Lecture 3

VHDL Fundamentals

Required reading

• P. Chu, *RTL Hardware Design using VHDL*

  *Chapter 2, Overview of Hardware Description Languages*
  *Chapter 3, Basic Language Constructs of VHDL*

Recommended reading

• Wikipedia – The Free On-line Encyclopedia

Differences between Hardware Description Languages (HDL) and Traditional Programming Languages (PL)

Traditional PL

• Modeled after a sequential process
  - Operations performed in a sequential order
  - Help human’s thinking process to develop an algorithm step by step
  - Resemble the operation of a basic computer model

HDL

• Characteristics of digital hardware
  - Connections of parts
  - Concurrent operations
  - Concept of propagation delay and timing
• Characteristics cannot be captured by traditional PLs
• Require new languages: HDL
Use of an HDL program

- Formal documentation
- Input to a simulator
- Input to a synthesizer

• Highlights of modern HDL:
  - Encapsulate the concepts of entity, connectivity, concurrency, and timing
  - Incorporate propagation delay and timing information
  - Consist of constructs for structural implementation
  - Incorporate constructs for behavioral description (sequential execution of traditional PL)
  - Describe the operations and structures in gate level and RT level.
  - Consist of constructs to support hierarchical design process

Levels of design description

Levels supported by HDL

- Algorithmic level
- Register Transfer Level
- Logic (gate) level
- Circuit (transistor) level
- Physical (layout) level

Level of description most suitable for synthesis

Register Transfer Level (RTL) Design Description

Combinational Logic

Combinational Logic

Registers

Two HDLs used today

- VHDL and Verilog
- Syntax and “appearance” of the two languages are very different
- Capabilities and scopes are quite similar
- Both are industrial standards and are supported by most software tools
Brief History of VHDL

• VHDL is a language for describing digital hardware used by industry worldwide

• VHDL is an acronym for VHSIC (Very High Speed Integrated Circuit) Hardware Description Language

Genesis of VHDL

State of art circa 1980

• Multiple design entry methods and hardware description languages in use
• No or limited portability of designs between CAD tools from different vendors
• Objective: shortening the time from a design concept to implementation from 18 months to 6 months

A Brief History of VHDL

• June 1981: Woods Hole Workshop
• July 1983: contract to develop VHDL awarded by the United States Air Force to
  • Intermetrics (language experts)
  • Texas Instruments (chip design experts)
  • IBM (computer system design experts)
• August 1985: VHDL Version 7.2 released

Versions of VHDL

• Four versions of VHDL:
  • IEEE-1076 1987
  • IEEE-1076 1993 ➔ most commonly supported by CAD tools
  • IEEE-1076 2000 (minor changes)
  • IEEE-1076 2002 (minor changes)
  • IEEE-1076 2008
Verilog

- Essentially identical in function to VHDL
- Simpler and syntactically different
  - C-like
  - Gateway Design Automation Co., 1985
  - Gateway acquired by Cadence in 1990
  - IEEE Standard 1364-1995 (Verilog-95)
  - Early de facto standard for ASIC design
- Subsequent versions
  - Verilog 2001 (major extensions) → dominant version used in industry
  - Verilog 2005 (minor changes)
    (new features and capabilities aiding design verification and design modeling)
- Programming language interface to allow connection to non-Verilog code

VHDL vs. Verilog

<table>
<thead>
<tr>
<th></th>
<th>Government Developed</th>
<th>Commercially Developed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ada based</td>
<td></td>
<td>C based</td>
</tr>
<tr>
<td>Strongly Type Cast</td>
<td></td>
<td>Mildly Type Cast</td>
</tr>
<tr>
<td>Case-insensitive</td>
<td></td>
<td>Case-sensitive</td>
</tr>
<tr>
<td>Difficult to learn</td>
<td></td>
<td>Easier to Learn</td>
</tr>
<tr>
<td>More Powerful</td>
<td></td>
<td>Less Powerful</td>
</tr>
</tbody>
</table>

How to learn Verilog by yourself?

- Features of VHDL and Verilog
  - Technology/vendor independent
  - Portable
  - Reusable
Naming and Labeling (1)

- VHDL is case insensitive
  
  Example:
  
  Names or labels
  databus
  Databus
  DataBus
  DATABUS
  are all equivalent

Naming and Labeling (2)

General rules of thumb (according to VHDL-87)

1. All names should start with an alphabet character (a-z or A-Z)
2. Use only alphabet characters (a-z or A-Z) digits (0-9) and underscore (_)
3. Do not use any punctuation or reserved characters within a name (!, ?, ., &, +, -, etc.)
4. Do not use two or more consecutive underscore characters (__) within a name (e.g., Sel__A is invalid)
5. All names and labels in a given entity and architecture must be unique

Valid or invalid?

7segment_display
A87372477424
Adder/Subtractor
/reset
And_or_gate
AND__OR__NOT
Kogge-Stone-Adder
Ripple&Carry_Adder
My adder

Extended Identifiers

Allowed only in VHDL-93 and higher:

1. Enclosed in backslashes
2. May contain spaces and consecutive underscores
3. May contain punctuation and reserved characters within a name (!, ?, ., &, +, -, etc.)
4. VHDL keywords allowed
5. Case sensitive

Examples:
/rdy/ /My design/ /!a/
/RDY/ /my design/ /^a/

Free Format

- VHDL is a “free format” language

  No formatting conventions, such as spacing or indentation imposed by VHDL compilers. Space and carriage return treated the same way.

  Example:

  if (a-b) then
  or
  if (a-b) then
  or
  if (a = b) then
  are all equivalent

Readability standards & coding style

Adopt readability standards based on one of the
the two main textbooks:
Chu or Brown/Vranesic

Use coding style recommended in
OpenCores Coding Guidelines
linked from the course web page

Strictly enforced by the TA and the Instructor.
Penalty points may be enforced for not following
these recommendations!!!
Comments

- Comments in VHDL are indicated with a "double dash", i.e., "--"
  - Comment indicator can be placed anywhere in the line
  - Any text that follows in the same line is treated as a comment
  - Carriage return terminates a comment
  - No method for commenting a block extending over a couple of lines

Examples:
-- main subcircuit
Data_in <= Data_bus; -- reading data from the input FIFO

Example: NAND Gate

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Design Entity

- Most basic building block of a design.
- One entity can have many different architectures.

Example VHDL Code

- 3 sections to a piece of VHDL code
- File extension for a VHDL file is .vhd
- Name of the file should be the same as the entity name (nand_gate.vhd) [OpenCores Coding Guidelines]
Entity Declaration

- Entity Declaration describes the interface of the component, i.e. input and output ports.

Entity declaration – simplified syntax

ENTITY entity_name IS
    PORT ( 
        port_name : port_mode signal_type;
        port_name : port_mode signal_type;
        ...........
        port_name : port_mode signal_type);
END entity_name;

Port Mode IN

Port Mode OUT

Port Mode OUT (with extra signal)

Port Mode INOUT (typically avoided)
Port Modes - Summary

The Port Mode of the interface describes the direction in which data travels with respect to the component:

- **In**: Data comes into this port and can only be read within the entity. It can appear only on the right side of a signal or variable assignment.

- **Out**: The value of an output port can only be updated within the entity. It cannot be read. It can only appear on the left side of a signal assignment.

- **Inout**: The value of a bi-directional port can be read and updated within the entity model. It can appear on both sides of a signal assignment.

Architecture (Architecture body)

- Describes an implementation of a design entity
- Architecture example:

  ```vhdl
  ARCHITECTURE model OF nand_gate IS
  BEGIN
  z <= a NAND b;
  END model;
  ```

Architecture – simplified syntax

```vhdl
ARCHITECTURE architecture_name OF entity_name IS
[ declarations ]
BEGIN
  code
END architecture_name;
```

Entity Declaration & Architecture

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY nand_gate IS
  PORT:
    a : IN STD_LOGIC;
    b : IN STD_LOGIC;
    z : OUT STD_LOGIC;
END nand_gate;
ARCHITECTURE dataflow OF nand_gate IS
  BEGIN
    z <= a NAND b;
  END dataflow;
```

Tips & Hints

Place each entity in a different file.
The name of each file should be exactly the same as the name of an entity it contains.

These rules are not enforced by all tools but are worth following in order to increase readability and portability of your designs.

Tips & Hints

Place the declaration of each port, signal, constant, and variable in a separate line.

These rules are not enforced by all tools but are worth following in order to increase readability and portability of your designs.
VHDL Library

- A place to store the analyzed design units
- Normally mapped to a directory in host computer
- Software define the mapping between the symbolic library and physical location
- Default library: "work"
- Library "ieee" is used for many ieee packages

• E.g.

```vhdl
library ieee; use ieee.std_logic_1164.all;
```

• Line 1: invoke a library named ieee
• Line 2: makes std_logic_1164 package visible to the subsequent design units
• The package is normally needed for the std_logic/std_logic_vector data type

Library Declarations

```vhdl
LIBRARY ieee;  USE ieee.std_logic_1164.all;
ENTITY nand_gate IS
  PORT (a: IN STD_LOGIC; b: IN STD_LOGIC; z: OUT STD_LOGIC);
END
ARCHITECTURE model OF nand_gate IS
  BEGIN
    z <= a NAND b;
  END;
```

Library declarations - syntax

```vhdl
LIBRARY library_name; USE library_name.package_name.package_parts;
```

Fundamental parts of a library

```vhdl
LIBRARY
  PACKAGE 1
    TYPES
    CONSTANS
    FUNCTIONS
    PROCEDURES
    COMPONENTS
  PACKAGE 2
    TYPES
    CONSTANS
    FUNCTIONS
    PROCEDURES
    COMPONENTS
```
Libraries

- **ieee**
  
  Specifies multi-level logic system, including STD_LOGIC, and STD_LOGIC_VECTOR data types

- **std**
  
  Specifies pre-defined data types (BIT, BOOLEAN, INTEGER, REAL, SIGNED, UNSIGNED, etc.), arithmetic operations, basic type conversion functions, basic text I/O functions, etc.

- **work**
  
  Holds current designs after compilation

Need to be explicitly declared

Visible by default

Processing of VHDL code

- **Analysis**
  
  - Performed on “design unit” basis
  
  - Check the syntax and translate the unit into an intermediate form
  
  - Store it in a library

- **Elaboration**
  
  - Bind architecture body with entity
  
  - Substitute the instantiated components with architecture description
  
  - Create a “flattened” description

- **Execution**
  
  - Simulation or synthesis

STD_LOGIC

**STD_LOGIC Demystified**

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY nand_gate IS
  PORT (a : IN STD_LOGIC;
        b : IN STD_LOGIC;
        z : OUT STD_LOGIC);
END nand_gate;

ARCHITECTURE dataflow OF nand_gate IS
  BEGIN
    z <= a NAND b;
  END dataflow;
```

What is **STD_LOGIC** you ask?

**STD_LOGIC type demystified**

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'U'</td>
<td>Uninitialized</td>
</tr>
<tr>
<td>'X'</td>
<td>Forcing (Strong driven) Unknown</td>
</tr>
<tr>
<td>'0'</td>
<td>Forcing (Strong driven) 0</td>
</tr>
<tr>
<td>'1'</td>
<td>Forcing (Strong driven) 1</td>
</tr>
<tr>
<td>'Z'</td>
<td>High Impedance</td>
</tr>
<tr>
<td>'W'</td>
<td>Weak (Weakly driven) Unknown</td>
</tr>
<tr>
<td>'L'</td>
<td>Weak (Weakly driven) 0</td>
</tr>
<tr>
<td>'H'</td>
<td>Weak (Weakly driven) 1</td>
</tr>
<tr>
<td>'-'</td>
<td>Don't Care</td>
</tr>
</tbody>
</table>

**BIT versus STD_LOGIC**

- **BIT type** can only have a value of ‘0’ or ‘1’

- **STD_LOGIC** can have nine values
  

  - Useful mainly for simulation

  - ‘0’, ‘1’, ‘Z’, ‘-’ are synthesizable

  (your codes should contain only these four values)
More on STD_LOGIC Meanings (1)

Contestion on the bus

More on STD_LOGIC Meanings (2)

More on STD_LOGIC Meanings (3)

More on STD_LOGIC Meanings (4)

- Do not care.
- Can be assigned to outputs for the case of invalid inputs (may produce significant improvement in resource utilization after synthesis).
- Must be used with great caution.

For example in VHDL, the direct comparison $'- = 'L'$ gives FALSE.

The "std_match" functions defined in the numeric_std package must be used to make this value work as expected:

Example:

```vhdl
if (std_match(address, "-11---") then...
eselif (std_match(address, "-01---") then...
else ...
end if;
```

Resolving logic levels

<table>
<thead>
<tr>
<th></th>
<th>U</th>
<th>X</th>
<th>0</th>
<th>1</th>
<th>Z</th>
<th>W</th>
<th>L</th>
<th>H</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>U</td>
<td>U</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>0</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>Z</td>
<td>W</td>
<td>L</td>
<td>H</td>
<td>X</td>
</tr>
<tr>
<td>Z</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>Z</td>
<td>W</td>
<td>L</td>
<td>H</td>
<td>X</td>
</tr>
<tr>
<td>W</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>L</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td>W</td>
<td>L</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>H</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>1</td>
<td>H</td>
<td>W</td>
<td>W</td>
<td>H</td>
<td>X</td>
</tr>
<tr>
<td>-</td>
<td>U</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

STD_LOGIC Rules

- In ECE 545 use std_logic or std_logic_vector for all entity input or output ports.
- Do not use integer, unsigned, signed, bit for ports.
- You can use them inside of architectures if desired.
- You can use them in generics.
- Instead use std_logic_vector and a conversion function inside of your architecture.

[Consistent with OpenCores Coding Guidelines]
### Modeling Wires and Buses

#### Standard Logic Vectors

```vhdl
SIGNAL a: STD_LOGIC;
SIGNAL b: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL d: STD_LOGIC_VECTOR(15 DOWNTO 0);
SIGNAL e: STD_LOGIC_VECTOR(8 DOWNTO 0);
```

- `a` <= `'1'`;  
- `b` <= "0000";  
- `c` <= B"0000";  
- `d` <= X"AF67";  
- `e` <= O"723";  

---

**Binary base assumed by default**

- `a` <= "0000";  
- `b` <= "1111";  
- `c` <= a & b;  
- `d` <= '0' & "00001111";  
- `e` <= X"AF67" & X"00001111";  

---

**Binary base explicitly specified**

- `a` <= "0000";  
- `b` <= "1111";  
- `c` <= a & b;  
- `d` <= "00001111";  
- `e` <= O"723" & X"00001111";  

---

**Hexadecimal base**

- `a` <= "0000";  
- `b` <= "1111";  
- `c` <= a & b;  
- `d` <= "00001111";  
- `e` <= O"723" & X"00001111";  

---

**Octal base**

- `a` <= "0000";  
- `b` <= "1111";  
- `c` <= a & b;  
- `d` <= "00001111";  
- `e` <= O"723" & X"00001111";  

---

#### Types of VHDL Description: Convention used in this class

- **dataflow**  
- **structural**  
- **behavioral**

- **Concurrent statements**  
- **Components and interconnects**  
- **Sequential statements**

- **Testbenches**  
- **Registers**  
- **State machines**  
- **Decoders**

**Subset most suitable for synthesis**
Types of VHDL Description: Alternative convention

**VHDL Descriptions**
- **Structural**
  - Components & interconnects
- **Behavioral**
  - Concurrent statements
  - Sequential statements
- **Behavioral**
  - Dataflow
- **Behavioral**
  - Algorithmic

**Entity xor3_gate**

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY xor3_gate IS
  PORT(
    A : IN STD_LOGIC;
    B : IN STD_LOGIC;
    C : IN STD_LOGIC;
    Result : OUT STD_LOGIC
  );
end xor3_gate;
```

**Dataflow Architecture (xor3_gate)**

```vhdl
ARCHITECTURE dataflow OF xor3_gate IS
SIGNAL U1_OUT: STD_LOGIC;
BEGIN
  U1_OUT <= A XOR B;
  Result <= U1_OUT XOR C;
END dataflow;
```

**Dataflow Description**
- Describes how data moves through the system and the various processing steps.
- Dataflow uses series of concurrent statements to realize logic.
- Dataflow is the most useful style to describe combinational logic
- Dataflow code also called "concurrent" code
- Concurrent statements are evaluated at the same time; thus, the order of these statements doesn’t matter
- This is not true for sequential/behavioral statements

**Structural Architecture in VHDL 93**

```vhdl
ARCHITECTURE structural OF xor3_gate IS
SIGNAL U1_OUT: STD_LOGIC;
BEGIN
  U1: entity work.xor2(dataflow)
  PORT MAP (I1 => A, I2 => B, Y => U1_OUT);
  U2: entity work.xor2(dataflow)
  PORT MAP (I1 => U1_OUT, I2 => C, Y => Result);
END structural;
```

This order...
- U1_out <= A XOR B;
- Result <= U1_out XOR C;

Is the same as this order...
- Result <= U1_out XOR C;
- U1_out <= A XOR B;
**xor2**

**xor2.vhd**

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY xor2 IS
    PORT(
        I1 : IN STD_LOGIC;
        I2 : IN STD_LOGIC;
        Y : OUT STD_LOGIC);
END xor2;

ARCHITECTURE dataflow OF xor2 IS
    BEGIN
        Y <= I1 xor I2;
    END dataflow;
```

---

**Structural Architecture in VHDL 87**

```vhdl
ARCHITECTURE structural OF xor3_gate IS
    SIGNAL U1_OUT : STD_LOGIC;

COMPONENT xor2
    PORT(
        I1 : IN STD_LOGIC;
        I2 : IN STD_LOGIC;
        Y : OUT STD_LOGIC);
END COMPONENT;

BEGIN
    U1: xor2 PORT MAP (I1 => A,
                       I2 => B,
                       Y  => U1_OUT);
    U2: xor2 PORT MAP (I1 => U1_OUT,
                       I2 => C,
                       Y  => Result);
END structural;
```

---

**Structural Description**

- Structural design is the simplest to understand. This style is the closest to schematic capture and utilizes simple building blocks to compose logic functions.
- Components are interconnected in a hierarchical manner.
- Structural descriptions may connect simple gates or complex, abstract components.
- Structural style is useful when expressing a design that is naturally composed of sub-blocks.

---

**Behavioral Architecture (xor3 gate)**

```vhdl
ARCHITECTURE behavioral OF xor3 IS
    BEGIN
        xor3_behave: PROCESS (A, B, C)
        BEGIN
            IF ((A XOR B XOR C) = '1') THEN
                Result <= '1';
            ELSE
                Result <= '0';
            END IF;
        END PROCESS xor3_behave;
    END behavioral;
```

---

**Behavioral Description**

- It accurately models what happens on the inputs and outputs of the black box (no matter what is inside and how it works).
- This style uses PROCESS statements in VHDL.

---

**Numbers, characters and strings**

- **Number:**
  - Integer: 0, 1234, 98E7
  - Real: 0.0, 1.23456 or 9.87E6
  - Base 2: 2#101101#
- **Character:**
  - ‘A’, ‘Z’, ‘1’
- **Strings**
  - “Hello”, “101101”
- **Note**
  - 0 and ‘0’ are different
  - 2#101101# and “101101” are different
Objects

- A named item that hold a value of specific data type
- Four kinds of objects
  - Signal
  - Variable
  - Constant
  - File (cannot be synthesized)
- Related construct
  - Alias

Signal

- Declared in the architecture body's declaration section
- Signal declaration:
  `signal signal_name, signal_name, ... : data_type`
- Signal assignment:
  `signal_name <= projected_waveform;`
- Ports in entity declaration are considered as signals
- Can be interpreted as wires or “wires with memory” (i.e., FFs, latches etc.)

Variable

- Declared and used inside a process
- Variable declaration:
  `variable variable_name, ... : data_type`
- Variable assignment:
  `variable_name := value_expression;`
- Contains no “timing info” (immediate assignment)
- Used as in traditional PL: a “symbolic memory location” where a value can be stored and modified
- No direct hardware counterpart

Constant

- Value cannot be changed
- Constant declaration:
  `constant const_name, ... : data_type := value_expression`
- Used to enhance readability
  - E.g.,
    `constant BUS_WIDTH: integer := 32;`
    `constant BUS_BYTES: integer := BUS_WIDTH / 8;`

Alias

- Not a object
- Alternative name for an object
- Used to enhance readability
  - E.g.,
    `signal: std_logic_vector(15 downto 0);
    alias cp: std_logic_vector(15 downto 0) is word(15 downto 0);
    alias reg1: std_logic_vector(2 downto 0) is word(2 downto 0);
    alias reg2: std_logic_vector(2 downto 0) is word(5 downto 3);
    alias reg3: std_logic_vector(2 downto 0) is word(2 downto 0);`
4. Data type and operators

- Standard VHDL
- IEEE1164_std_logic package
- IEEE numeric_std package

Data type

- Definition of data type
  - A set of values that an object can assume.
  - A set of operations that can be performed on objects of this data type.
- VHDL is a strongly-typed language
  - An object can only be assigned with a value of its type
  - Only the operations defined with the data type can be performed on the object

Data types in standard VHDL

- integer:
  - Minimal range: \(-2^{31-1}\) to \(2^{31-1}\)
  - Two subtypes: natural, positive
- boolean: \(\text{false, true}\)
- bit: \('0', '1'\)
  - Not capable enough
- bit_vector: a one-dimensional array of bit

Operators in standard VHDL

<table>
<thead>
<tr>
<th>operator</th>
<th>description</th>
<th>data type of operand a</th>
<th>data type of operand b</th>
<th>data type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \times b)</td>
<td>multiplication</td>
<td>integer</td>
<td>boolean, bit, bit-vector</td>
<td>integer</td>
</tr>
<tr>
<td>(a + b)</td>
<td>addition</td>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>(a - b)</td>
<td>subtraction</td>
<td>integer</td>
<td>integer</td>
<td>integer</td>
</tr>
<tr>
<td>(a &amp; b)</td>
<td>and</td>
<td>boolean, bit, bit-vector</td>
<td>same as a</td>
<td>boolean</td>
</tr>
<tr>
<td>(a \or b)</td>
<td>or</td>
<td>boolean, bit, bit-vector</td>
<td>same as a</td>
<td>boolean</td>
</tr>
<tr>
<td>(a \xor b)</td>
<td>xor</td>
<td>boolean, bit, bit-vector</td>
<td>same as a</td>
<td>boolean</td>
</tr>
</tbody>
</table>

Overloaded operator

IEEE std_logic_1164 package

- Which standard VHDL operators can be applied to std_logic and std_logic_vector?
- Overloading: same operator of different data types
- Overloaded operators in std_logic_1164 package
### Type conversion function in std_logic_1164 package:

<table>
<thead>
<tr>
<th>function</th>
<th>data type of operand</th>
<th>data type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>to_bit(a)</td>
<td>std_logic</td>
<td>bit</td>
</tr>
<tr>
<td>to_std_logic(a)</td>
<td>bit</td>
<td>std_logic</td>
</tr>
<tr>
<td>to_bit_vector(a)</td>
<td>std_logic_vector</td>
<td>bit_vector</td>
</tr>
<tr>
<td>to_std_logic_vector(a)</td>
<td>bit_vector</td>
<td>std_logic_vector</td>
</tr>
</tbody>
</table>

### Operators over an array data type

#### Relational operators for array
- Operands must have the same element type but their lengths may differ.
- Two arrays are compared element by element, from the left most element.
- All following returns true:
  - "011"="011", "011">"010", "011">"00010", "0110">"011"

#### Concatenation operator (&)
- E.g.,
  - y <= "00" & a(7 downto 2);
  - y <= a(7) & a(7) & a(7 downto 2);
  - y <= a(1 downto 0) & a(7 downto 2);

### Array aggregate

- Aggregate is a VHDL construct to assign a value to an array-typed object.
- E.g.,
  - a <= "10100000";
  - a <= (7=>'1', 6=>'0', 5=>'1', 4=>'0', 3=>'0', 2=>'0', 1=>'0', 0=>'0');
  - a <= (7|5=>'1', others=>'0');
- E.g.,
  - a <= "00000000"
  - a <= (others=>'0');
  - a <= "01000000"

### IEEE numeric_std package

- How to infer arithmetic operators?
- In standard VHDL:
  - ```vhdl```
signal a, b, sum: integer;
  ```
  ... sum <= a + b;
  What's wrong with integer data type?
• IEEE numeric_std package: define integer as an array of elements of std_logic
• Two new data types: unsigned, signed
• The array interpreted as an unsigned or signed binary number
• E.g.,
  signal x, y: signed(15 downto 0);
• Need invoke package to use the data type
  library ieee;
  use ieee.std_logic_1164.all;
  use ieee.numeric_std.all;

Overloaded operators in IEEE numeric_std package

<table>
<thead>
<tr>
<th>overloaded operator</th>
<th>description</th>
<th>data type of operand a</th>
<th>data type of operand b</th>
<th>data type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs a</td>
<td>absolute value of a</td>
<td>signed</td>
<td>signed</td>
<td></td>
</tr>
<tr>
<td>a + b</td>
<td>sum of a and b</td>
<td>unsigned, natural</td>
<td>unsigned, natural</td>
<td>unsigned</td>
</tr>
<tr>
<td>a - b</td>
<td>difference of a and b</td>
<td>signed, integer</td>
<td>signed, integer</td>
<td>signed</td>
</tr>
<tr>
<td>a * b</td>
<td>product of a and b</td>
<td>boolean</td>
<td>boolean</td>
<td>boolean</td>
</tr>
</tbody>
</table>

New functions in IEEE numeric_std package

<table>
<thead>
<tr>
<th>function</th>
<th>description</th>
<th>data type of operand a</th>
<th>data type of operand b</th>
<th>data type of result</th>
</tr>
</thead>
<tbody>
<tr>
<td>shift_left(a, b)</td>
<td>shift left</td>
<td>unsigned, signed</td>
<td>natural</td>
<td>sum in a</td>
</tr>
<tr>
<td>shift_right(a, b)</td>
<td>shift right</td>
<td>unsigned, signed</td>
<td>natural</td>
<td>sum in a</td>
</tr>
<tr>
<td>rotate_left(a, b)</td>
<td>rotate left</td>
<td>unsigned, signed</td>
<td>natural</td>
<td>sum in a</td>
</tr>
<tr>
<td>rotate_right(a, b)</td>
<td>rotate right</td>
<td>unsigned, signed</td>
<td>natural</td>
<td>sum in a</td>
</tr>
<tr>
<td>reduce_sum(a, b)</td>
<td>reduce sum</td>
<td>unsigned, signed, signed, std_logic</td>
<td>natural</td>
<td>sum in a, boolean</td>
</tr>
<tr>
<td>to_integer(a)</td>
<td>convert to integer</td>
<td>unsigned, signed, natural</td>
<td>integer</td>
<td>signed</td>
</tr>
<tr>
<td>to_signed(a)</td>
<td>convert to signed</td>
<td>natural, integer</td>
<td>signed</td>
<td></td>
</tr>
</tbody>
</table>

Type conversion

• Std_logic_vector, unsigned, signed are defined as an array of element of std_logic
• They considered as three different data types in VHDL
• Type conversion between data types:
  – type conversion function
  – Type casting (for “closely related” data types)

Type conversion between number-related data types

<table>
<thead>
<tr>
<th>data type of a</th>
<th>to data type</th>
<th>conversion function / type casting</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned</td>
<td>std_logic_vector</td>
<td>std_logic_vector(a)</td>
</tr>
<tr>
<td>unsigned</td>
<td>unsigned</td>
<td>unsigned(a)</td>
</tr>
<tr>
<td>unsigned</td>
<td>std_logic_vector</td>
<td>std_logic_vector(a)</td>
</tr>
<tr>
<td>unsigned</td>
<td>std_logic_vector</td>
<td>std_logic_vector(a)</td>
</tr>
<tr>
<td>unsigned</td>
<td>integer</td>
<td>to_integer(a)</td>
</tr>
<tr>
<td>natural</td>
<td>to_unsigned(a, size)</td>
<td>to_unsigned(a, size)</td>
</tr>
<tr>
<td>integer</td>
<td>signed</td>
<td>to_signed(a, size)</td>
</tr>
</tbody>
</table>
• E.g.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u6, u7: unsigned(3 downto 0);
signal sg: signed(3 downto 0);
```

Ok

```vhdl
u3 <= u2 + u1;  -- ok, both operands unsigned
u4 <= u2 + 1;  -- ok, operands unsigned and natural
```

Wrong

```vhdl
u5 <= sg;  -- type mismatch
u6 <= 5;  -- type mismatch
```

Fix

```vhdl
u5 <= unsigned(sg);  -- type casting
u6 <= to_unsigned(5,4);  -- conversion function
```

Wrong

```vhdl
u7 <= sg + u1;  -- + undefined over the types
```

Fix

```vhdl
u7 <= unsigned(sg) + u1;  -- ok, but be careful
```

Wrong

```vhdl
s3 <= u3;  -- type mismatch
s4 <= 5;  -- type mismatch
```

Fix

```vhdl
s3 <= std_logic_vector(u3);  -- type casting
s4 <= std_logic_vector(to_unsigned(5,4));
```

Wrong

```vhdl
s5 <= s2 + s1;  -- + undefined over std_logic_vector
s6 <= s2 + 1;  -- + undefined
```

Fix

```vhdl
s5 <= std_logic_vector(unsigned(s2) + unsigned(s1));
s6 <= std_logic_vector(unsigned(s2) + 1);
```

Non-IEEE package

• Packagea by Synopsys
• std_logic_arith:
  – Similar to numeric_std
  – New data types: unsigned, signed
  – Details are different
• std_logic_unsigned/ std_logic_signed
  – Treat std_logic_vector as unsigned and signed numbers
  – i.e., overload std_logic_vector with arith operations

Software vendors frequently store them in ieee library:

• E.g.,

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_arith_unsigned.all;

signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);

s5 <= s2 + s1;  -- ok, + overloaded with std_logic_vector
s6 <= s2 + 1;  -- ok, + overloaded with std_logic_vector
```
• Only one of the std_logic_unsigned and std_logic_signed packages can be used
• The std_logic_unsigned/std_logic_signed packages beat the motivation behind a strongly-typed language
• Numeric_std is preferred