Part IV

Basic Elements in VHDL
Identifiers

- An **identifier** can be any length, in other words, as many characters as desired.
- An **identifier** is case insensitive, meaning that there is no difference between uppercase and lowercase letters. (For example, XOR2_OP2 and XOr2_Op denote the same identifier.)
- The allowed characters are a-z (lowercase letters), A-Z (uppercase letters), 0-9 (numerals), and _ (underscore).
- The first character is a letter and the last character must not be underscore.
- No adjacent underscore are allowed.
Identifiers (cont’d)

The identifiers discussed in the previous slide are called basic identifiers and their construction rules can, at times, be overly restrictive. For example digital system data books often designate a part name with leading numerals, such as 74HC00. Digital system data books also often designate active-0 signal state with a leading slash, such as /ALARM, or a trailing dash, such as ALARM-. These sample names are not legal VHDL names. The inability to express these names in VHDL impedes describing Existing designs in VHDL and often leads to cumbersome and awkward. Hence VHDL-93 provides an enhanced set of rules constructing identifiers that include both the basic identifiers and more general identifiers, called extended identifiers.
Extended Identifiers

- Rules of constructing extended identifiers in VHDL-93 are given:
  - An extended identifier can be any length, in other words, as many characters as desired.
  - An extended identifier must be delimited by leading and trailing backslashes, \(\text{\textbackslash}\) (for example, \texttt{\textbackslash 2XOR\_OP\textbackslash}).
  - The allowed characters are any graphic character. Graphic characters include all the characters allowed for VHDL-93 basic identifiers plus special characters such as dash “-”, asterisk “*”, A circumflex “A,” and e umlaut “e”.
  - Within the enclosing backslashes, graphic characters can appear in any order, except that a backslash used as part of an extended identifier must be denoted by two adjacent backslashes. (For example, XOR\2 is denoted by \texttt{\textbackslash XOR\textbackslash 2\textbackslash}).
  - An extended identifier is case sensitive, meaning that there is a difference between uppercase and lowercase letters. (For example, \texttt{\textbackslash XOR2\_OP\textbackslash} and \texttt{\textbackslash XOR2\_Op\textbackslash} denote different identifiers.)
Extended Identifier (cont’d)

- An extended identifier is different from any keyword or basic identifier. (Thus, \entity\ is a legal extended identifier because it is different entity. Also, the extended identifier \XOR2_OP\ and the basic identifier XOR2_Op do not denote same name.)
Data Objects

- There are four types of objects in VHDL
  - Constants
  - Variables
  - Signals
  - Files
- The scope of an object is as follows:
  - Objects declared in a package are available to all VHDL descriptions that use that package
  - Objects declared in an entity are available to all architectures associated with that entity
  - Objects declared in an architecture are available to all statements in that architecture
  - Objects declared in a process are available only within that process
Data Objects (cont’d)

- **Signal** - Values scheduled in the future, only means to communicate between processes. It is “Global”, it uses up simulator time.

- **Constants** - Value fixed during initialization and can only be manipulated by a debugger. It names a specific value.

- **Variables** - Value assigned immediately when the line is executed and is valid within processes and subprograms. (Local storage or wires)

  **Note:** Names are not case sensitive, but must not be reserved words.
Data Objects

Constants

- Name assigned to a specific value of a type
- Allow for easy update and readability
- Declaration of constant may omit value so that the value assignment may be deferred
  - Facilitates reconfiguration
- Declaration syntax:

  ```
  constant CONSTANT_NAME : TYPE_NAME [:= VALUE];
  ```

- Declaration examples:

  ```
  constant PI : real := 3.14;
  constant SPEED : integer;
  constant VEC3 : Bit_Vector (0 TO 3) := “1101”;
  ```
Data Objects

Variables

- **Provide convenient mechanism for local storage**
  - E.g. loop counters, intermediate values, etc.

- **Scope is process in which they are declared**
  - VHDL ‘93 provides for global variables, to be discussed in the Advanced Concepts in VHDL module

- **All variable assignments take place immediately**
  - No delta or user specified delay is incurred

- **Declaration syntax:**

```
variable VARIABLE_NAME : TYPE_NAME [:= VALUE];
```

- **Declaration examples :**

```
variable OPCODE : Bit_Vector(3 DOWNTO 0) := "0000";
variable FREQ : integer;
```
Data Objects
Variables (cont’d)

- Can be scalar or array and can be constrained and initialized

```vhdl
variable ABC : Bit;
variable DEF : integer range 0 to 9 := 3;
```

- Local Data inside a process or (subprogram)
- Assigned immediately
- Variable assignments do not use any simulated time
- Right hand side must match the type of the left hand Side type
- Right hand side can be an expression using operators ...
Data Objects

Signals

- Ports are signals that communicate with modules
- Architecture signals are only visible internally
- Global signals are stored in packages
- Signals can be scalar or array and can be initialized
  
  Example:  
  ```
  signal ABC: Bit := '0';
  ```

- Signals are declared in entity or architecture and assigned with `<=`

- Assignment executes in simulated time (scheduled)
  
  Example:  
  ```
  ABC <= '1' after 5 ns;
  ```
Data Objects
Signals (cont’d)

- Used for communication between VHDL components
- Real, physical signals in system often mapped to VHDL signals
- ALL VHDL signal assignments require either delta cycle or user-specified delay before new value is assumed
- Declaration syntax:

  ```vhdl
  signal SIGNAL_NAME : TYPE_NAME [:= VALUE];
  ```

- Declaration and assignment examples:

  ```vhdl
  signal BRDY : Bit;
  BRDY <= '0' after 5ns, '1' after 10ns;
  ```
library IEEE; use IEEE.STD_LOGIC_1164.all;
package SIGDECL is
  signal VCC : std_logic := '1';
  signal GROUND : std_logic := '0';
  subtype BUS_TYPE is Bit_vector (0 TO 7)
end SIGDECL;

use Work.SIGDECL.all;
entity BOARD_DESIGN is
  port (DATA_IN : in BUS_TYPE;
        DATA_OUT : out BUS_TYPE);
  signal SYS_CLK : STD_LOGIC := '1';
end BOARD_DESIGN;

architecture DATAFLOW of BOARD_DESIGN is
  signal INT_BUS:BUS_TYPE;
begin
  ......
This example highlights the difference between signals and variables.

Assuming 1 -> 0 transition on in_sig, what are the resulting values for y in both cases?
Data Objects
Signals Versus Variables (cont’d)

A key difference between variables and signals is the assignment delay.

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>OUT_1</th>
<th>OUT_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1+d</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1+2d</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

A: 0 ->1

Holds prev. value

```
architecture SIG_EX of TEST is
    signal A, B, C, OUT_1, OUT_2 : Bit;
begin
    process (A, B, C, OUT_1, OUT_2)
    begin
        OUT_1 <= A NAND B;
        OUT_2 <= OUT_1 XOR C;
    end process;
end SIG_EX;
```
## Data Objects

**Signals Versus Variables (cont’d)**

### Architecture VAR_EX of TEST

```vhdl
architecture VAR_EX of TEST is
  signal A, B, C, OUT_4 : Bit;
  begin
    process (A, B, C)
    variable OUT_3 : Bit;
    begin
      OUT_3 := A NAND B;
      OUT_4 <= OUT_3 XOR C;
    end process;
  end VAR_EX;
```

<table>
<thead>
<tr>
<th>Time</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>OUT_3</th>
<th>OUT_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1+d</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- **A**: 0 → 1
- **Schedule Change**: Immediate Change
- **Schedule Change**: Schedule Change
## Data Objects
### Signals Versus Variables (cont’d)

<table>
<thead>
<tr>
<th>Signal Assignment</th>
<th>Variable Assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal Values are Scheduled</td>
<td>Variable Values are not Scheduled</td>
</tr>
<tr>
<td>Can have delay</td>
<td>Values updated without delay</td>
</tr>
<tr>
<td>Signal updated only when all</td>
<td>Variables updated immediately within</td>
</tr>
<tr>
<td>processes are suspended (waiting).</td>
<td>the process (while the process is executing)</td>
</tr>
</tbody>
</table>
Files provide a way for a VHDL design to communicate with the host environment.

File declarations make a file available for use to a design.

Files can be opened for reading and writing.
  - In VHDL87, files are opened and closed when their associated objects come into and out of scope.
  - In VHDL93 explicit FILE_OPEN() and FILE_CLOSE() procedures were added.

The package STANDARD defines basic file IO routines for VHDL types.

The package TEXTIO defines more powerful routines handling IO of text files.
Data Types

All declarations in VHDL, i.e., ports, signals, and variables must specify their corresponding type or subtype before being used.
Data Types (cont’d)

- VHDL is a rich language with many different data types. *The most common data types are*
  - *bit*: a 1-bit value representing a wire. (Note: IEEE standard 1164 defines a 9-valued replacement for bit called std_logic).
  - *bit_vector*: an array of bits. (Replaced by std_logic_vector in IEEE 1164)
  - *boolean*: a True/False value.
  - *integer*: a signed integer value, typically implemented as a 32-bit data type.
  - *real*: a floating point value.
  - *enumerated*: used to create custom data types.
  - *record*: used to append multiple data types as a collection.
  - *array*: can be used to create single or multiple dimension arrays.
  - *access*: similar to pointers in C or Pascal
  - *file*: used to read and write disk files. Useful for simulation
  - *physical*: used to represent values such as time, voltage, etc. using symbolic units of measure (such as ‘ns’ or ‘ma’)
Data Types

Character Literals

- A single ASCII character in single quotes or an identifier for a non printing character:
  - ‘a’
  - ‘A’
  - cr
  - esc
  - ‘ ‘
  - ‘ ‘ ‘

- Character string is an array of characters in double quotes:
  - “hold time out of range”
Data Types

Bit Literals

- Bit can only be ‘0’ ‘1’
  Vendors often extend VHDL defined types with other values
- Bit_Vector is an array of Bits defined in **double quotes**
  **Example:** “001100”
- Base precedes sequence
  Binary B, default if no base specified
  Octal O
  Hexadecimal X
  **Example:** X”7E”
Data Types

Bit Type Extensions

- Problems with type BIT is resolved with the use of Multi-Valued Logic (MVL)
- Until early 1992, there was no standard
- Nine state system is defined and agreed by IEEE (Standard IEEE 1164)
- Vendors usually extend the data types for simulation (MVL7, IEEE.std_logic_1164, MVL9)
  - Z -- high impedance for three state driver
  - Also Uninitialized, X - unknown, Strong 0 and 1, Weak L and H, Don’t Care, etc.

Examples:
```plaintext
type BIT3 is ('0', '1', 'Z');
type VALUE4 is ('X', '0', '1', 'Z');
```
**Data Types**

**Standard Logic**

```vhdl
-- Defined in Package "StdLogic_1164"
"Std_logic" has same values as "Std_ulogic"
```

- **std_ulogic**
- **std_ulogic_vector**

```vhdl
type STD_ULOGIC is ('U' uninitialized, 'X' forcing to unknown, '0' forcing to 0, '1' forcing to 1, 'Z' high impedance (Three State), 'W' weak unknown, 'L' weak 0, 'H' weak 1, '-' don't care);
```

- **std_logic**
- **std_logic_vector**
Data Types

Rules to use std.Logic and std_ulogic

```vhdl
signal A, B, Z : std_ulogic;
signal RES_Z : std_logic;
```

```vhdl
Z <= A;
RES_Z <= A;
```

```vhdl
Z <= A;
RES_Z <= A;
```

```vhdl
Z <= A;
RES_Z <= A;
```

```vhdl
Z <= B;
RES_Z <= B;
```

Cannot assign std_logic_vector to std_ulogic_vector
Data Type
std_Logic Properties

- std_ulogic
- std_ulogic_vector
- std_logic
- std_logic_vector

- **Std_ulogic benefits**
  - Gives errors when accidentally have two drivers
- **Std_logic benefits**
  - Other standards based on it
    - Gate Level Simulation
    - Mathematical Functions
  - Synthesis tools only output netlists in one data type
    - Required so tri-state busses work
  - Generally no simulation speed overhead

**std_logic is best when used for RTL**
Data Type

bit_vector

VHDL has package standard type declarations including bit_vector which is an array of bits (unconstrained)

Assigned examples to C of type bit_vector of 4-Bits:

- \( C := \text{"1010"} ; \)  
  -- Constant bit_vector
- \( C := S \& T \& M \& V ; \)  
  -- 4 1-bit signals concatenated
- \( C := ('1', '0', '1', '0') ; \)  
  -- 4 Bit Aggregate
- \( C := 3 ; \)  
  -- Invalid
Data Type
Boolean Literals

- Boolean is pre-defined to be False, True
- Relational Operators (=, <=, >=, /=) produce a Boolean result
- Boolean values can be tested in if statements and assigned
- A Boolean is not a Bit and has no pre-defined relationship to a Bit
- A user conversion routine could define a Bit/Boolean relationship.
Data Type
Built-in and Pre-defined Types

- A few types are built-in the language/compiler
- Most types pre-declared in the package standard:
  Bit, Boolean, Character, Bit_Vector, string, Line, Text
- Example of type declaration from package:
  type Boolean is (False, True);
- User needs to declare types: default initial value
  Real array
  Integer array
  Special types
- Constants, signals and variables must be of a previously declared type
Data Type
Scalar Types

- Integer
  - Minimum range for any implementation as defined by standard:
    - -2,147,483,647 to 2,147,483,647
  - Example assignments to a variable of type integer:

```vhdl
architecture TEST_INT of test is
begin
  process (X)
  variable A: integer;
  begin
    A := 1;         -- OK
    A := -1;        -- OK
    A := 1.0;       -- illegal
  end process;
end TEST_INT;
```
Data Type
Scalar Types (cont’d)

- Integer
- Represent Integer values with fixed point
- Operators such as +, -, * can be used
- Minimum range specified by standard is
  -2, 147, 483, 647 to +2, 147, 483, 647

Example:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15</td>
<td>-21</td>
</tr>
</tbody>
</table>

- Users can specify a particular range of values (constraint)
Data Type
Scalar Types (cont’d)

- Real
  - Minimum range for any implementation as defined by standard:  -1.0E38
  - Example assignments to a variable of type real:

```vhdl
architecture TEST_REAL of TEST is
begin
    process (X)
    variable A: real;
    begin
        A := 1.3;  -- OK
        A := -7.5;  -- OK
        A := 1;  -- illegal
        A := 1.7E13;  -- OK
        A := 5.3 ns;  -- illegal
    end process;
end TEST_REAL;
```
Data Type
Scalar Types (cont’d)

Real Literals
Represent Floating Point values
Format is: + or - number.number [E + or - number]
Example:

  decimal point is required

1.0
1
-1.0 E 10
1.5 E -20
5.3

illegal integer value
Data Type
Range Constraints

- Simulator checks for valid type and range on assignment

- Range constraint specifies a restricted range

- Examples:

  ```
  integer range 0 to 9
  0 to 9 (Integer type is implicit because of literals)
  real range 1.0 to 1.1
  integer range (no constraint)
  ```

- Used to qualify the intended usage of data
Data Type
Slice of an Array

- A subscript array reference can be used on either side of a signal or variables assignment statement.

Example:

```vhdl
port(
    A: in Bit_Vector(0 TO 4); -- Direction of Declaration
    C: out Bit_Vector(8 DOWNTO 1)); -- and Slice **must be** same

C(6 DOWNTO 3) <= A(0 TO 3); -- OK
C(5 DOWNTO 0) <= A(3 DOWNTO 0) -- ILLEGAL
```

**Note:** Size of array on left and right must be equal.
**Data Type**

**Array Assignment**

```vhdl
signal Z_BUS: std_logic_vector(3 DOWNTO 0);
signal C_BUS: std_logic_vector(0 TO 3);

Z_BUS <= C_BUS;
```

Elements are assigned by position, not element number.
Be consistent in defining the direction of your arrays..
Data Type
Array Example

architecture ...
begin
  process
    variable DATA : Bit_Vector(0 TO 31);
    variable START : integer range 0 TO 24;
    variable DATA_OUT : Bit_Vector (0 TO 7)
    ...
    begin
      for i in 0 to 7 loop
        DATA_OUT(i) := DATA(i + START);
      end loop;
    end process;
  ...
end process;
Data Type
Array Example

architecture EXAMPLE of ARRAY is
begin
  process(A)
    type BIT4 is ARRAY(0 TO 3) of Bit;
    type BIT8 is ARRAY (0 TO 7) of Bit;
    variable Q8 : Bit8;
    variable Q4A, Q4B : Bit4;
  begin
    Q4A := ('1', '1','1','1'); -- aggregate assignment
    Q4B := "1010" ; -- aggregate assignment
    Q8 := ('0','1', others => '0');  -- aggregate assignment
    Q8 := (others => '0'); -- aggregate assignment
    Q8(2) := '1'; -- assignment to a slice
    Q8(5 TO 7) := "111"; -- assignment to a slice
    Q8 := "010" & "00110"; -- concatenation
  end process;
end EXAMPLE;
Data Type
Scalar Types

type BINARY is (ON, OFF);
... some statements ...
architecture TEST_ENUM of TEST is
begin
  process (X)
    variable A: BINARY;
  begin
    A := ON; -- OK
    ... more statements ...
    A := OFF; -- OK
    ... more statements ...
  end process;
end TEST_ENUM;
Data Type
Scalar Types (cont’d)

- **Physical**
  - Require associated units
  - Range must be specified
  - Example of physical type declaration:

```vhdl
    type RESISTANCE is range 0 to 1000000
    UNITS
    OHM;  -- ohm
    KOHM = 1000 OHM;  -- i.e. 1 KOHM
    MOHM = 1000 KOHM;  -- i.e. 1 MOHM
    end UNITS;
```

Time is the only physical type predefined in VHDL standard.
Data Type
Scalar Types (cont’d)

- Defined in package standard:
  
  ```
  type TIME is range 0 to INTEGER’HIGH
  UNITS
  
  fs; -- femtosecond
  ps = 1000 fs; -- picosecond
  ns = 1000 ps; -- nanosecond
  us = 1000 ns; -- microsecond
  ms = 1000 us; -- millisecond
  sec = 1000 ms; -- second
  min = 60 sec; -- minute
  hr = 60 min; -- hour
  
  end UNITS;
  ```
Data Type

Enumerated Types

- Defines legal values through an enumerated list of:
  - Character Literals: ‘x’
  - Identifiers: STATE_IDLE

Example:

```plaintext
type INSTRUCTION is (ADD, SUB, LDA, LDB, STA, STB, XFR);
type VALUE is ('X', '0', '1', 'Z');
```

- Uses symbolic codes instead of numeric values
- Provides for more abstract representation in source design
- Simulator systems require more values than ‘0’, ‘1’
- User defines list of possible values
architecture BEHAVE of MP is

    type INSTRUCTION is (ADD, LDA, LDB); -- a list of identifiers

begin
    process
        variable INST: INSTRUCTION;
        variable A, B: integer;
    begin
        case INST is
            when LDA => A := DATA; -- load accumulator A
            when LDB => B := DATA; -- load accumulator B
            when ADD => A := A + B; -- add two accumulators
        end case;

    .........
Data Types

Composite Types: Array

- Array
  - group of elements of same type
  - elements can be scalar or composite type
  - multi-dimensional arrays allowed
  - useful for modeling RAMs, ROMs, Busses
  - used for lookup tables
  - object(s) accessed by index
Data Types
Composite Types: Array (cont’d)

- Array
  - Used to group elements of the same type into a single VHDL object
  - Range may be unconstrained in declaration
  - Range would then be constrained when array is used
  - Example declaration for one-dimensional array (vector):

```vhdl
type DATA_BUS is array (0 to 31) of Bit;
```

<table>
<thead>
<tr>
<th>0</th>
<th>...element indices...</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>...array values...</td>
<td>1</td>
</tr>
</tbody>
</table>

- Declaration:
  - `variable X: DATA_BUS;`
  - `variable Y: Bit`
  - `Y := X(12);` -- `Y` gets value of element at index 12
Example one-dimensional array using `downto` (DOWNTO keyword must be used if leftmost index is greater than rightmost index)

```
type REGISTER is ARRAY (15 DOWNTO 0) of bit;
```

```
15...element indices... 0

0    ...array values... 1

variable X: REGISTER;
variable Y: bit

Y := X(4);  -- Y gets value of element at index 4
```
Data Types
Composite Types: Records

- Record
  - group of different type objects
  - element can be scalar or composite types
  - useful for modeling data packets, instructions
  - object(s) accessed by name
Data Types

Composite Types: Records (cont’d)

- **Records**
  - Used to group elements of possibly different types into a single VHDL object
  - Elements are indexed via field names
  - Examples of record declaration and usage:

```vhdl
type BINARY is (ON, OFF);
type SWITCH_INFO is
  record
    STATUS : BINARY;
    IDNUMBER : integer;
  end record;

variable SWITCH : SWITCH_INFO;
SWITCH.STATUS := ON;  -- status of the switch
SWITCH.IDNUMBER := 30;  -- e.g. number of the switch
```
Data Types
Composite Types: Record Example

```plaintext
type DELAY_ENTRY is
  record
    FALL : TIME;
    RISE : TIME;
  end RECORD;

variable DELAY_TIME : DELAY_ENTRY;
begin
  DELAY_TIME.FALL := 10NS
  DELAY_TIME.RISE := 12NS;
```

Data Types

Subtypes

- Subtype
  - Allows for user defined constraints on a data type
e.g. a subtype based on an unconstrained VHDL type
  - May include entire range of base type
  - Assignments that are out of the subtype range are illegal
    Range violation detected at run time rather than compile time
    because only base type is checked at compile time
  - Subtype declaration syntax :

    ```
    subtype NAME is BASE_TYPE range <user range>;
    ```

- Subtype example :

  ```
  subtype FIRST_TEN is integer range 0 TO 9;
  ```
Data Types

Subtypes (cont’d)

- Subset of a base type
- May contain 1 or all elements of the base type
- Allows assignments of elements between base type and subtype that are within the range of the subtype
- The IEEE resolved type ‘std_logic’ is a subtype of the resolved base ‘std_ulogic’
- Adds constraints to increase usefulness of range checking and minimize number of options in CASE statements
Data Types

Subtypes (cont’d)

- A new subtype can be created by constraining an existing type

  Example:

  ```
  subtype DIGIT is integer range 0 TO 9;
  type INSTRUCTION is (ADD, SUB, MUL, DIV, LDA, STA, OUTA, XFR);
  subtype ARITHMETIC is INSTRUCTION range ADD TO DIV;
  subtype BYTE is Bit_Vector(1 TO 8);  -- creating an array subtype
  ```

- Simulators perform range and type checking during assignments
- More convenient for frequently used data types (no repeated constraints)
- Provides for more object oriented design and better documentation
Data Types

Summary

- All declarations of VHDL ports, signals, and variables must include their associated type or subtype

- Three forms of VHDL data types are:
  - Access -- pointers for dynamic storage allocation
  - Scalar -- includes Integer, Real, Enumerated, and Physical
  - Composite -- includes Array, and Record

- A set of built-in data types are defined in VHDL standard
  - User can also define own data types and subtypes
Attributes

Attributes provide information about certain items in VHDL, e.g.:

- Types, subtypes, procedures, functions, signals, variables, constants, entities, architectures, configurations, packages, components
- General form of attribute use:

  NAME'ATTRIBUTE_IDENTIFIER  -- read as “tick”

- VHDL has several predefined, e.g.:
  - X'EVENT  -- TRUE when there is an event on signal X
  - X'LAST_VALUE  -- returns the previous value of signal X
  - Y'HIGH  -- returns the highest value in the range of Y
  - X'STABLE(t)  -- TRUE when no event has occurred on signal X in the past ‘t’ time
Attributes
Register Example (cont’d)

- The following example shows how attributes can be used to make an 8-bit register
- Specifications:
  - Triggers on rising clock edge
  - Latches only on enable high
  - Has a data setup time of x_setup
  - Has propagation delay of prop_delay

```vhdl
entity 8_BIT_REG is
generic (X_SETUP, PROP_DELAY : TIME);
port(ENABLE, CLK : in QSIM_STATE;
    A : in QSIM_STATE_VECTOR (7 DOWNTO 0);
    B : out QSIM_STATE_VECTOR (7 DOWNTO 0));
end 8_BIT_REG;
```

qsim_state type is being used - includes logic values 0, 1, X, and Z
The following architecture is a first attempt at the register:

```
architecture FIRST_ATTEMPT of 8_BIT_REG is
begin
    process (CLK)
    begin
        if (ENABLE = '1') and A'STABLE(X_SETUP) and (CLK = '1') then
            B <= A after DELAY;
        end if;
    end process;
end FIRST_ATTEMPT;
```

What happens if `a` does not satisfy its setup time requirement of `x_setup`?
The following architecture is a second and more robust attempt
The use of 'LAST_VALUE ensures the clock is rising from a value of '0'

```vhdl
architecture BEHAVIOR of 8_BIT_REG is
begin
  process (CLK)
  begin
    if (ENABLE = '1') and A'(STABLE(X_SETUP) and
       (CLK = '1') and (CLK'LASTVALUE = '0') then
      B <= A after DELAY;
    end if;
  end process;
end BEHAVIOR;
```

A THEN clause could be added to define the behavior when the requirements are not satisfied
Operators

- Operators can be chained to form complex expressions, e.g.:

  ```
  RES <= A and not(B) or not(A) and B;
  ```

- Can use parentheses for readability and to control the association of operators and operands.

- Defined precedence levels in decreasing order:

  - Miscellaneous operators: **, abs, not
  - Multiplication operators: *, /, mod, rem
  - Sign operator: +, -
  - Addition operators: +, -, &
  - Shift operators: sll, srl, sla, sra, rol, ror
  - Relational operators: =, /=, <, <=, >, >=
  - Logical operators: AND, OR, NAND, NOR, XOR, XNOR
# Logical Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>conjunction</td>
</tr>
<tr>
<td>or</td>
<td>disjunction</td>
</tr>
<tr>
<td>xor</td>
<td>exclusive disjunction</td>
</tr>
<tr>
<td>xnor</td>
<td>complement exclusive disjunction</td>
</tr>
<tr>
<td>nand</td>
<td>complement conjunction</td>
</tr>
<tr>
<td>nand</td>
<td>complement conjunction</td>
</tr>
<tr>
<td>nor</td>
<td>complement disjunction</td>
</tr>
<tr>
<td>not</td>
<td>complement</td>
</tr>
</tbody>
</table>

VHDL-93: The predefined logic operator **xnor** is provided only in VHDL-93.
Logical Operators (cont’d)

There are a few rules for evaluating VHDL logic operators that unfortunately do not align exactly with switching algebra theory, so it is important to note them. The **and**, **or**, **nand** and **nor** logic operators are called short-circuit operators because the right operand is not evaluated if the value of the left operand determines the result of the logic operator. For example, consider the following signals assignment statement.

```
TEST_SIG <= OPR_A and OPR_B;
```
Operators

Examples

The concatenation operator &

```vhdl
variable SHIFTED, SHIFTIN : Bit_Vector (0 TO 3);
...
SHIFTED := SHIFTIN(1 TO 3) & '0';
```

The exponentiation operator **

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>x := 5**5</td>
<td>5^5</td>
</tr>
<tr>
<td>y := 0.5**3</td>
<td>0.5^3</td>
</tr>
<tr>
<td>x := 4**0.5</td>
<td>4^0.5</td>
</tr>
<tr>
<td>y := 0.5**(-2)</td>
<td>0.5^(-2)</td>
</tr>
</tbody>
</table>
Operators

Examples

The addition operator +

```vhdl
entity ADD is
  port ( B,C: in Bit;  Z: out Bit);
end ADD;
archtecture RTL of ADD is
begin
  Z <= A + B;
end RTL;
```

Logical Operators

```vhdl
library IEEE;
use IEEE.Std_Logic_1164.all;
entity MVLS is
  port (A, B,C: in Std_Logic;  Z: out Std_Logic);
end ADD;
archtecture LOGIC of MVLS is
begin
  Z <= A AND NOT (B OR C);
end RTL;
```
Operators
Examples

Relational Operators

if $A = B$ then

$Z \leq '1';$

$>$
greater than

$\geq$
greater than or equal to

$=$
equality

$\leq$
less than or equal to

$<$
less than

$\neq$
inequality
Operators

Questions

Which statements are correct in VHDL?

```
architecture QUIZ of OPERATORS is
    signal BOOL : boolean;
    signal A_INT, B_INT, Z_INT : integer range 0 TO 15;
    signal Z_BIT : Std_Logic;
    signal A_VEC, B_VEC, Z_VEC : Std_Logic_Vector (3 DOWNTO 0);
begin
    Z_BIT <= A_INT = B_INT;
    BOOL <= A_INT > B_VEC;
    Z_VEC <= A_VEC & B_VEC;
    Z_VEC <= A_VEC(1 TO 0) & A_VEC(1 DOWNTO 0);
    Z_VEC <= A_VEC(1 DOWNTO 0) & B_VEC(1 DOWNTO 0);
end;
```
Summary

- VHDL is a worldwide standard for the description and modeling of digital hardware.
- VHDL gives the designer many different ways to describe hardware.
- Familiar programming tools are available for complex and simple problems.
- Sequential and concurrent modes of execution meet a large variety of design needs.
- Packages and libraries support design management and component reuse.