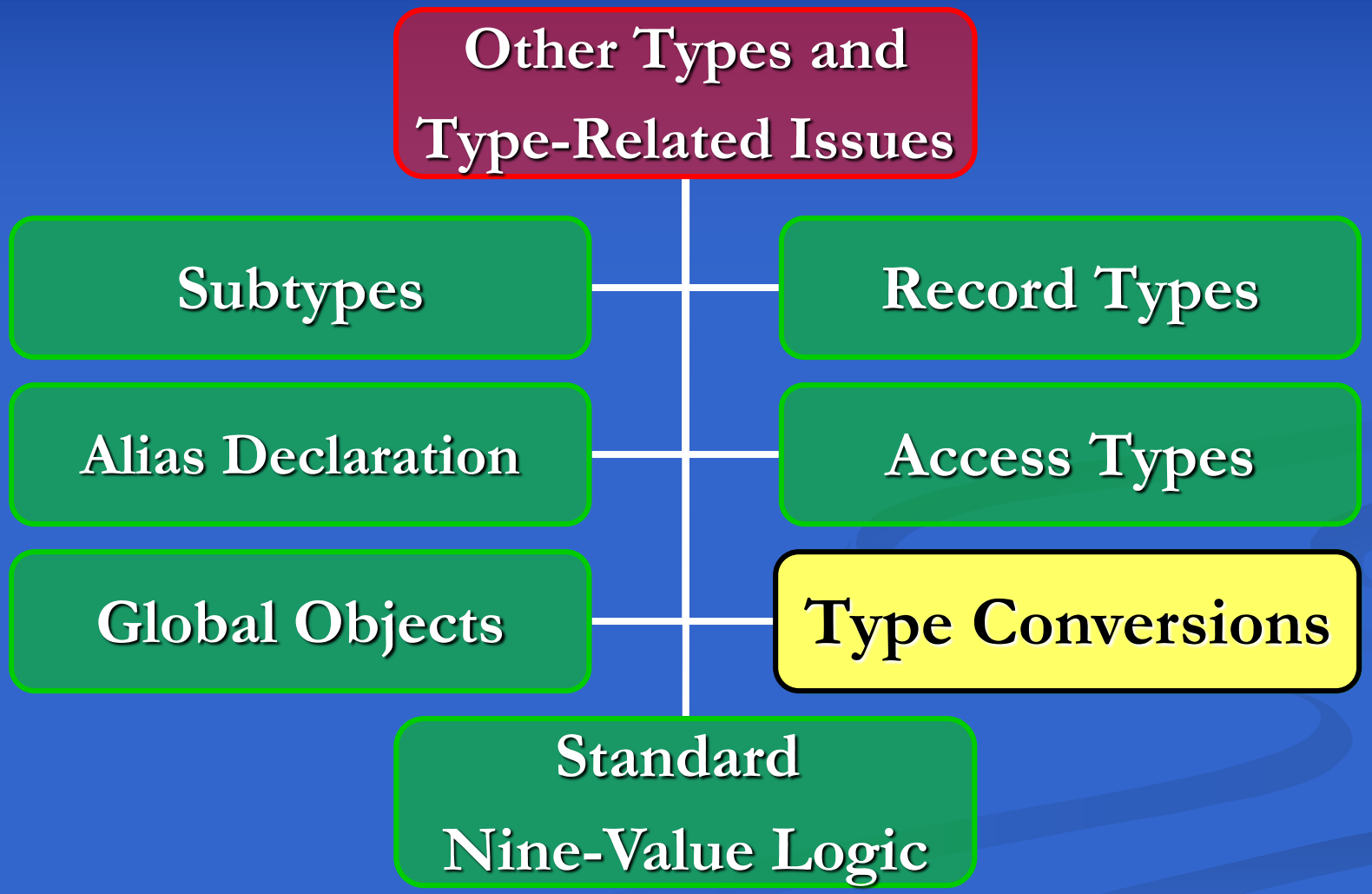
- A signal declared in a package :
  - Can be written to or read by all VHDL bodies that the package is visible to.
- Concurrent writing to a shared signal will be possible only if the signal is resolved. A function for resolving multiple driving values is defined for resolved signals.

- A shared variable declared in a package is accessible to all bodies that use the package.
- The scope of shared variables declared in an architecture is only within the body of the architecture:

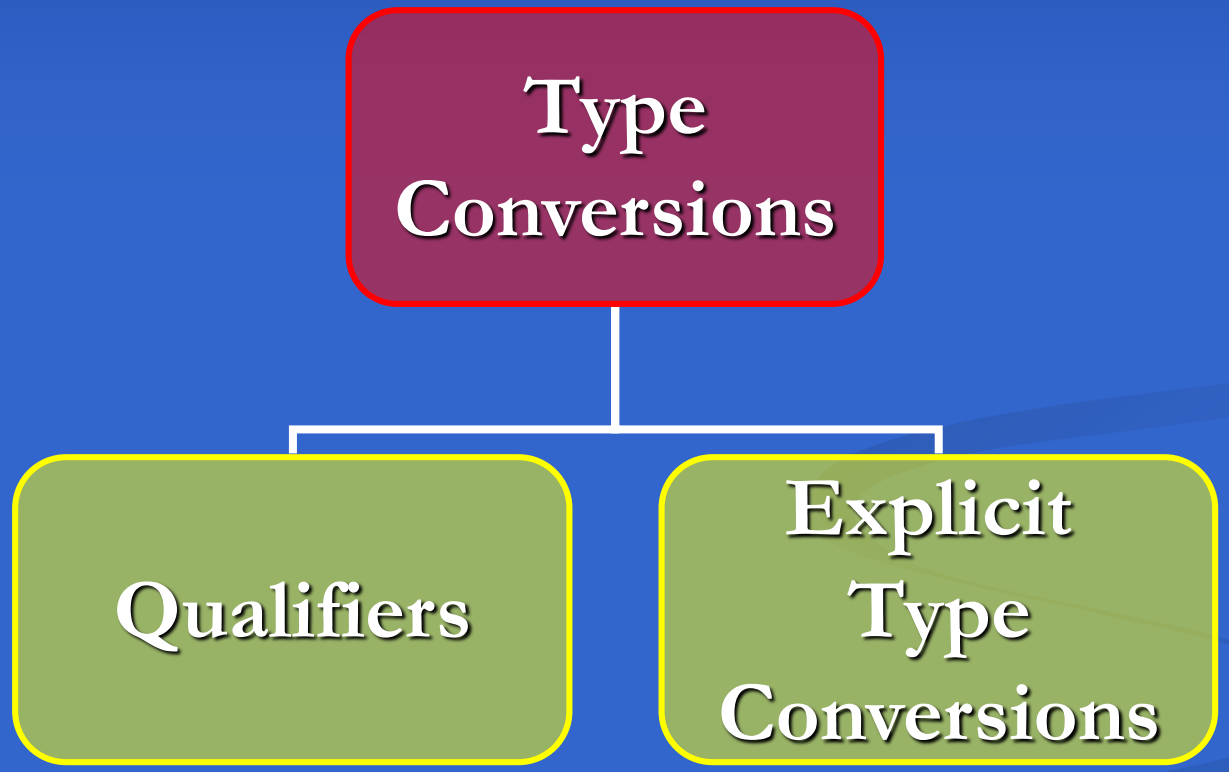
```
SHARED VARIABLE dangerous : INTEGER := 0;
```

- Shared variables are not protected against multiple simultaneous read and write operations. However, signal semaphores for creating such a protection can be done in VHDL.

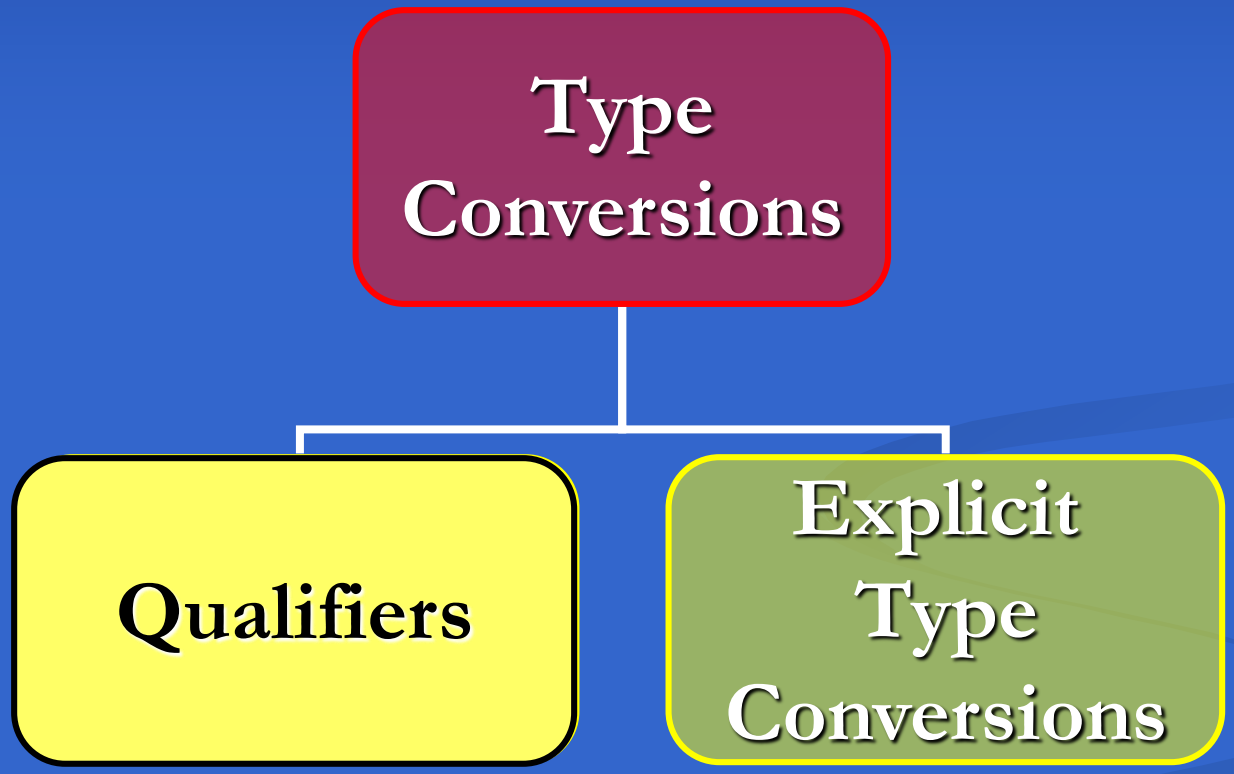
Type Conversions



Type Conversions



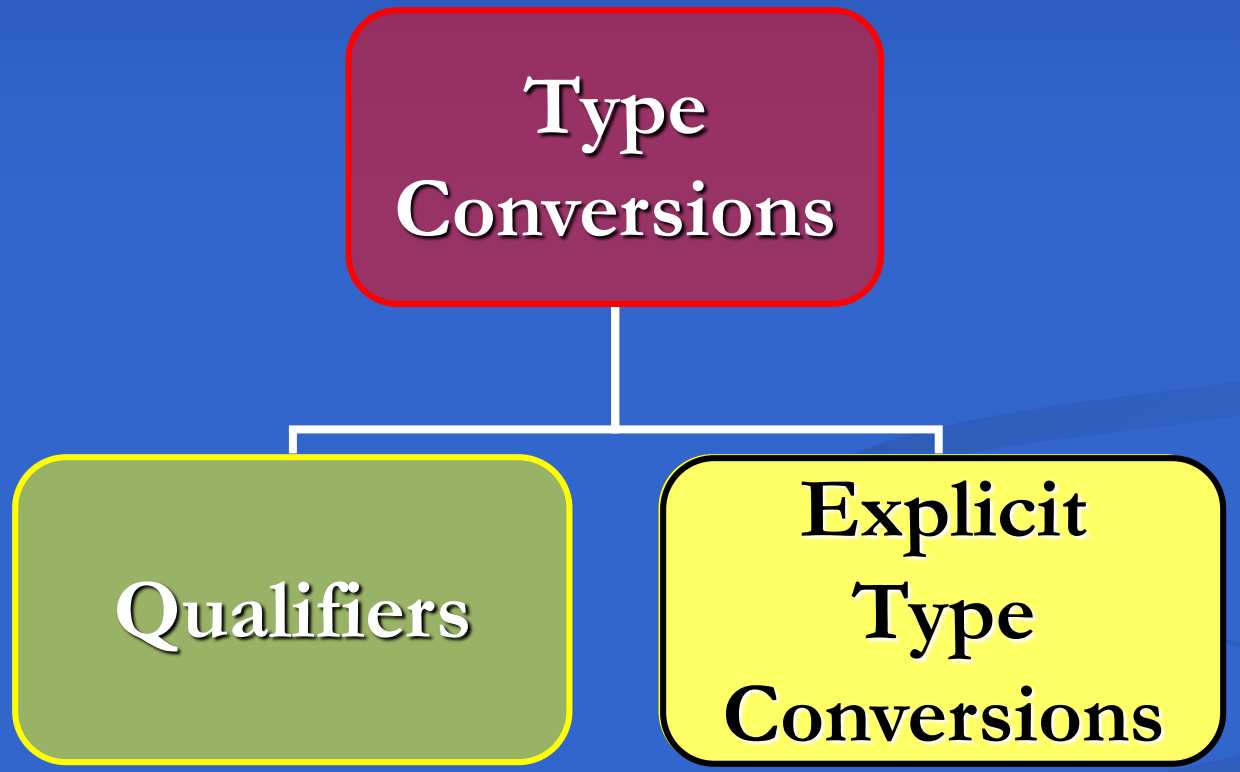
Qualifiers



Qualifiers

```
SA8: (s_byte(0), s_byte(1), s_byte(2), s_byte(3)) <= s_byte (5 DOWNTO 2);

SA9: (s_byte(0), s_byte(1), s_byte(2), s_byte(3)) <= (OTHER => 'X');

SA9: (s_byte(0), s_byte(1), s_byte(2), s_byte(3)) <= v41_byte'(OTHER => 'X');
```

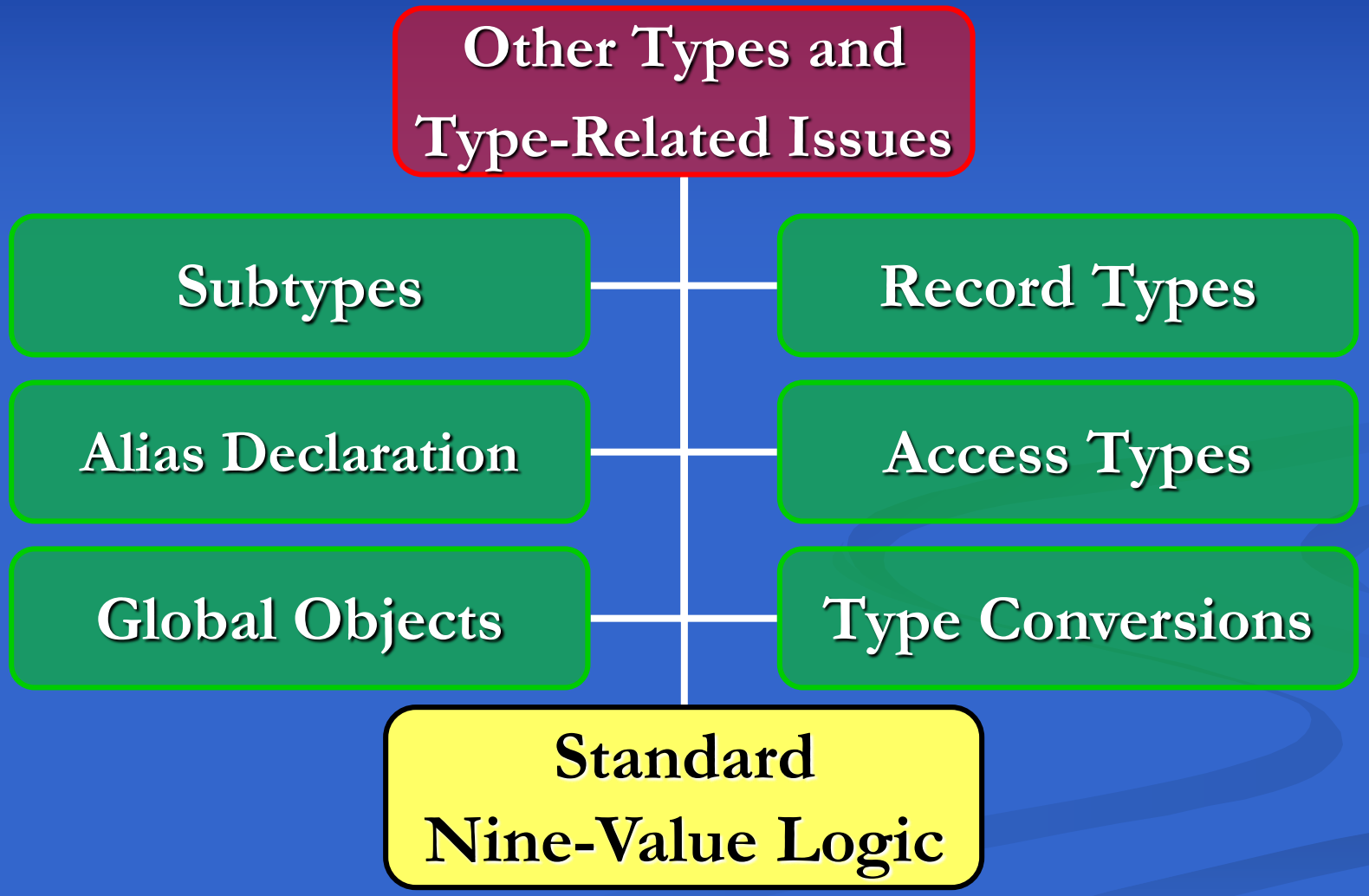
Explicit Type Conversions



Explicit Type Conversions

```
TYPE v41_byte IS ARRAY (7 DOWNTO 0) of v41;
TYPE v41_octal IS ARRAY (7 DOWNTO 0) of v41;
   . . . . . .
SIGNAL sb : v41_byte;
SIGNAL so : v41_octal;

sb <= so;    -- NOT ALLOWED

sb <= v41_byte (so);    -- Explicit Type Conversion
so <= v41_octal (sb);
```

Standard Nine-Value Logic

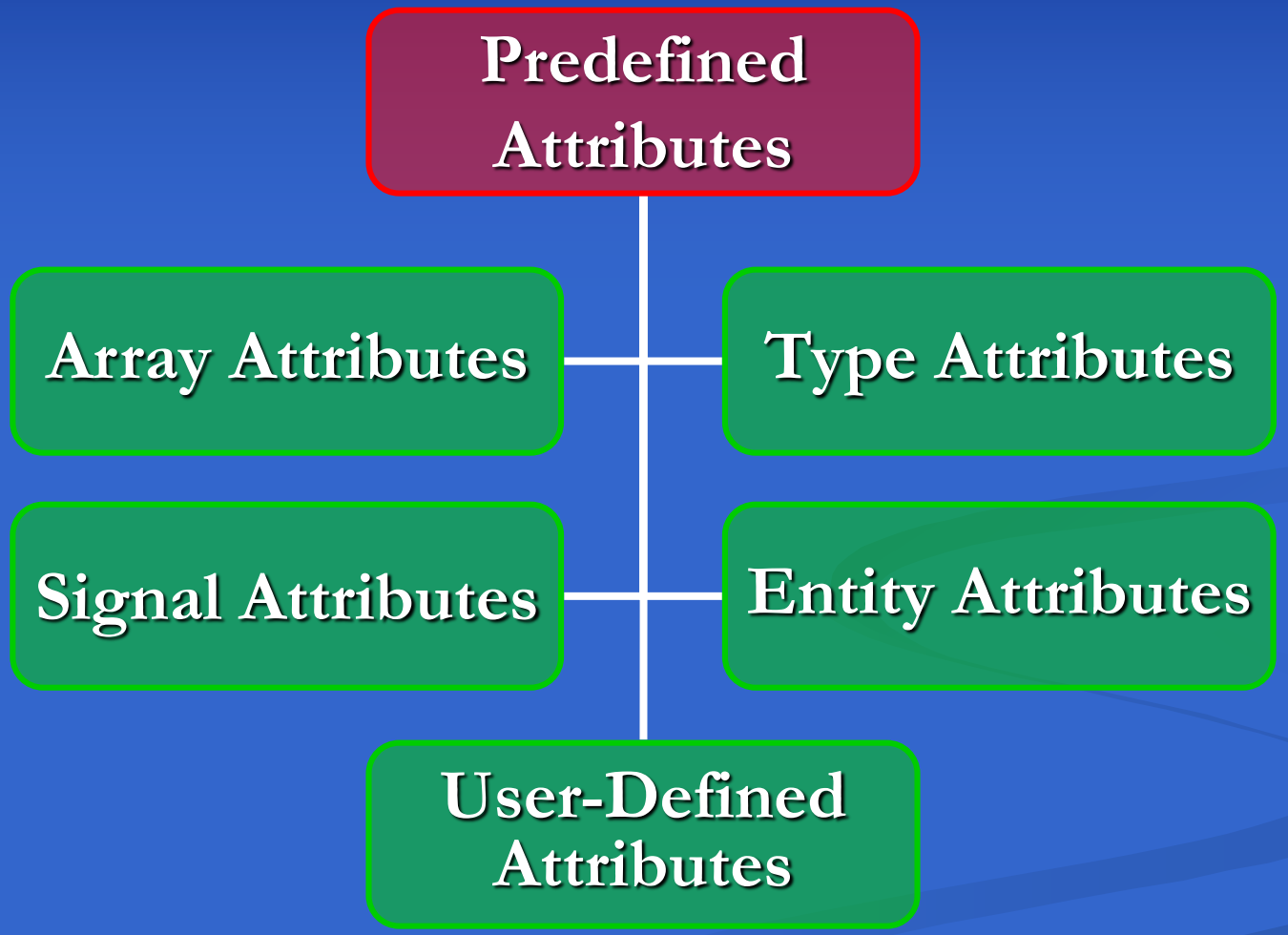


Standard Nine-Value Logic

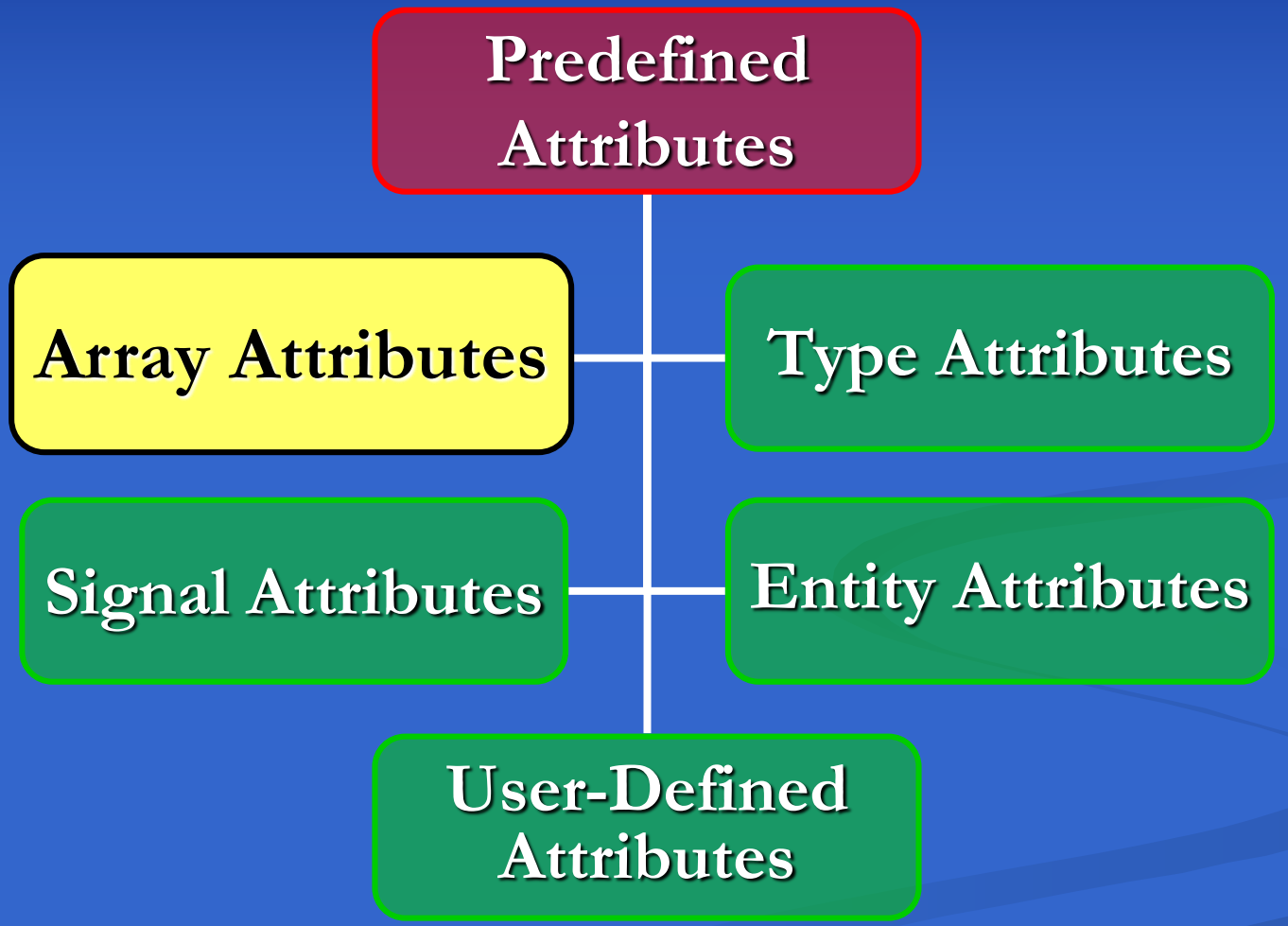
TYPE					
X01	‘X’,	‘0’,	‘1’		
X01Z	‘X’,	‘0’,	‘1’,	‘Z’	
UX01	‘U’,	‘X’,	‘0’,	‘1’	
UX01Z	‘U’,	‘X’,	‘0’,	‘1’,	‘Z’

- *std_logic* Sub-types

Predefined Attributes



Array Attributes

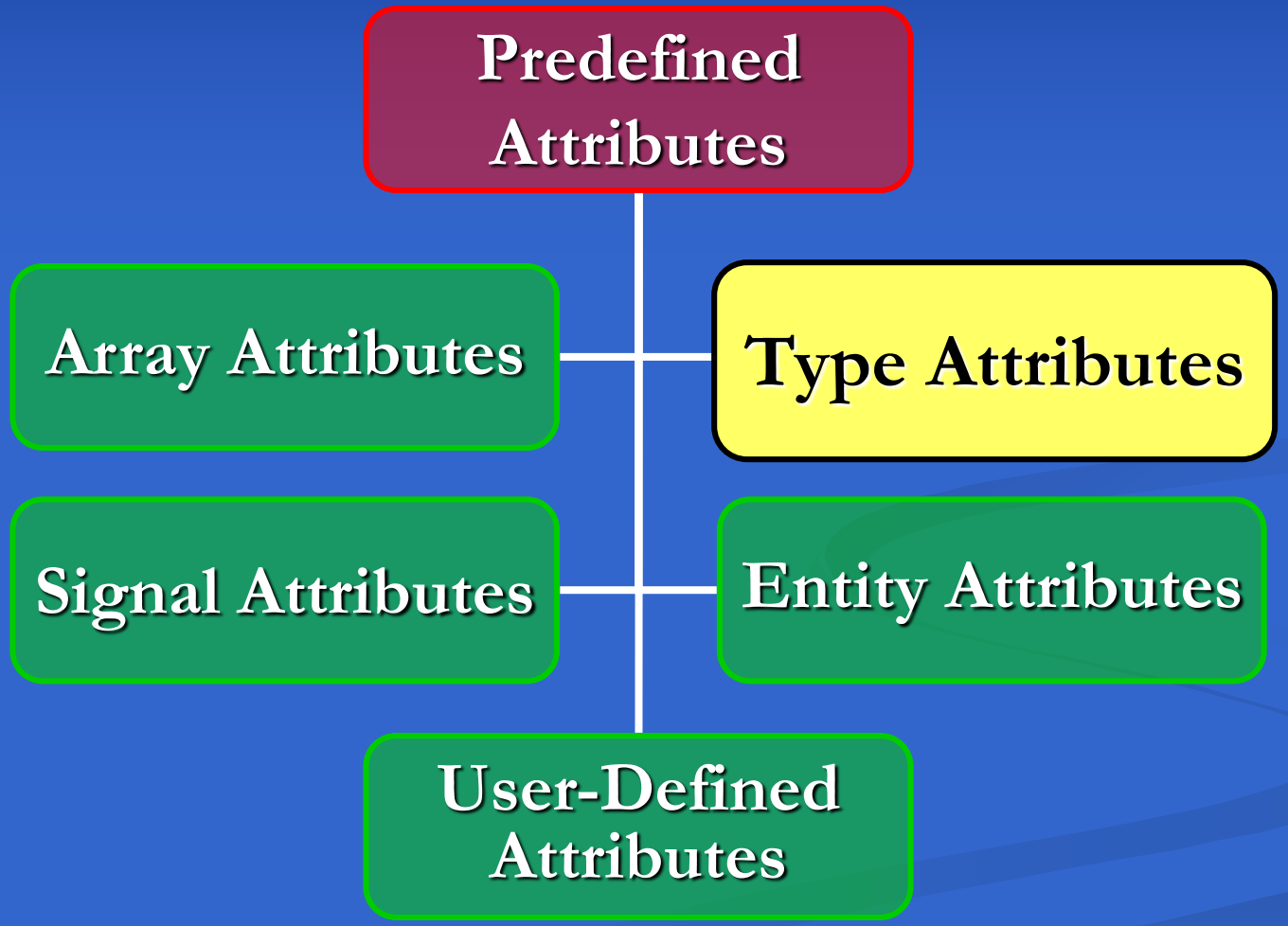


Array Attributes

Attribute	Description	Example	Result
‘LEFT	Left bound	s_4by8 ‘LEFT	3
‘RIGHT	Right bound	s_4by8 ‘RIGHT s_4by8 ‘RIGHT(2)	0 7
‘HIGH	Upper bound	s_4by8 ‘HIGH(2)	7
‘LOW	Lower bound	s_4by8 ‘LOW(2)	0
‘RANGE	Range	s_4by8 ‘RANGE(2) s_4by8 ‘RANGE(1)	0 TO 7 3 DOWNTO 0
‘REVERSE_RANGE	Reverse range	s_4by8 ‘REVERSE_RANGE(2) s_4by8 ‘REVERSE_RANGE(1)	7 DOWNTO 0 0 TO 3
‘LENGTH	Length	s_4by8 ‘LENGTH	4
‘ASCENDING	TRUE If Ascending	S_4by8 ‘ASCENDING(2) s_4by8 ‘ASCENDING(1)	TRUE FALSE

- Predefined Array Attributes

Type Attributes



Type Attributes

Attribute	Description	Example	Result
‘BASE	Base of type	v3l’BASE	v4l
‘LEFT	Left bound of type or subtype	v3l’LEFT v4l’LEFT	‘0’ ‘X’
‘RIGHT	Right bound of type or subtype	v3l’RIGHT v4l’RIGHT	‘Z’ ‘Z’
‘HIGH	Upper bound of type or subtype	INTEGER’HIGH v3l’HIGH	Large ‘Z’
‘LOW	Lower bound of type or subtype	POSITIVE’LOW v4l’LOW	1 ‘X’
‘POS(V)	Position of value V in *base* of type.	v4l’POS(‘Z’) v3l’POS(‘X’)	3 0
‘VAL(P)	Value at Position P in *base* of type.	v4l’VAL(3) v3l’VAL(3)	‘Z’ ‘Z’

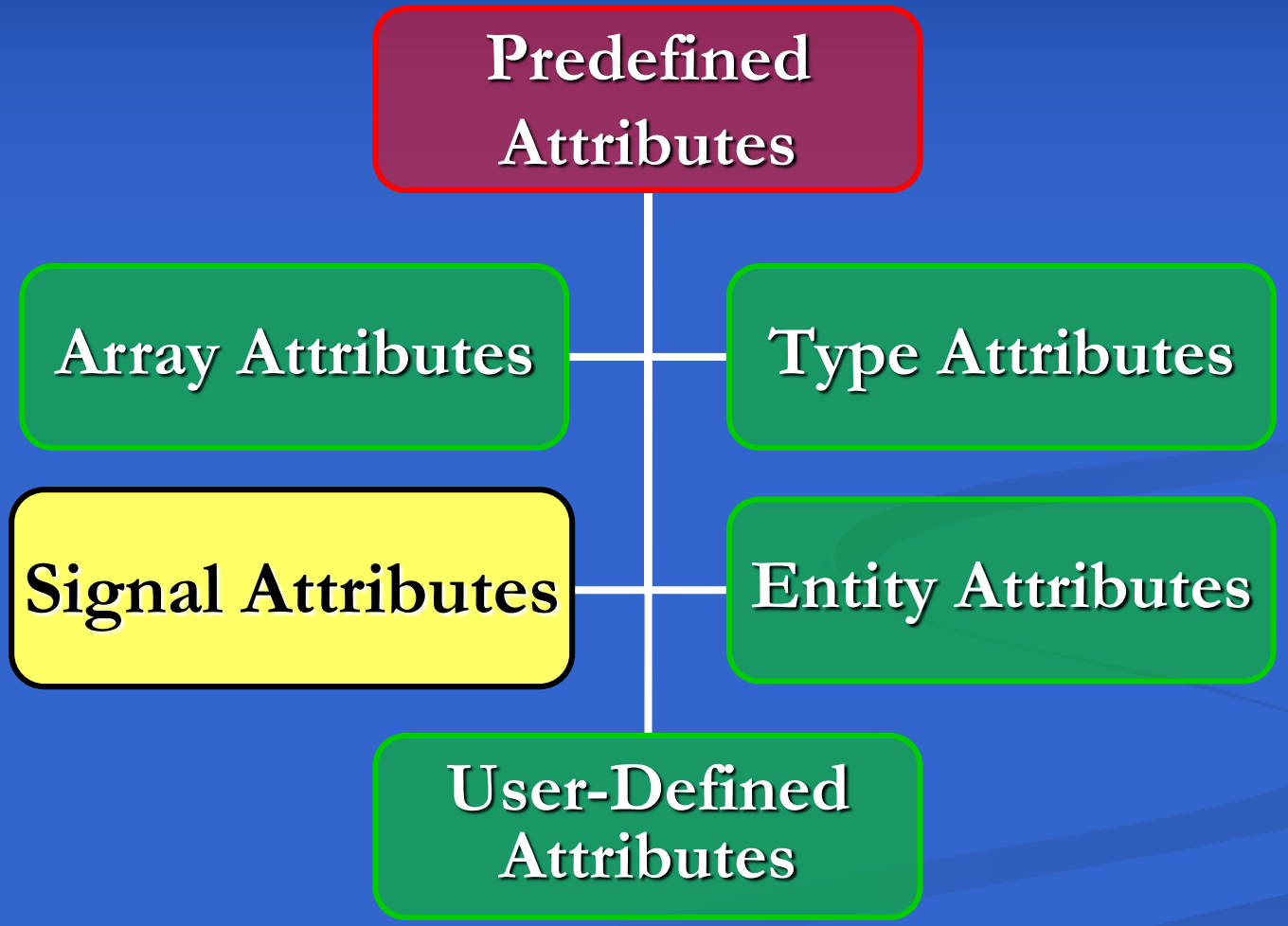
- Predefined Type Attributes

Type Attributes

Attribute	Description	Example	Result
‘SUCC(V)	Value, after value V in *base* of type.	v3l’SUCC(‘1’)	‘Z’
‘PRED(V)	Value, before value V in base of type.	v3l’PRED(‘1’)	‘0’
‘LEFTOF(V)	Value, left of value V in *base* of type.	v3l’LEFTOF(‘1’) v3l’LEFTOF(‘X’)	‘0’ Error
‘RIGHTOF(V)	Value, right of value V in *base* of type.	v3l’RIGHTOF(‘1’) v3l’RIGHTOF(‘X’)	‘Z’ ‘0’
‘ASCENDING	TRUE if range is ascending	v4l’ASCENDING	TRUE
‘IMAGE (V)	Converts value V of type to string.	v4l’IMAGE(‘Z’) opcode’IMAGE(lda)	“Z” “lda”
‘VALUE(S)	Converts string S to value of type.	opcode’VALUE(“nop”)	nop

- Predefined Type Attributes (Continued)

Signal Attributes



Signal Attributes

Attribute	T/E	Example	Kind	Type
Attribute description for the specified example				
‘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 no parameter or 0, the resulting signal is TRUE if *s1* has not changed in the current simulation time.				

- Predefined Signal Attributes. Signal *s1* is of Type BIT

Signal Attributes

Attribute	T/E	Example	Kind	Type
Attribute description for the specified example				
‘EVENT	EV	s1’EVENT	VALUE	BOOLEAN
In a simulation cycle, if s1 changes, this attribute becomes TRUE.				
‘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_VALUE is 0.				
‘LAST_VALUE	EV	s1’LAST_VALUE	VALUE	As s1
The value of s1 before the most recent event occurred on this signal.				

- Predefined Signal Attributes. Signal *s1* is of Type BIT (Continued)

Signal Attributes

Attribute	T/E	Example	Kind	Type
Attribute description for the specified example				
'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 or 0, the 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 delta time.				
'LAST_ACTIVE	TR	s1'LAST_ACTIVE	VALUE	TIME
The amount of time since the last transaction occurred on *s1*. If *s1*'ACTIVE is TRUE, *s1*'LAST_ACTIVE is 0.				

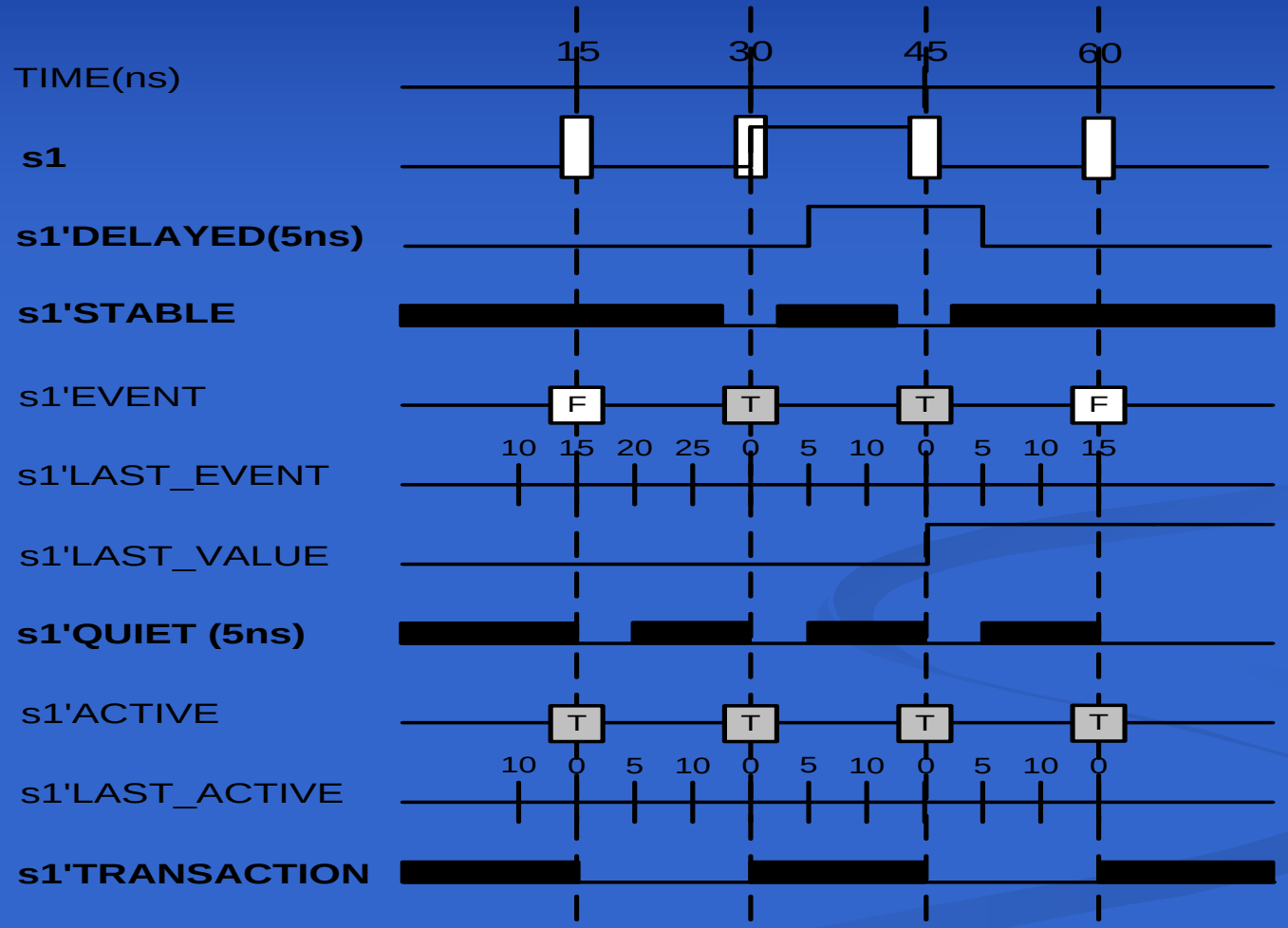
- Predefined Signal Attributes. Signal *s1* is of Type BIT (Continued)

Signal Attributes

Attribute	T/E	Example	Kind	Type
Attribute description for the specified example				
‘TRANSACTION	TR	s1’TRANSACTION	SIGNAL	BIT
A signal that toggles each time a transaction occurs on *s1*. Initial value of this attribute is not defined.				
‘DRIVING	-	s1’DRIVING	VALUE	BOOLEAN
If *s1* is being driven in a process, *s1*’DRIVING is TRUE in the same process.				
‘DRIVING_VALUE	-	s1’DRIVING_VALUE	VALUE	As *s1*
The driving value of *s1* from within the process this attribute is being applied.				

- Predefined Signal Attributes. Signal *s1* is of Type BIT (Continued)

Signal Attributes



- **Results of Signal Attributes when Applied to the BIT Type Signal, *s1***

Signal Attributes

```
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
   q <= d WHEN (c = '0' AND c'EVENT);
END falling_edge;
```

- A Simple Falling Edge Flip-Flop Using Signal Attributes

Signal Attributes

```
FF: BLOCK (c = '0' AND NOT c'STABLE) BEGIN
    qf <= GUARDED din;
END BLOCK FF;
--
LT: BLOCK (c = '0' AND c'EVENT) BEGIN
    ql <= GUARDED din;
END BLOCK LT;
```

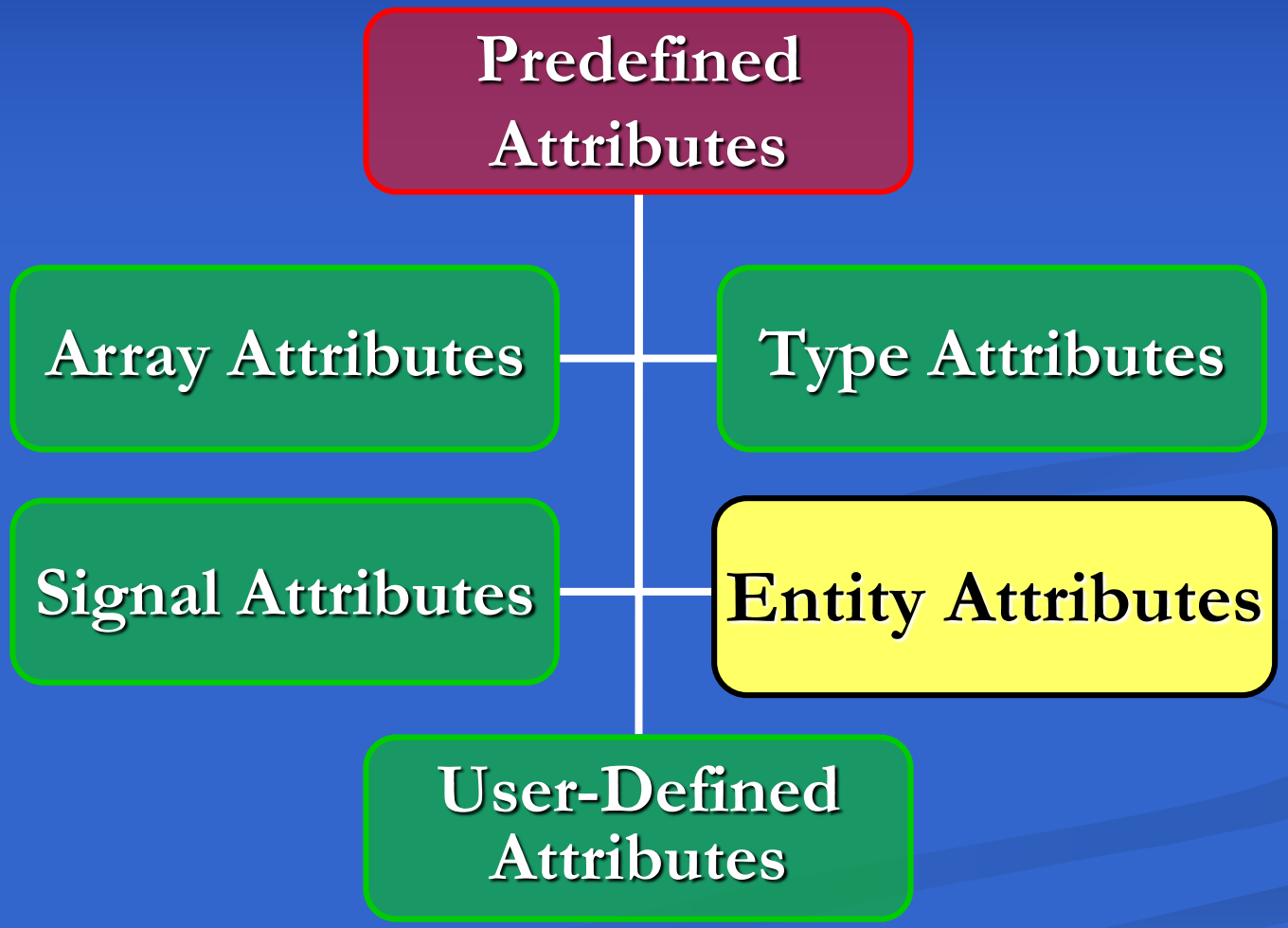
- A Simple Falling Edge Flip-Flop Using Signal Attributes

Signal Attributes

```
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 t'EVENT) AND
                         (t'DELAYED'STABLE(20 NS))
                       ) ELSE tmp;
   q <= tmp AFTER 8 NS;
END toggle;
```

- A Simple Toggle Flip-Flop Using Signal Attributes

Entity Attributes



Entity Attributes

```
ENTITY multiplexer_n_tester IS END ENTITY;
--
ARCHITECTURE timed OF multiplexer_n_tester IS
   SIGNAL a : BIT_VECTOR(7 DOWNTO 0);
   SIGNAL s : BIT_VECTOR(2 DOWNTO 0);
   SIGNAL w1 : BIT;
   FOR UUT1: mux_n
      USE ENTITY
          components.multiplexer(customizable);
BEGIN
   UUT1: mux_n PORT MAP (a, s, w1);
   onehot_data (a, 123 NS, 9);
   consecutive_data (s, 79 NS, 11);
END ARCHITECTURE timed;
```

- Applying Entity Attributes

Entity Attributes

```
ENTITY multiplexer IS
    PORT (ins: IN BIT_VECTOR; s: IN BIT_VECTOR;
          w: OUT BIT);
END ENTITY multiplexer;
--
ARCHITECTURE customizable OF multiplexer IS BEGIN
    ASSERT FALSE
    REPORT customizable'SIMPLE_NAME SEVERITY NOTE;
    ASSERT FALSE
    REPORT customizable'PATH_NAME SEVERITY NOTE;
    ASSERT FALSE
    REPORT customizable'INSTANCE_NAME SEVERITY NOTE;
    w <= mux(ins, s);
END ARCHITECTURE customizable;
```

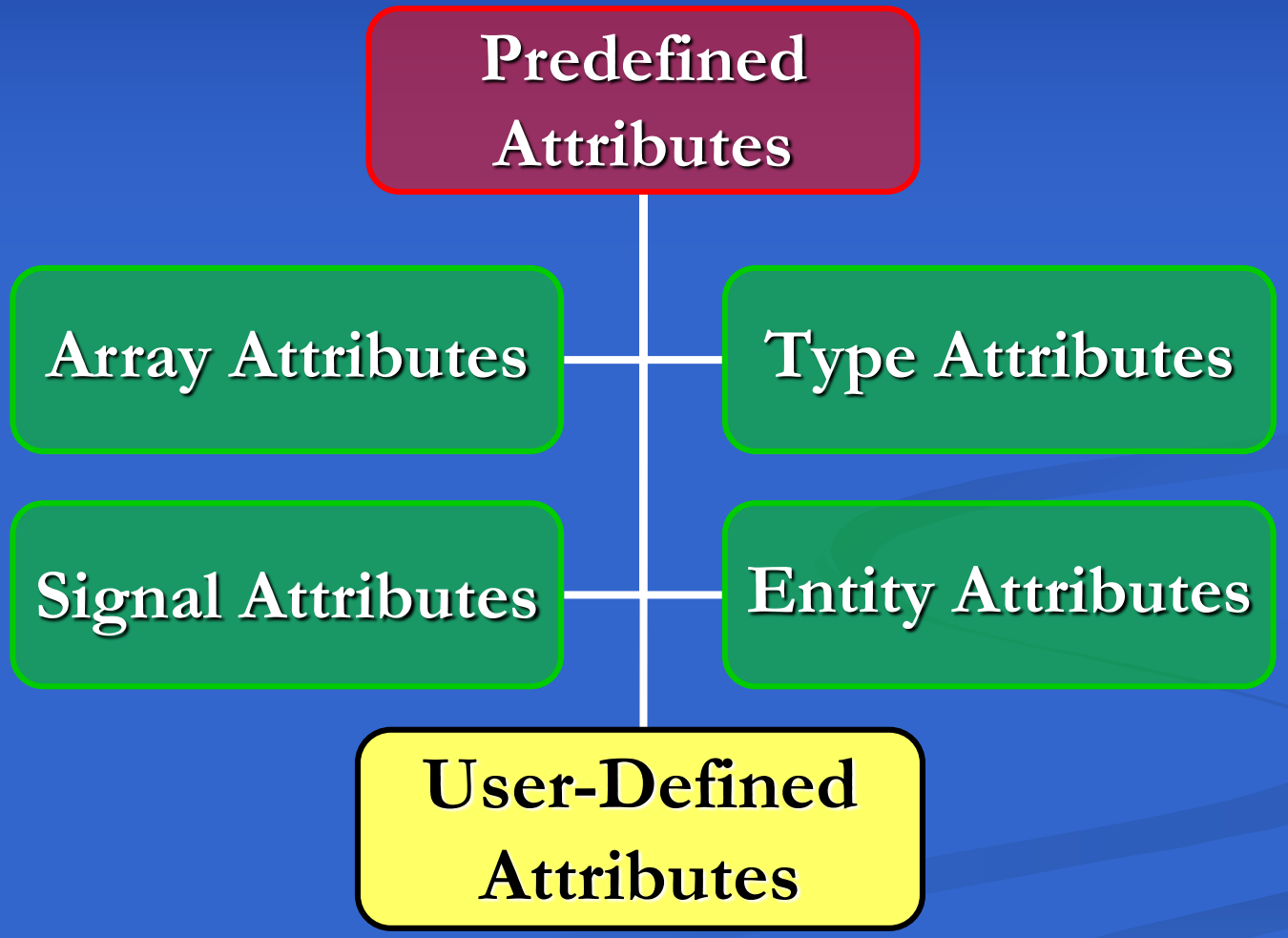
- Applying Entity Attributes (Continued)

Entity Attributes

```
--# ** Note: customizable
--# ** Note: :multiplexer_n_tester:uut1:
--# ** Note: :multiplexer_n_tester(timed)
             :uut1@multiplexer(customizable):
```

- Entity Attribute Examples

User-Defined Attributes



User-Defined Attributes

```
PACKAGE utility_attributes IS
   TYPE timing IS RECORD
      rise, fall : TIME;
   END RECORD;
   ATTRIBUTE delay : timing;
   ATTRIBUTE sub_dir : STRING;
END utility_attributes;
```

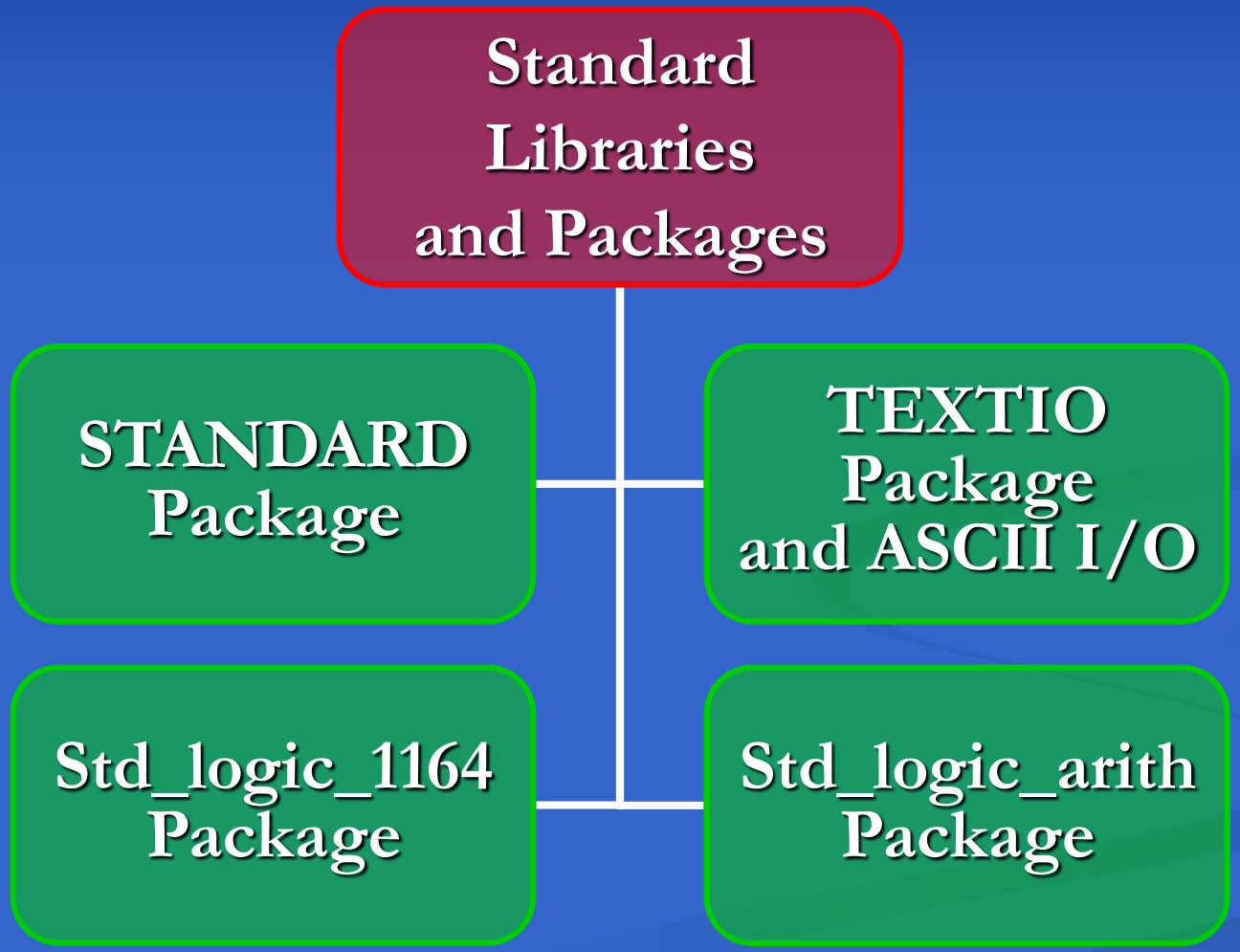
- Attribute Definitions

User-Defined Attributes

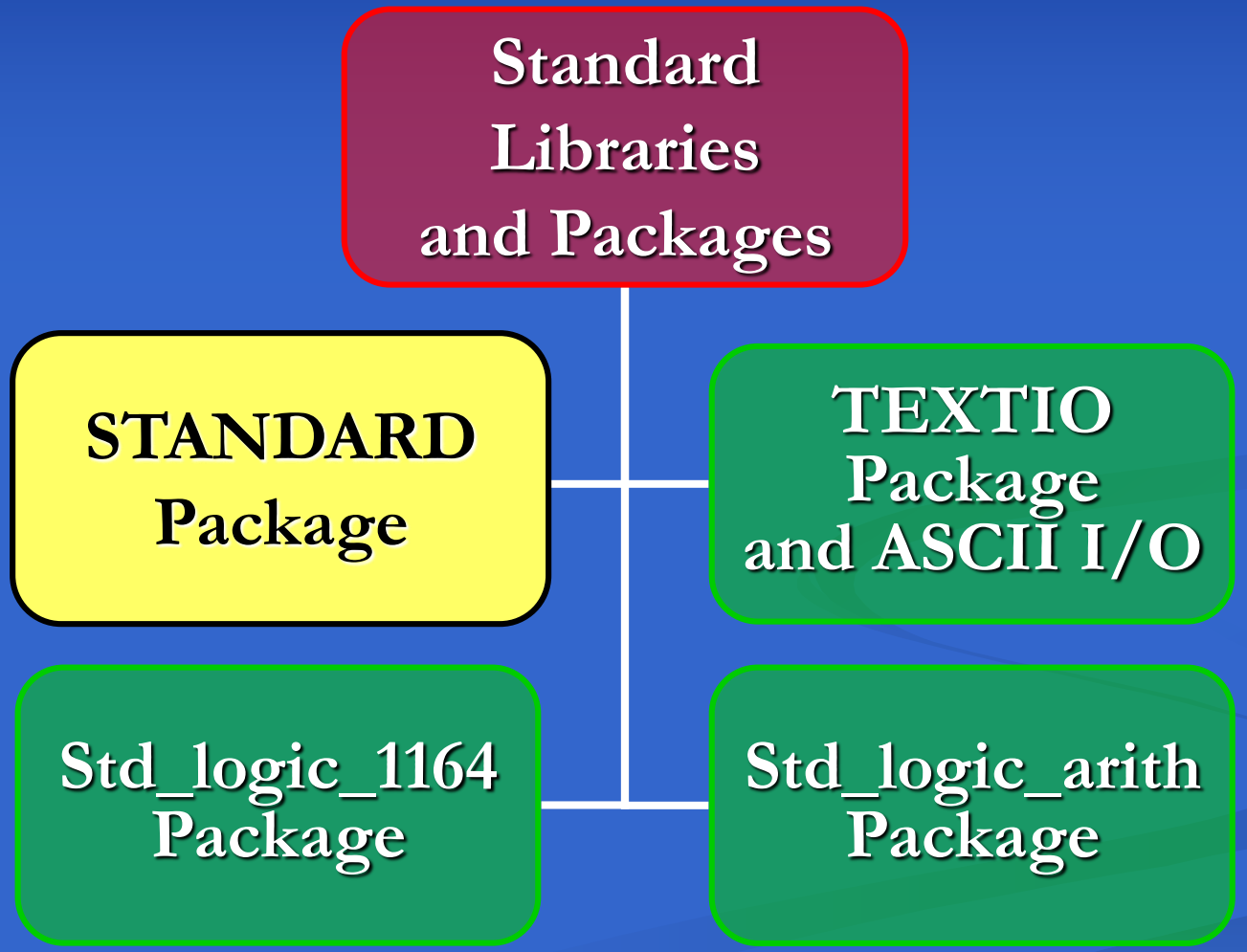
```
USE WORK.utility_attributes.ALL;
ENTITY multiplexer IS
   PORT (ins: IN BIT_VECTOR; s: IN BIT_VECTOR;
         w: OUT BIT);
   ATTRIBUTE sub_dir OF multiplexer :
             ENTITY IS "/user/vhdl";
   ATTRIBUTE delay OF w : SIGNAL IS (8 NS, 10 NS);
END ENTITY multiplexer;
--
ARCHITECTURE customizable OF multiplexer IS BEGIN
   w <= '1' AFTER w'delay.rise
         WHEN mux(ins, s) = '1'
         ELSE '0' AFTER w'delay.fall;
END ARCHITECTURE customizable;
```

- Using Attributes

Standard Libraries and Packages



STANDARD Package



STANDARD Package

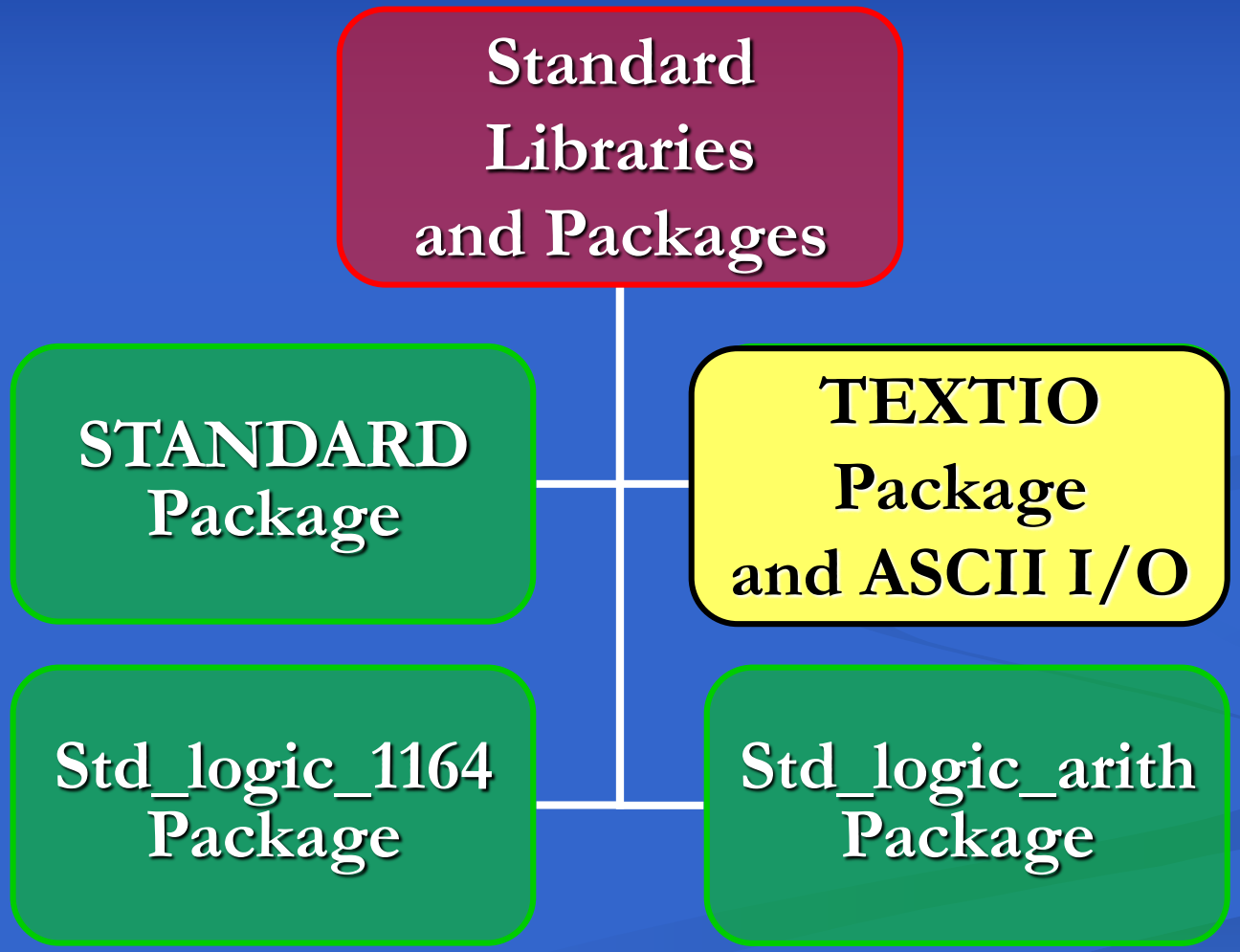
- The STANDARD package is in the STD library.
- An internal language package
- Does not exist as a VHDL code.

- Basic types such as BIT, BIT_VECTOR, INTEGER are included
- VHDL logical, relational, and arithmetic operations are over loaded for basic types of this package.
- Arithmetic operations are defined for INTEGER and REAL types
- Does not overload these operations for the BIT and BOOLEAN types

- Users can develop their own binary arithmetic functions.

- The standard numeric package (IEEE 1076.3) is the NUMERIC_BIT package that contains overloading of arithmetic operations for the BIT and BIT_VECTOR types.

TEXTIO Package and ASCII I/O



TEXTIO Package and ASCII I/O

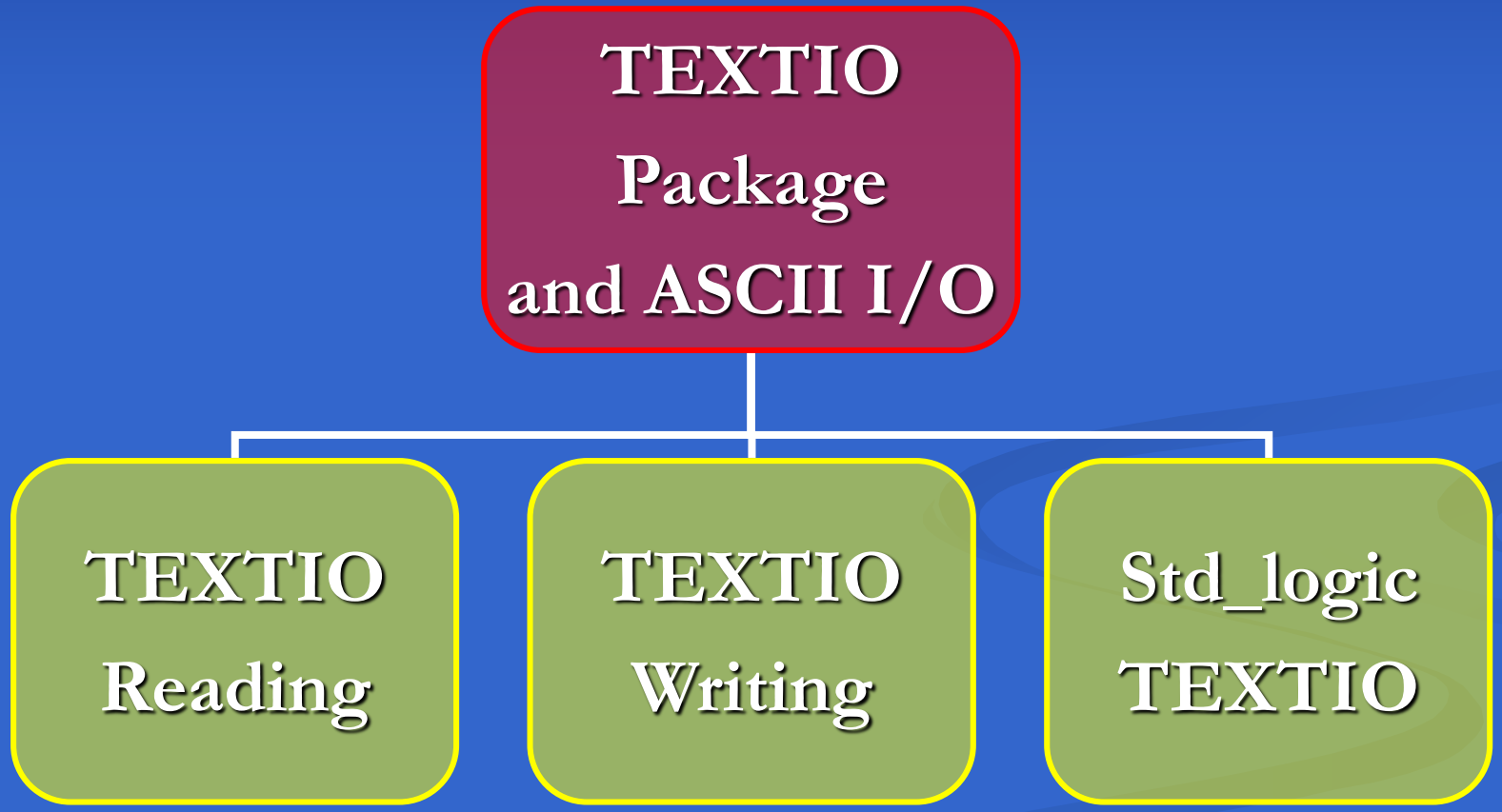
```
FILE f: TEXT;
FILE f: TEXT IS "input.txt";
FILE f: TEXT OPEN READ_MODE IS "input.txt";

FILE_OPEN (f, "input.txt", READ_MODE);
FILE_OPEN (f, "output.txt", WRITE_MODE);
FILE_OPEN (f, "output.txt", APPEND_MODE);

FILE_CLOSE (f);
```

```
VARIABLE l: LINE;
```

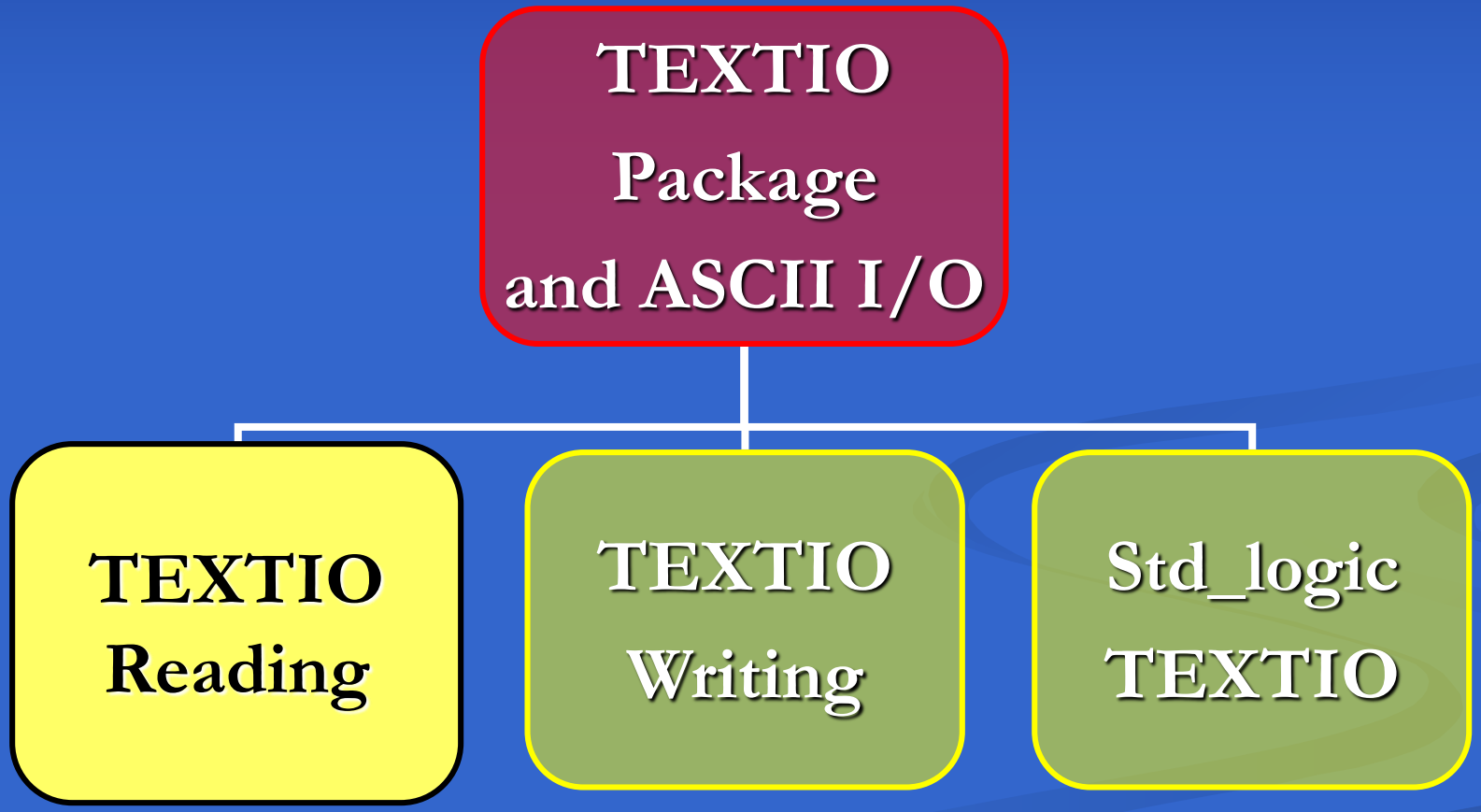
TEXTIO Package and ASCII I/O

```
READLINE (f, l) -- reads a line of file f and places
                    it in buffer l of type LINE
READ (l, v, …)  -- reads a value v of its type form l

WRITE (l, v, …) -- writes the value v to LINE l

WRIELINE (f, l) -- writes l to file f. Function

ENDFILE (f)     -- returns TRUE if the end of file f
                           is reached
```

TEXTIO Package and ASCII I/O



TEXTIO Reading



TEXTIO Reading

```
PROCEDURE GetData
    (SIGNAL s : OUT BIT_VECTOR; FILE f : TEXT)
IS
    VARIABLE lbuf : LINE;
    VARIABLE t : TIME;
    VARIABLE d : BIT_VECTOR (s'RANGE);
BEGIN
    WHILE NOT ENDFILE (f) LOOP
        READLINE (f, lbuf);
        READ (lbuf, t);
        READ (lbuf, d);
        s <= TRANSPORT d AFTER t;
    END LOOP;
    FILE_CLOSE (f);
END PROCEDURE GetData;
```

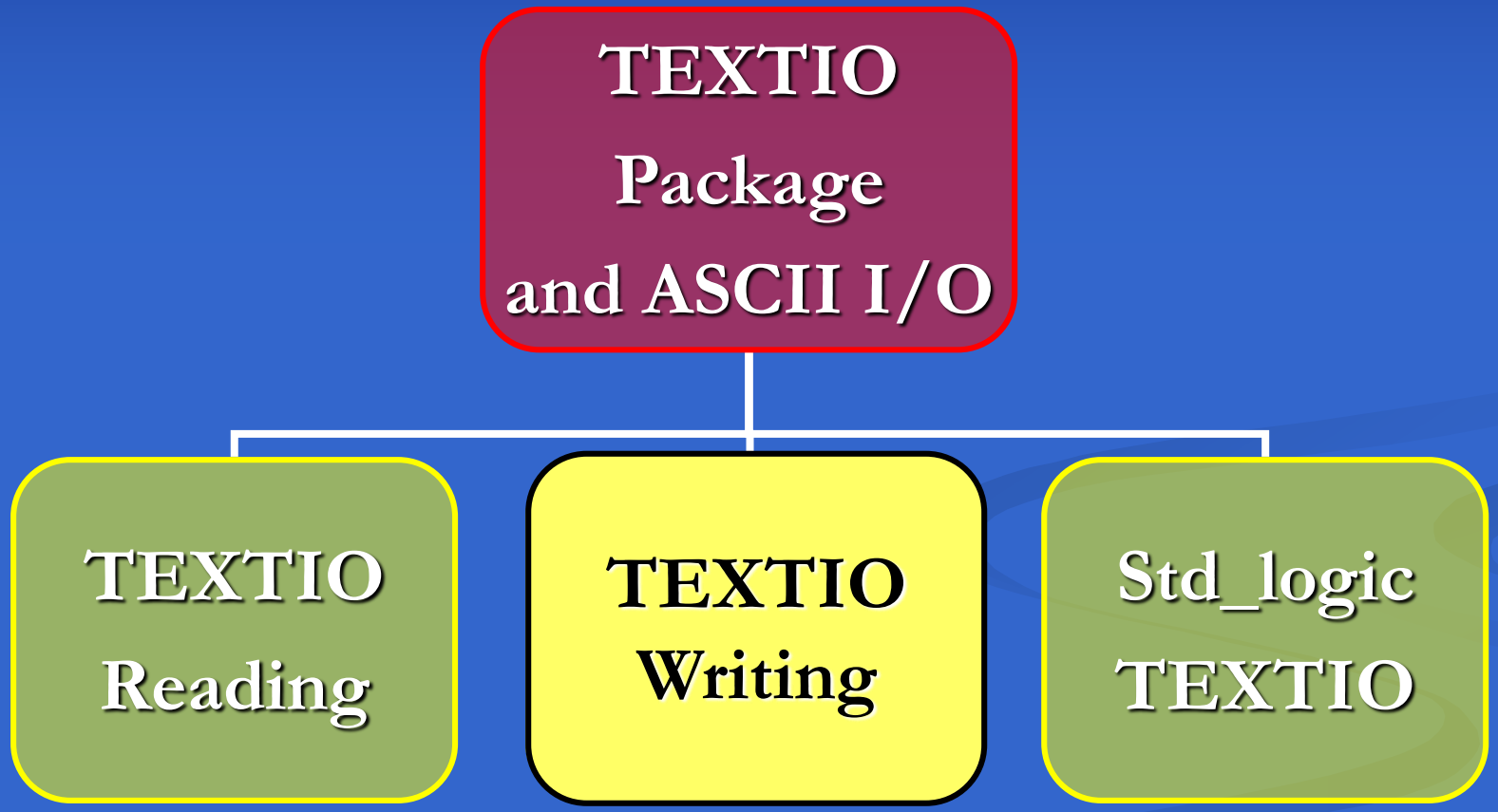
- Reading a TEXTIO File

TEXTIO Reading

```
  0 ns 00111000
 10 ns 00101111
 35 ns 10110000
 45 ns 11101010
 50 ns 01100001
 55 ns 00101110
 95 ns 11100011
110 ns 00011100
```

- Sample Data File

TEXTIO Writing



TEXTIO Writing

```
USE STD.TEXTIO.ALL;
ENTITY multiplexer8_tester IS END ENTITY;
--
ARCHITECTURE timed OF multiplexer8_tester IS
   SIGNAL a, b, w1 : BIT_VECTOR (7 DOWNTO 0);
   SIGNAL s : BIT := '0';
   FILE Ain : TEXT OPEN READ_MODE IS "Ain.dat";
   FILE Bin : TEXT OPEN READ_MODE IS "Bin.dat";
BEGIN
   UUT1: ENTITY WORK.multiplexer8 (conditional)
         PORT MAP (a, b, s, w1);
   ....................
   ....................
END ARCHITECTURE timed;
```

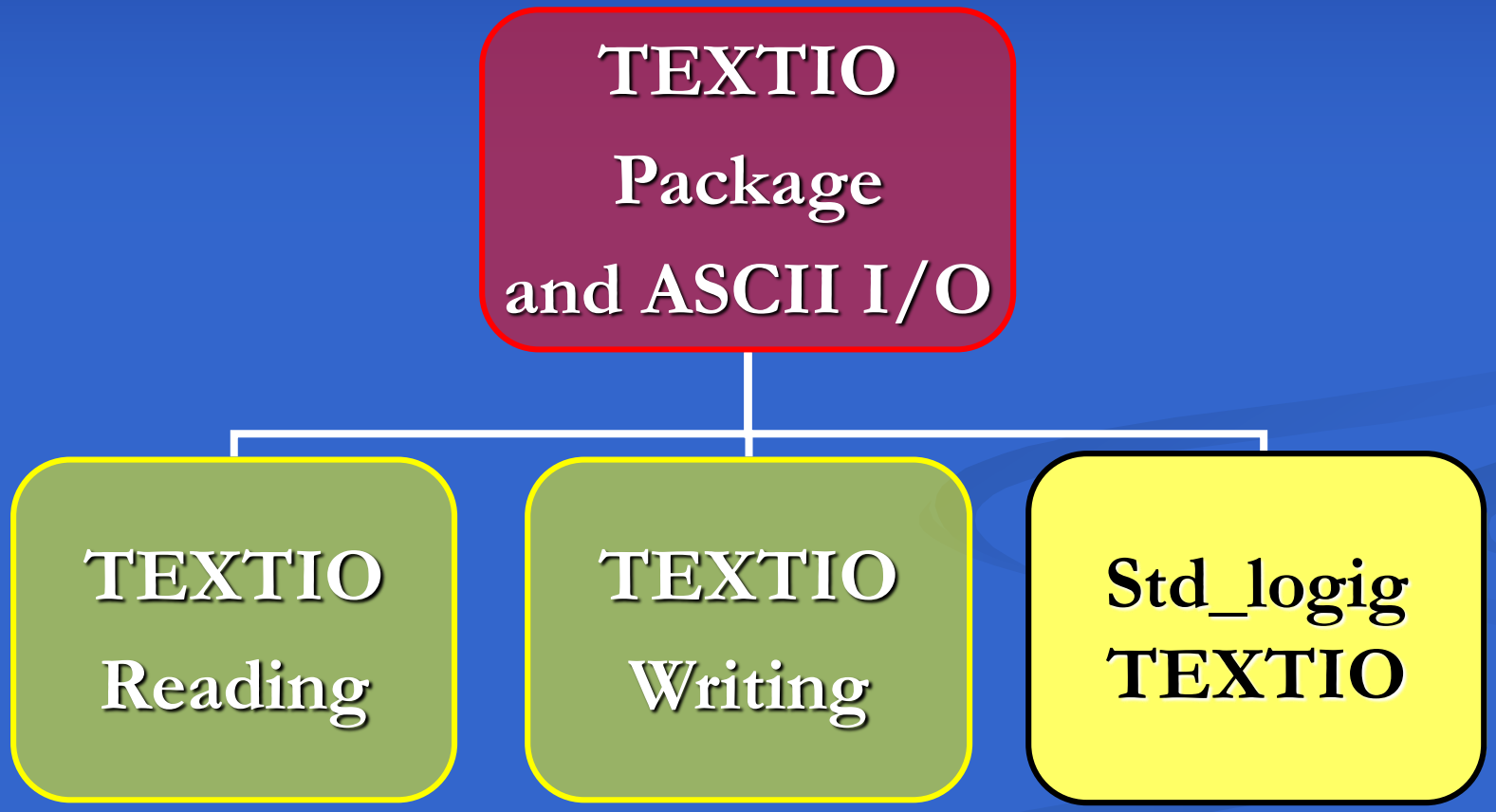
- Using Text Data for Input and Output

TEXTIO Writing

```
    ....................
    PROCESS (w1)
       FILE Wout : TEXT OPEN WRITE_MODE IS"Wout.dat";
       VARIABLE lbuf : LINE;
    BEGIN
       WRITE (lbuf, NOW, RIGHT, 8, NS);
       WRITE (lbuf, w1, RIGHT, 9);
       WRITELINE (Wout, lbuf);
    END PROCESS;
    GetData (a, Ain);
    GetData (b, Bin);
    s <= NOT s AFTER 25 NS WHEN NOW <= 140 NS
                           ELSE '0';
END ARCHITECTURE timed;
```

- Using Text Data for Input and Output (Continued)

Std_logic TEXTIO

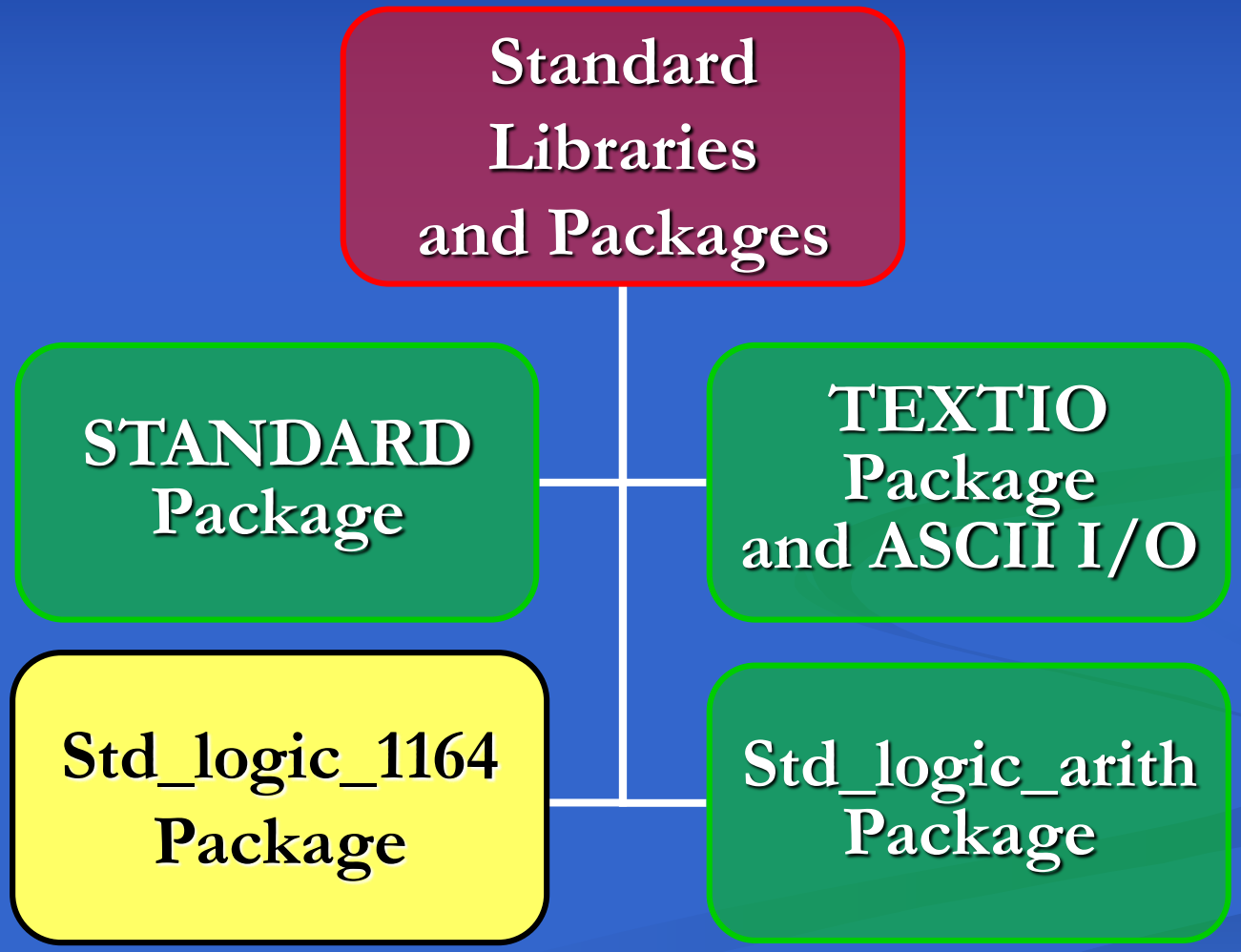


Std_logic TEXTIO

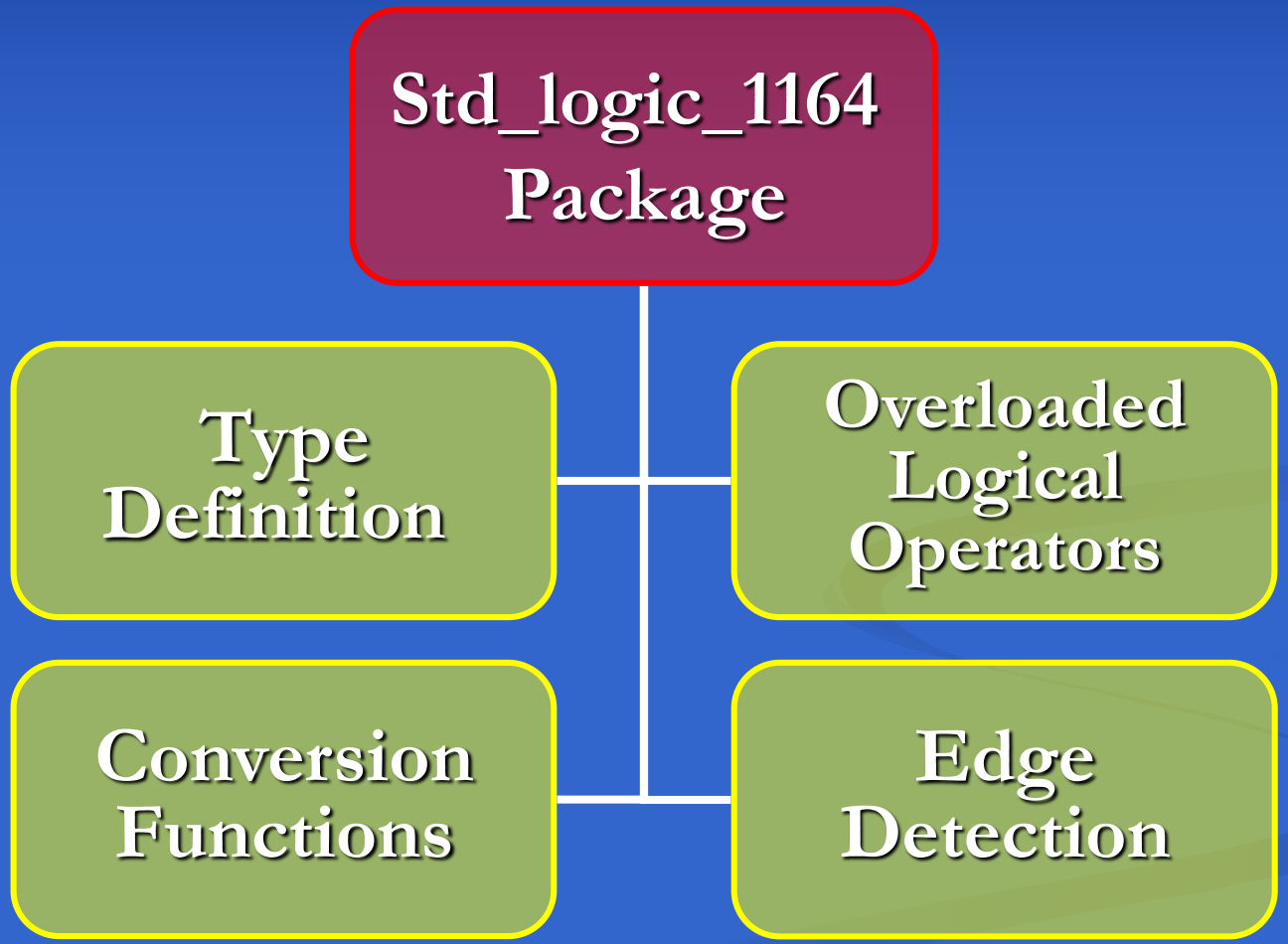
```
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE STD.TEXTIO.ALL;
USE IEEE.std_logic_TEXTIO.ALL;
```

- std_logic TEXTIO Package

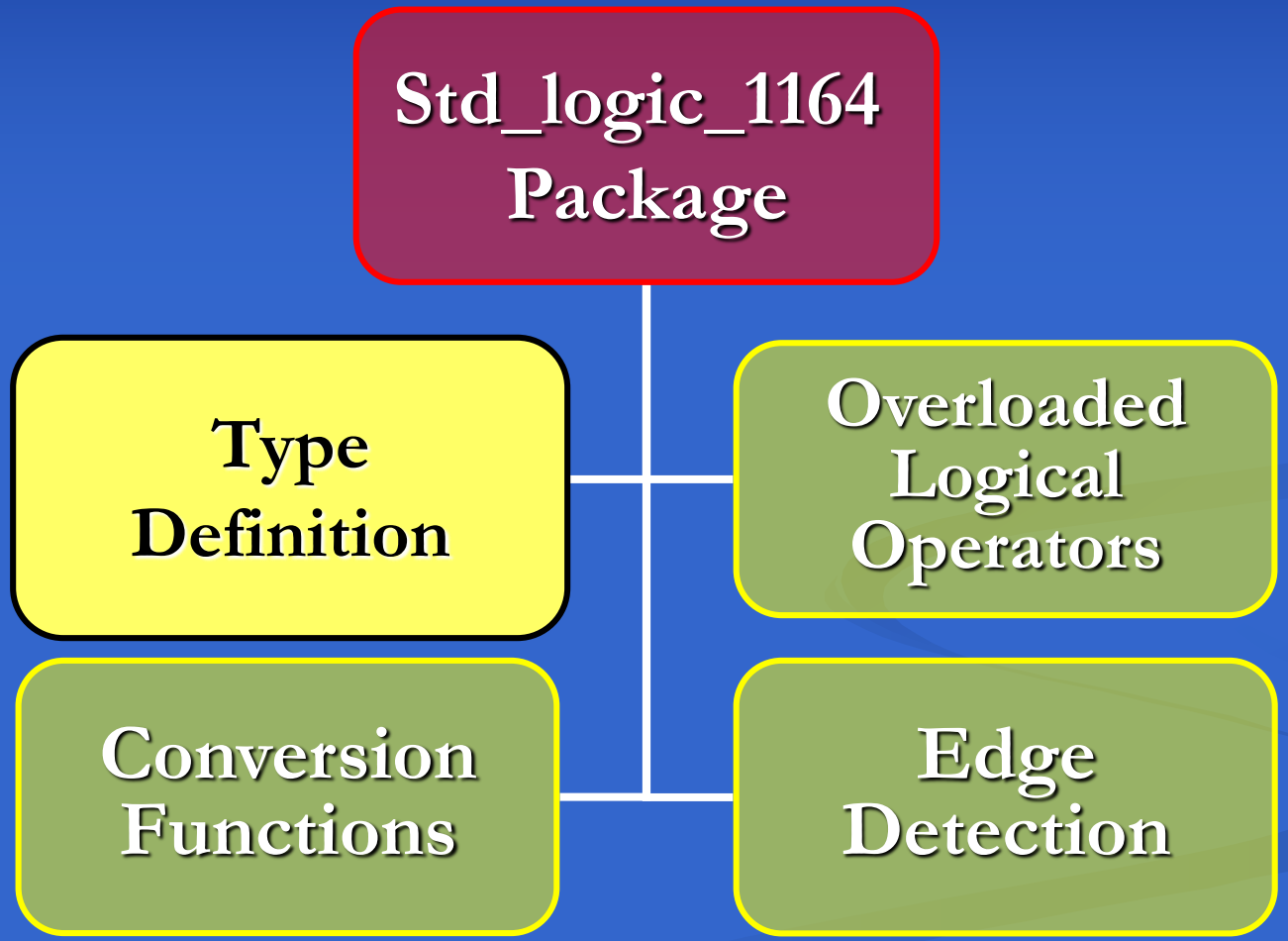
Std_logic_1164 Package



Std_logic_1164 Package



Type Definition



Type Definition

```
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
                   );
```

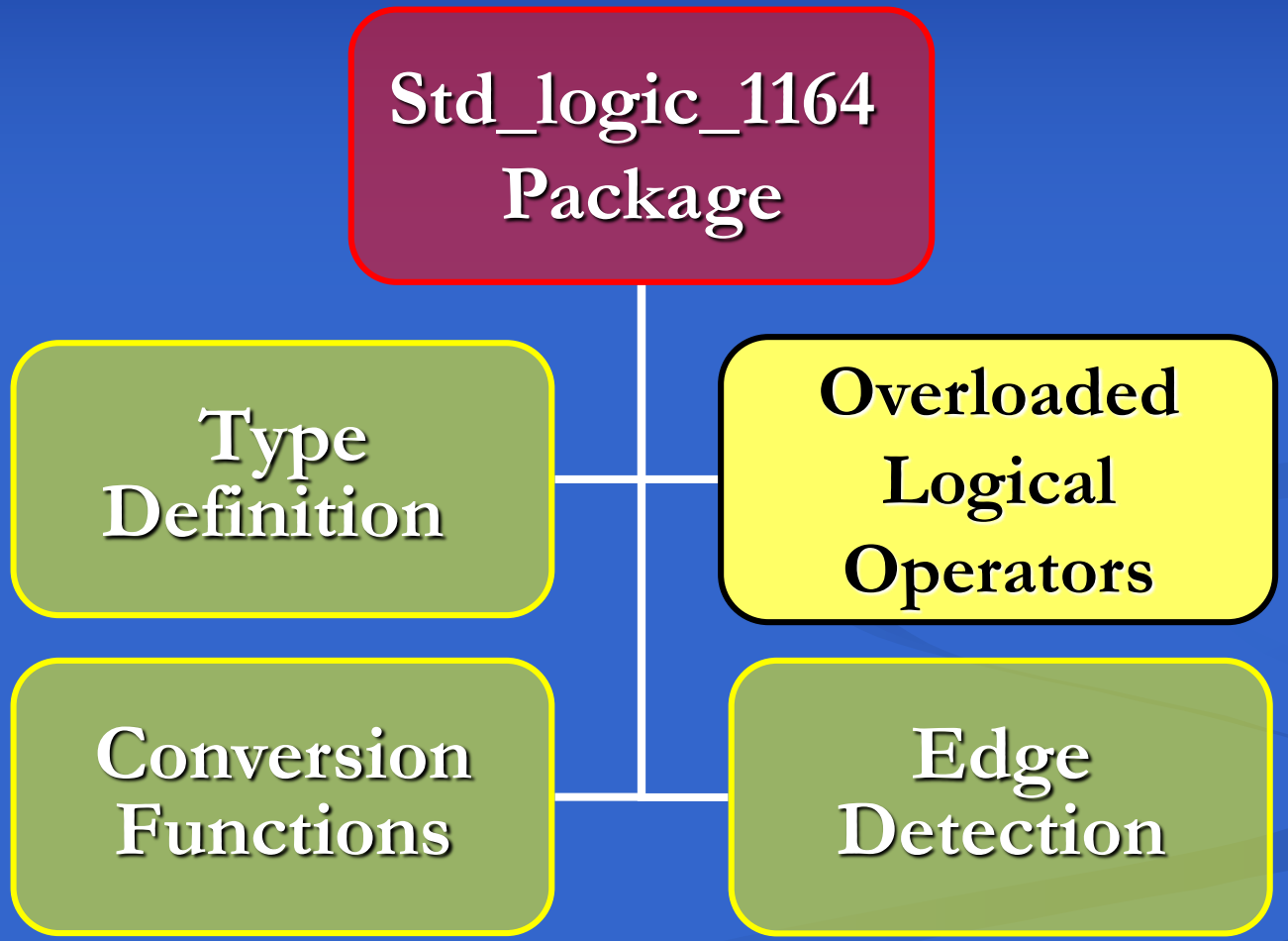
- *Std_logic* Enumeration Values

Type Definition

```
SUBTYPE X01 IS resolved std_ulogic
        RANGE 'X' TO '1'; -- ('X','0','1')
SUBTYPE X01Z IS resolved std_ulogic
        RANGE 'X' TO 'Z'; -- ('X','0','1','Z')
SUBTYPE UX01 IS resolved std_ulogic
        RANGE 'U' TO '1'; -- ('U','X','0','1')
SUBTYPE UX01Z IS resolved std_ulogic
        RANGE 'U' TO 'Z'; -- ('U','X','0','1','Z')
```

- Subtypes of the *std_logic* Type

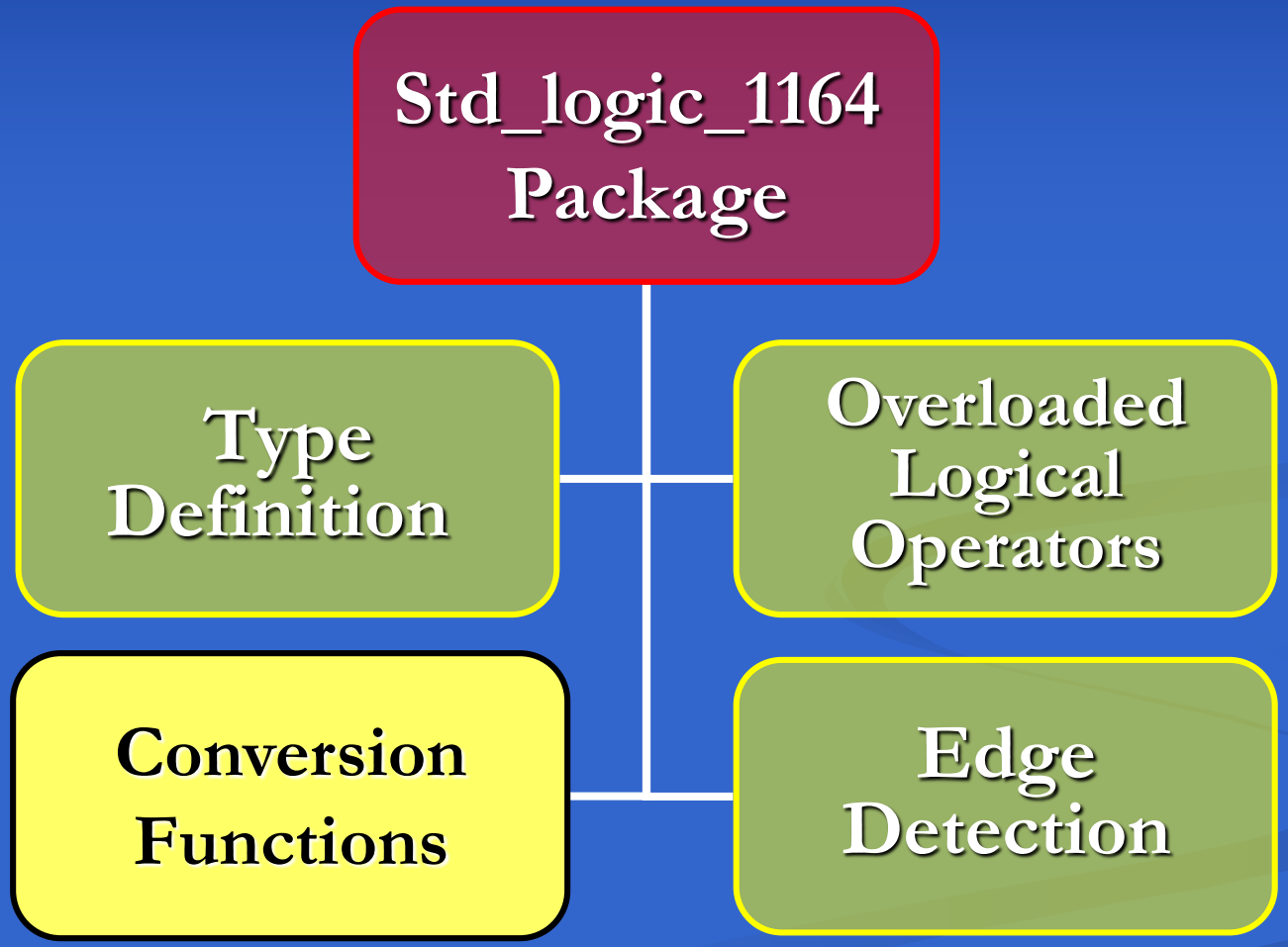
Overloaded Logical Operators



Overloaded Logical Operators

- Overloaded logic operators for
  - "AND", "NAND", "OR", "NOR", "XOR", "XNOR" and "NOT"
- Overloaded for *std_ulogic*, and because *std_logic* is considered a subtype of *std_ulogic*, they also work for the *std_logic* type.
- Also overloaded for *std_logic_vector* and *std_ulogic_vector* and combinations of the two arrays.

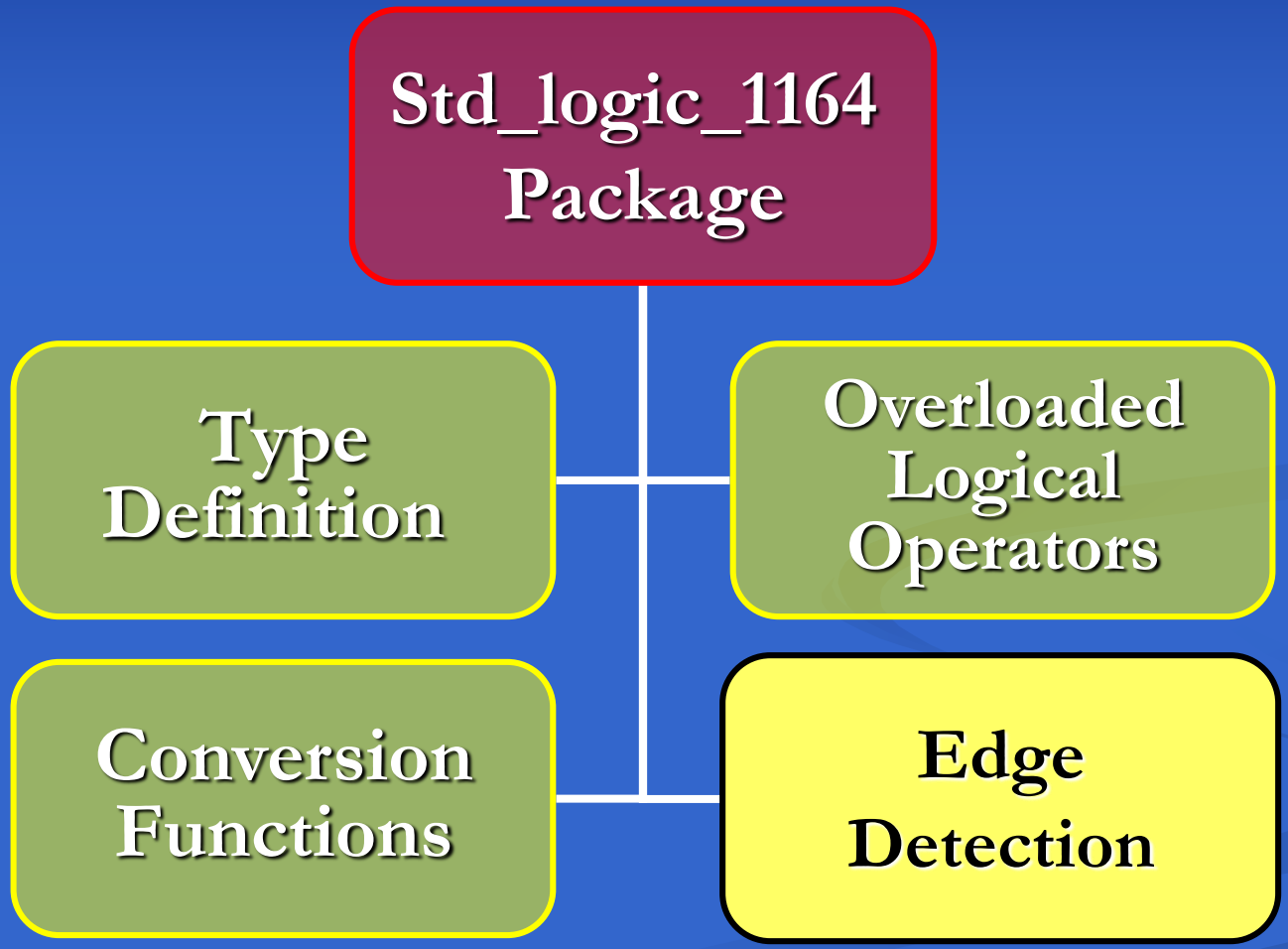
Conversion Functions



Conversion Functions

- Functions for conversions to and from **BIT** and ***std_logic*** and its subsets.
- ***To_StdLogicVector*** converts
  - ***BIT_VECTOR*** or ***std_ulogic_vector*** to ***std_logic_vector***.
- ***TO_X01*** that converts
  - BIT, ***std_logic***, ***std_ulogic*** and their vectorized versions to ***X01*** and ***std_logic_vector***.

Edge Detection

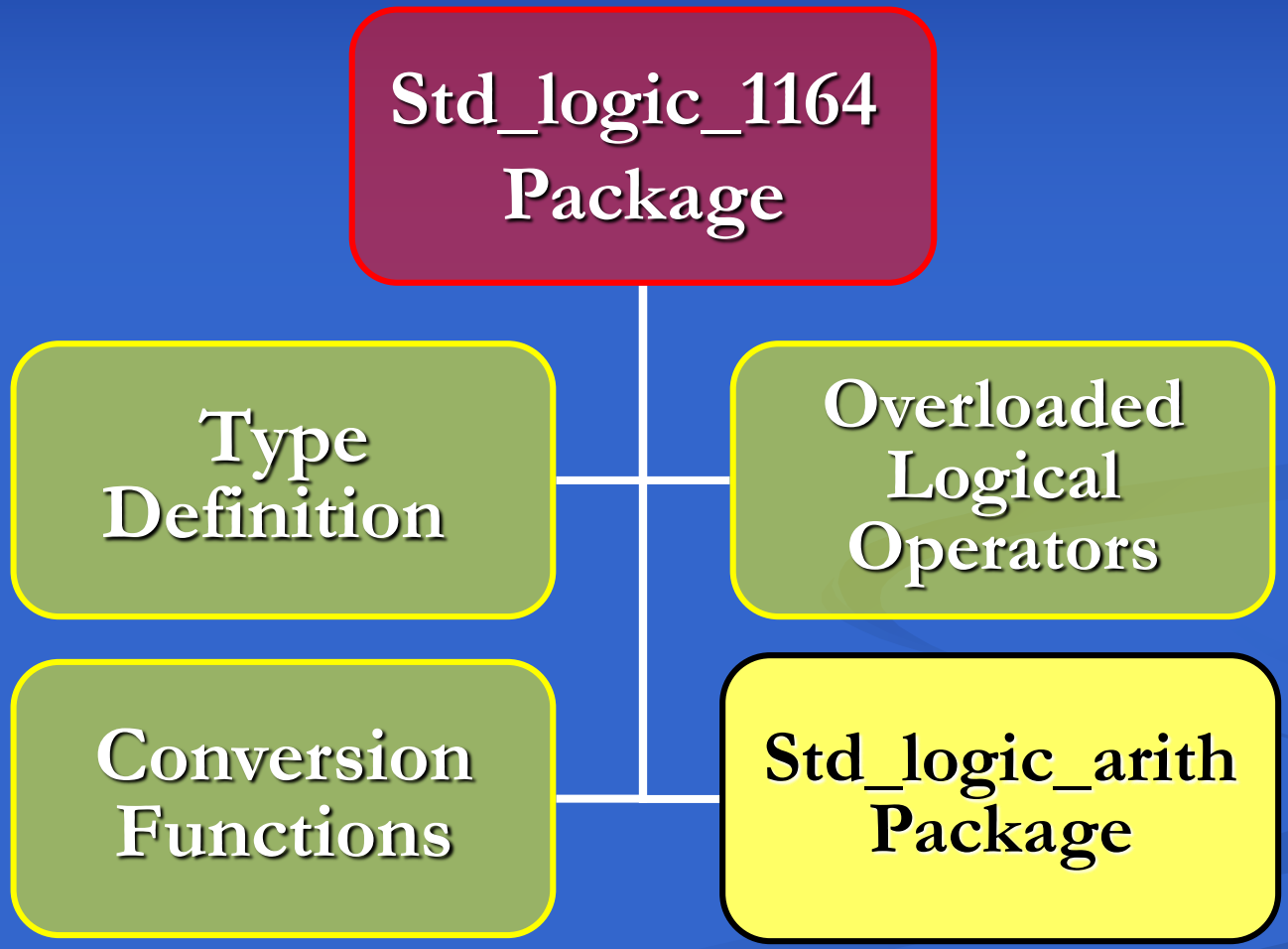


Edge Detection

- Edge detection functions
  - *rising_edge*
  - *falling_edge*

- Recognized by most synthesis tools for flip-flop clock edge detection.

Std_logic_arith Package



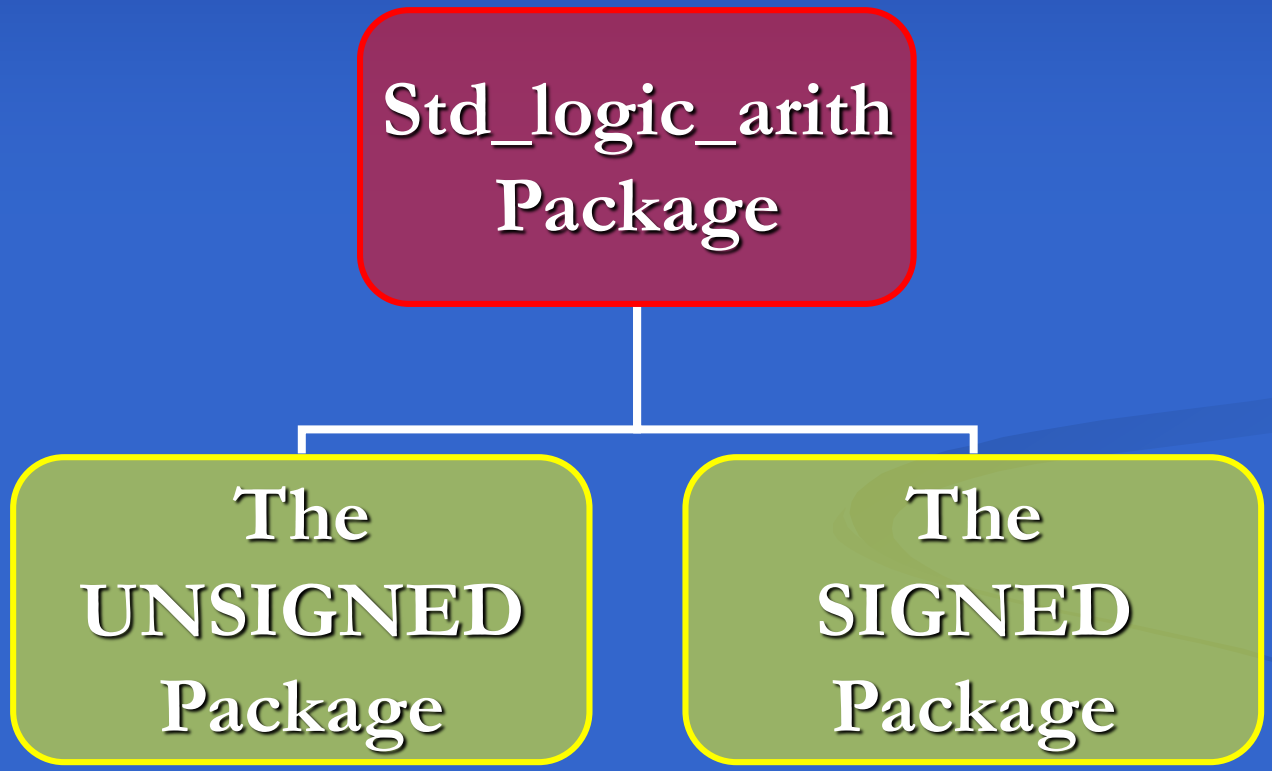
Std_logic_arith Package

- The IEEE standard arithmetic package
- An important package that eases the use of the VHDL language for arithmetic and logical functions

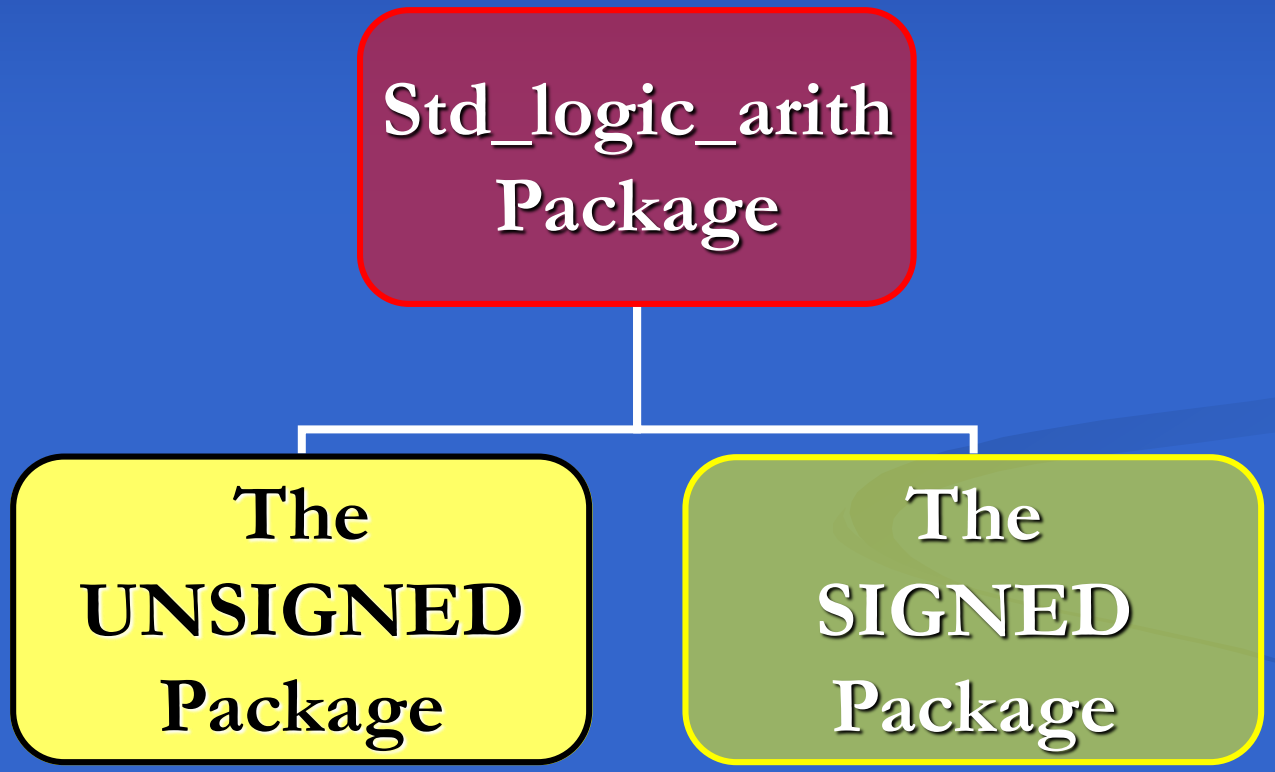
- The *std_logic_arith*
  - Defines ***SIGNED*** and ***UNSIGNED*** unconstrained arrays of *std_logic*.
  - Overloads all arithmetic and relational operators of VHDL for
    - SIGNED, INTEGER, and NATURAL types
    - UNSIGNED, INTEGER, and NATURAL.

  - With this overloading, we can use "+" for adding a signed or an unsigned *std_logic_vector* with an integer.
  - We can mix a signed or unsigned vector with an integer in relational operations, i.e., ">", "<", "<=", ">=", "=", and "\".

Std_logic_arith Package



The UNSIGNED Package



The UNSIGNED Package

- Once an object is declared as SIGNED or UNSIGNED, conversion to the other type and conversion to *std_logic* becomes difficult.

- The *std_logic_unsigned* package sits on top of the *std_logic_arith* package.
- Assumes all *std_logic_vector* declarations are unsigned and overloads all arithmetic and relational operators for unsigned numbers declared as *std_logic_vector*.

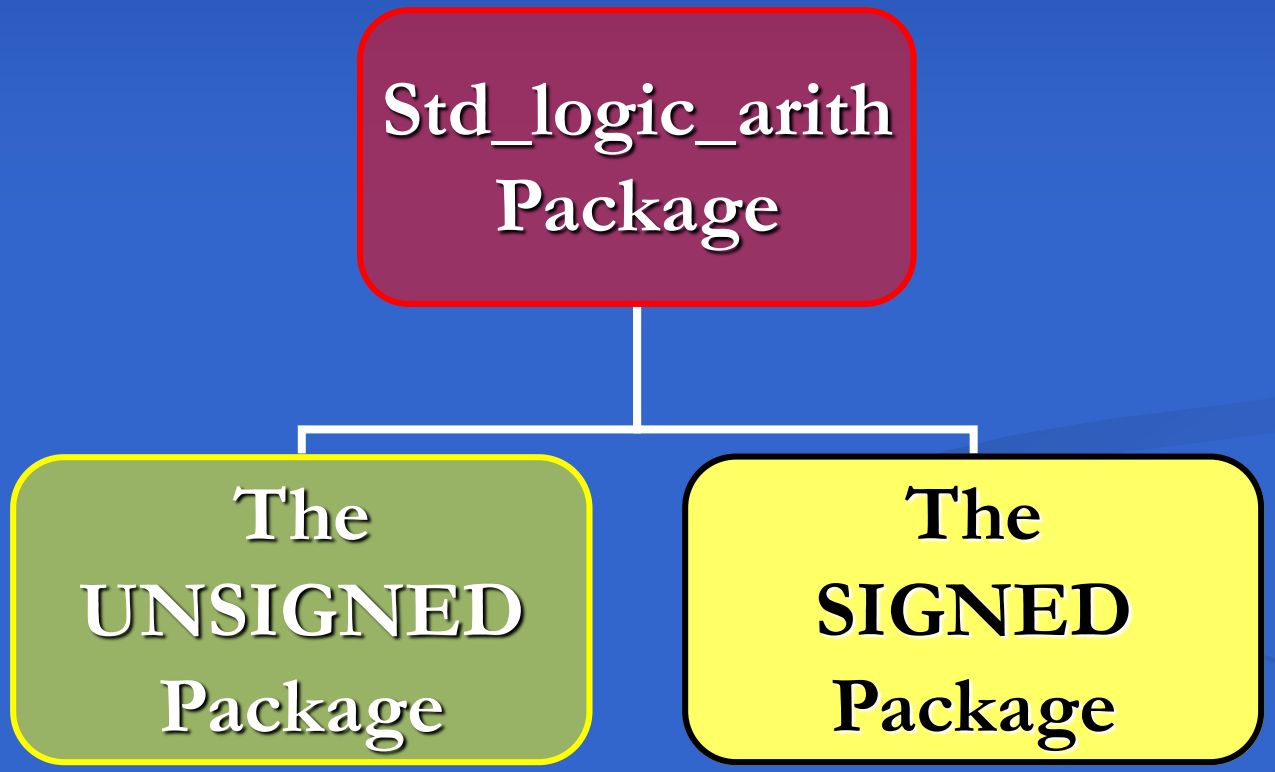
- The unsigned package already includes the arithmetic package.

The UNSIGNED Package

```
-- Use This:
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_UNSIGNED.ALL;

-- OR The Following:
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_SIGNED.ALL;
```

- Using Unsigned and Signed

The SIGNED Package



The SIGNED Package

- Sits on top of the *std_logic_arith* package
- Forces all logical and relational operators to treat their operands as signed 2's complement numbers.
- The type mark recognized in this package is *std_logic_vector* that is treated as a signed type.
- If a design requires both signed and unsigned arithmetic, the *std_logic_arith* or *NUMERIC_STD* must be used.
- SIGNED and UNSIGNED packages only allow signed or unsigned arithmetic.

Summary

- This chapter focused on
  - Linguistics aspects of VHDL
  - Types
  - Operators
  - Overloading
- Introduced standard libraries that define standard types and operators. Use of libraries and standard packages simplifies the use of VHDL for design or description of hardware based on standard technologies. A lot of times, use of packages eliminates the need for understanding many of difficult language constructs. However, for a better understanding of the languages and with a look into future technologies, the issues discussed in the earlier part of chapter become important.

Acknowledgment

Slides developed by:

Homa Alemzadeh

Last edited February 2019, by:

Saba Yousefzadeh