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

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
**Brief History of VHDL**

- June 1981: Woods Hole Workshop
- July 1983: contract awarded to develop VHDL
  - Intermetrics
  - IBM
  - Texas Instruments
- August 1985: VHDL Version 7.2 released

**Genesis of VHDL**

- 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

**VHDL**

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

  - VHDL is an acronym for Very High Speed Integrated Circuit Hardware Description Language

**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
Four 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

IEEE Extensions

- IEEE standard 1076.1 Analog and Mixed Signal Extensions (VHDL-AMS)
- IEEE standard 1076.2 VHDL Mathematical Packages
- IEEE standard 1076.3 Synthesis Packages
- IEEE standard 1076.4 VHDL Initiative Towards ASIC Libraries (VITAL)
- IEEE standard 1076.5 VHDL Register Transfer Level (RTL) Synthesis
- IEEE standard 1164 Multivalue Logic System for VHDL Model Interoperability
- IEEE standard 1029 VHDL Waveform and Vector Exchange to Support Design and Test Verification (WAVES)

Verilog

- Simpler and syntactically different
- C-like

- Gateway Design Automation Co., 1985
- Gateway acquired by Cadence in 1990
- IEEE Standard 1364-1995
- Early *de facto* standard for ASIC programming

- Programming language interface to allow connection to non-Verilog code

VHDL vs. Verilog

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

How to learn Verilog by yourself?
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         |
    | Database        |
    | DataBus         |
    | DATABUS         |

    are all equivalent

Naming and Labeling (2)

<table>
<thead>
<tr>
<th>General rules of thumb (according to VHDL-87)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. All names should start with an alphabet character (a-z or A-Z)</td>
</tr>
<tr>
<td>2. Use only alphabet characters (a-z or A-Z), digits (0-9) and underscore (_)</td>
</tr>
<tr>
<td>3. Do not use any punctuation or reserved characters within a name (!, ?, ., &amp; , +, - etc.)</td>
</tr>
<tr>
<td>4. Do not use two or more consecutive underscore characters (___) within a name (e.g., Sel___A is invalid)</td>
</tr>
<tr>
<td>5. All names and labels in a given entity and architecture must be unique</td>
</tr>
</tbody>
</table>

Valid or invalid?

<table>
<thead>
<tr>
<th>Names or labels</th>
</tr>
</thead>
<tbody>
<tr>
<td>7segment_display</td>
</tr>
<tr>
<td>A87372477424</td>
</tr>
<tr>
<td>Adder/Subtractor</td>
</tr>
<tr>
<td>/reset</td>
</tr>
<tr>
<td>And_or_gate</td>
</tr>
<tr>
<td>AND__OR__NOT</td>
</tr>
<tr>
<td>Kogge-Stone-Adder</td>
</tr>
<tr>
<td>Ripple&amp;Carry_Adder</td>
</tr>
<tr>
<td>My adder</td>
</tr>
</tbody>
</table>
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 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 lab instructors and myself. 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

Comments

• Explain Function of Module to Other Designers
• Explanatory, Not Just Restatement of Code
• Locate Close to Code Described
• Put near executable code, not just in a header
Example: NAND Gate

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]

LIBRARY decl;
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;

Design Entity

Design Entity - most basic building block of a design.

One entity can have many different architectures.

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

Driver resides outside the entity
Port Mode OUT

Entity

Port signal

Driver resides inside the entity

Output cannot be read within the entity

\[ z \leq x \leq c \]

Port Mode OUT (with extra signal)

Entity

Port signal

Driver resides inside the entity

Signal \( x \) can be read inside the entity

\[ z \leq x \leq c \leq x \]

Port Mode INOUT

Port signal

Entity

Signal can be read inside the entity

Driver may reside both inside and outside of the entity

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

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

**Library declarations - syntax**

```vhdl
LIBRARY library_name;
USE library_name.package_name.package_parts;
```
Fundamental parts of a library

Libraries

- **iei**
  - Specifies multi-level logic system, including STD_LOGIC, and STD_LOGIC_VECTOR data types
  - Need to be explicitly declared

- **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.
  - Visible by default

- **work**
  - Holds current designs after compilation

Libraries

**STD_LOGIC**

<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. Pull down</td>
</tr>
<tr>
<td>'H'</td>
<td>Weak (Weakly driven) 1. Pull up</td>
</tr>
<tr>
<td>'-'</td>
<td>Don't Care</td>
</tr>
</tbody>
</table>
More on STD_LOGIC Meanings (1)

Contention on the bus

More on STD_LOGIC Meanings (2)

More on STD_LOGIC Meanings (3)

Open Collector

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 comparison "1' = 'X" gives FALSE.

Resolving logic levels

<table>
<thead>
<tr>
<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>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>X</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>
</tr>
<tr>
<td>0</td>
<td>U</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>U</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</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>
</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>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>
</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>
</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>
</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**

**Signals**

```vhdl
SIGNAL a : STD_LOGIC;
SIGNAL b : STD_LOGIC_VECTOR(7 DOWNTO 0);
```

**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";
-- Binary base assumed by default
c <= B"0000";
-- Binary base explicitly specified
d <= X"AF67";
-- Hexadecimal base
e <= O"723";
-- Octal base
```

**Vectors and Concatenation**

```vhdl
SIGNAL a : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL b : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL c, d, e : STD_LOGIC_VECTOR(7 DOWNTO 0);
a <= "0000";
b <= "1111";
c <= a & b;
-- c = "00001111"
d <= '0' & "0001111";
-- d <= "00001111"
e <= '0' & '0' & '0' & '0' & '1' & '1' & '1' & '1';
-- e <= "00001111"
```

**Fixed Rotation in VHDL**

```vhdl
SIGNAL A :      STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL ArotL: STD_LOGIC_VECTOR(3 DOWNTO 0);
ArotL <=
```

**Fixed Shift in VHDL**

```vhdl
SIGNAL A :      STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL AshiftR: STD_LOGIC_VECTOR(3 DOWNTO 0);
AshiftR <=
```
VHDL Design Styles

Dataflow Architecture (xor3_gate)

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

Entity xor3_gate

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 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.

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;
Structural Architecture in VHDL 87

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;

xor2

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 93

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;

Behavioral Architecture (xor3 gate)

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;

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 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.
**Testbench Defined**

- Testbench = VHDL entity that applies stimuli (drives the inputs) to the Design Under Test (DUT) and (optionally) verifies expected outputs.
- The results can be viewed in a waveform window or written to a file.
- Since Testbench is written in VHDL, it is not restricted to a single simulation tool (portability).
- The same Testbench can be easily adapted to test different implementations (i.e. different architectures) of the same design.

**Simple Testbench**

**Possible sources of expected results used for comparison**

**Testbench Anatomy**

- ENTITY my_entity_tb IS 
  --TN entity has no ports
END my_entity_tb;
ARCHITECTURE behavioral OF tb IS
  --Local signals and constants
COMPONENT TestComp 
  --All Design Under Test component declarations
  PORT ( );
END COMPONENT;
BEGIN
  DUT: TestComp PORT MAP ( );
  testSequence: PROCESS
  -- Input stimuli
END PROCESS;
END behavioral;
Testbench for XOR3 (1)

LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY xor3_tb IS
END xor3_tb;

ARCHITECTURE behavioral OF xor3_tb IS
    COMPONENT xor3
        PORT(
            A : IN STD_LOGIC;
            B : IN STD_LOGIC;
            C : IN STD_LOGIC;
            R : OUT STD_LOGIC);
    END COMPONENT;

    -- Stimulus signals - signals mapped to the input and inout ports of tested entity
    SIGNAL test_vector : STD_LOGIC_VECTOR(2 DOWNTO 0);
    SIGNAL test_result : STD_LOGIC;

    BEGIN
        UUT : xor3 PORT MAP(
            A => test_vector(2),
            B => test_vector(1),
            C => test_vector(0),
            R => test_result);

        Testing : PROCESS
            BEGIN
                test_vector <= "000";
                WAIT FOR 10 ns;
                test_vector <= "001";
                WAIT FOR 10 ns;
                test_vector <= "010";
                WAIT FOR 10 ns;
                test_vector <= "011";
                WAIT FOR 10 ns;
                test_vector <= "100";
                WAIT FOR 10 ns;
                test_vector <= "101";
                WAIT FOR 10 ns;
                test_vector <= "110";
                WAIT FOR 10 ns;
                test_vector <= "111";
                WAIT FOR 10 ns;
            END PROCESS;
    END behavioral;

Testbench for XOR3 (2)

BEGIN
    PORT MAP(
        A IN test_vector(2),
        B IN test_vector(1),
        C IN test_vector(0),
        R OUT test_result);

    Testing : PROCESS
        BEGIN
            test_vector <= "000";
            WAIT FOR 10 ns;
            test_vector <= "001";
            WAIT FOR 10 ns;
            test_vector <= "010";
            WAIT FOR 10 ns;
            test_vector <= "011";
            WAIT FOR 10 ns;
            test_vector <= "100";
            WAIT FOR 10 ns;
            test_vector <= "101";
            WAIT FOR 10 ns;
            test_vector <= "110";
            WAIT FOR 10 ns;
            test_vector <= "111";
            WAIT FOR 10 ns;
        END PROCESS;

    Program control is passed to the first statement after BEGIN;

VHDL Design Styles

- **dataflow**: Concurrent statements
- **structural**: Components and interconnects
- **behavioral**: Sequential statements
  - Testbenches

What is a PROCESS?

- A process is a sequence of instructions referred to as sequential statements.
- A process can be given a unique name using an optional LABEL.
- This is followed by the keyword PROCESS.
- The keyword BEGIN is used to indicate the start of the process.
- All statements within the process are executed SEQUENTIALLY. Hence, order of statements is important.
- A process must end with the keywords END PROCESS.

Execution of statements in a PROCESS

- The execution of statements continues sequentially till the last statement in the process.
- After execution of the last statement, the control is again passed to the beginning of the process.
PROCESS with a WAIT Statement

- The last statement in the PROCESS is a WAIT instead of WAIT FOR 10 ns.
- This will cause the PROCESS to suspend indefinitely when the WAIT statement is executed.
- This form of WAIT can be used in a process included in a testbench when all possible combinations of inputs have been tested or a non-periodical signal has to be generated.

Testing: PROCESS
BEGIN
  test_vector<="00";
  WAIT FOR 10 ns;
  test_vector<="01";
  WAIT FOR 10 ns;
  test_vector<="10";
  WAIT FOR 10 ns;
  test_vector<="11";
  WAIT;
END PROCESS;

Program execution stops here.

WAIT FOR vs. WAIT

WAIT FOR: waveform will keep repeating itself forever

WAIT: waveform will keep its state after the last wait instruction.

Time values (physical literals) - Examples

<table>
<thead>
<tr>
<th>Unit</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Unit</td>
<td>femtoseconds (10^-15 seconds)</td>
</tr>
<tr>
<td>Derived Units</td>
<td></td>
</tr>
<tr>
<td>ps</td>
<td>picoseconds (10^-12 seconds)</td>
</tr>
<tr>
<td>ns</td>
<td>nanoseconds (10^-9 seconds)</td>
</tr>
<tr>
<td>us</td>
<td>microseconds (10^-6 seconds)</td>
</tr>
<tr>
<td>ms</td>
<td>miliseconds (10^-3 seconds)</td>
</tr>
<tr>
<td>sec</td>
<td>seconds</td>
</tr>
<tr>
<td>min</td>
<td>minutes (60 seconds)</td>
</tr>
<tr>
<td>hr</td>
<td>hours (3600 seconds)</td>
</tr>
</tbody>
</table>
Generating selected values of one input

```vhdl
SIGNAL test_vector : STD_LOGIC_VECTOR(2 downto 0);
BEGIN
    testing: PROCESS
    BEGIN
        test_vector <= "000";
        WAIT FOR 10 ns;
        test_vector <= "001";
        WAIT FOR 10 ns;
        test_vector <= "010";
        WAIT FOR 10 ns;
        test_vector <= "011";
        WAIT FOR 10 ns;
        test_vector <= "100";
        WAIT FOR 10 ns;
    END PROCESS;
END behavioral;
```

Generating all values of one input

```vhdl
SIGNAL test_vector : STD_LOGIC_VECTOR(3 downto 0) := "0000";
BEGIN
    testing: PROCESS
    BEGIN
        WAIT FOR 10 ns;
        test_vector <= test_vector + 1;
    end process TESTING;
END behavioral;
```

Arithmetic Operators in VHDL (1)

To use basic arithmetic operations involving `std_logic_vectors` you need to include the following library packages:

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
or
USE ieee.std_logic_signed.all;
```

Arithmetic Operators in VHDL (2)

You can use standard `+`, `-` operators to perform addition and subtraction:

```vhdl
signal A : STD_LOGIC_VECTOR(3 downto 0);
signal B : STD_LOGIC_VECTOR(3 downto 0);
signal C : STD_LOGIC_VECTOR(3 downto 0);
……
C <= A + B;
```

Different ways of performing the same operation

```vhdl
signal count: std_logic_vector(7 downto 0);
You can use:
count <= count + "00000001";
or
count <= count + 1;
or
count <= count + '1';
```

Different declarations for the same operator

```vhdl
Declarations in the package ieee.std_logic_unsigned:
function "+" ( L: std_logic_vector;
    R:std_logic_vector;
return std_logic_vector;
function "-" ( L: std_logic_vector;
    R:integer);
return std_logic_vector;
function "+" ( L: std_logic_vector;
    R:std_logic);
return std_logic_vector;
one of the following versions might be preferred:
function "+" ( L: std_logic_vector;
    R:std_logic_vector;
return std_logic_vector;
function "-" ( L: std_logic_vector;
    R:integer);
return std_logic_vector;
function "+" ( L: std_logic_vector;
    R:std_logic);
return std_logic_vector;
one of the following versions might be preferred:
function "+" ( L: std_logic_vector;
    R:std_logic_vector;
return std_logic_vector;
function "-" ( L: std_logic_vector;
    R:integer);
return std_logic_vector;
function "+" ( L: std_logic_vector;
    R:std_logic);
return std_logic_vector;
one of the following versions might be preferred:
function "+" ( L: std_logic_vector;
    R:std_logic_vector;
return std_logic_vector;
function "-" ( L: std_logic_vector;
    R:integer);
return std_logic_vector;
function "+" ( L: std_logic_vector;
    R:std_logic);
return std_logic_vector;
```
Operator overloading

- Operator overloading allows different argument types for a given operation (function)
- The VHDL tools resolve which of these functions to select based on the types of the inputs
- This selection is transparent to the user as long as the function has been defined for the given argument types.

Generating all possible values of two inputs

```vhdl
SIGNAL test_ab : STD_LOGIC_VECTOR(1 downto 0);
SIGNAL test_sel : STD_LOGIC_VECTOR(1 downto 0);
BEGIN
  double_loop : PROCESS
  BEGIN
    test_ab <= "00";
    test_sel <= "00";
    for I in 0 to 3 loop
      for J in 0 to 3 loop
        wait for 10 ns;
        test_ab <= test_ab + 1;
        test_sel <= test_sel + 1;
      end loop;
    end loop;
  END PROCESS;
END behavioral;
```

Generating periodical signals, such as clocks

```vhdl
CONSTANT clk1_period : TIME := 20 ns;
CONSTANT clk2_period : TIME := 200 ns;
SIGNAL clk1 : STD_LOGIC;
SIGNAL clk2 : STD_LOGIC := '0';
BEGIN
  clk1_generator: PROCESS
  clk1 <= '0';
  WAIT FOR clk1_period/2;
  clk1 <= '1';
  WAIT FOR clk1_period/2;
  END PROCESS;
  clk2 <= not clk2 after clk2_period/2;
END behavioral;
```

Generating one-time signals, such as resets

```vhdl
CONSTANT reset1_width : TIME := 100 ns;
CONSTANT reset2_width : TIME := 150 ns;
SIGNAL reset1 : STD_LOGIC;
SIGNAL reset2 : STD_LOGIC := '1';
BEGIN
  reset1_generator: PROCESS
  reset1 <= '1';
  WAIT FOR reset1_width;
  reset1 <= '0';
  WAIT;
  END PROCESS;
  reset2_generator: PROCESS
  WAIT FOR reset2_width;
  reset2 <= '0';
  WAIT;
  END PROCESS;
END behavioral;
```

Typical error

```vhdl
SIGNAL test_vector : STD_LOGIC_VECTOR(2 downto 0);
SIGNAL reset : STD_LOGIC;
BEGIN
  generator1: PROCESS
  reset <= '1';
  WAIT FOR 100 ns
  reset <= '0';
  test_vector <= "000";
  WAIT;
  END PROCESS;
  generator2: PROCESS
  WAIT FOR 200 ns
  test_vector <= "011";
  WAIT FOR 600 ns
  test_vector <= "011";
  END PROCESS;
END behavioral;
```