NAME: Junaid Girkar

SAP : 60004190057

Batch: SE Comps A-3

EXPERIMENT 10

VHDL CASE STUDY

AIM: To perform a case study on the VHDL programming language.

THEORY:

INTRODUCTION:

VHDL stands for VHSIC Hardware Description Language. VHSIC expands to Very High Speed Integrated Circuit. It's a programming language accustomed to model a digital system by dataflow, behavioural and structural form of modelling. This language was initially introduced in 1981 for the department of Defence (DoD) under the VHSIC program.

HISTORY:

In 1983, VHDL was originally developed at the dictation of the U.S. Department of Defense in order to document the behavior of the ASICs that provider firms were together with in instrumentation. the standard MIL-STD-454N[2] in requirement 64 in section 4.5.1 "ASIC documentation in VHDL" expressly needs documentation of "Microelectronic Devices" in VHDL.

The initial version of VHDL, designed to IEEE standard IEEE 1076-1987,[3] included a good vary of knowledge sorts, together with numerical (integer and real), logical (bit and boolean), character and time, plus arrays of bit called bit_vector and of character called string.

***STANDARDIZATION**:*

The IEEE Standard 1076 defines the VHSIC Hardware Description Language, or VHDL. it was originally developed under contract F33615-83-C-1003 from the United States Air Force awarded in 1983 to a team of Intermetrics, Inc. as language consultants and prime contractor, Texas Instruments as chip style consultants and IBM as computer-system style consultants. The language has undergone varied revisions and includes a form of sub-standards related to it that augment or extend it in important ways.

1076 was and continues to be a milestone within the style of electronic systems.

VHDL DESIGN AND SYNTAX:

VHDL was primarily based on Ada, and borrowed from it extensively in each syntax and ideas. This was then supplemented with hardware-specific ideas like multivalent logic, physical similarity, and an extended set of Boolean operators. VHDL may also index arrays in each ascending and descending order, whereas ADA (and most alternative programming languages as well) solely index in ascending order.

Most programming languages are, at heart, procedural — the computer executes one command after another in sequence. VHDL is totally different. it's a hardware language that describes a (real or simulated) organic structure. That structure is formed of an oversized range of modules, and each} module acts at a similar time as every alternative module.

So, among every module there's a procedural flow of directions that appears somewhat sort of a little, self-contained package program — with variables, management flows, conditionals, loops. every module has one or additional inputs along side one or additional outputs. The inputs are specified among a structure known as an entity, and therefore the self-contained logic is outlined in an architecture.

Consider the idea of an "AND gate" wherever we've 2 inputs and one output. If each the inputs are "on" (true, 1), then the output is "on"; otherwise, the output is "off." therefore using VHDL, we'd outline 2 inputs and one output. The accepted values of these inputs and outputs would be outlined in a std_logic module, that is imported like a library in a very regular programing language. The architecture would then outline the inner workings of our "AND gate" so it works as we just mentioned.

The std_logic module is a noteworthy hardware specific form of price. it's the same as the Boolean price gift in programming languages (one bit: true or false),

however it will have a variety of values, since it represents an actual electrical impulse in a physical system:

U: uninitialized. This signal hasn't been set yet.

X: unknown. not possible to work out this value/result.

0: logic 0

1: logic 1

Z: High impedance

W: Weak signal, cannot tell if it ought to be zero or one.

L: Weak signal that ought to in all probability go to zero

H: Weak signal that ought to in all probability go to one

-: don't care.

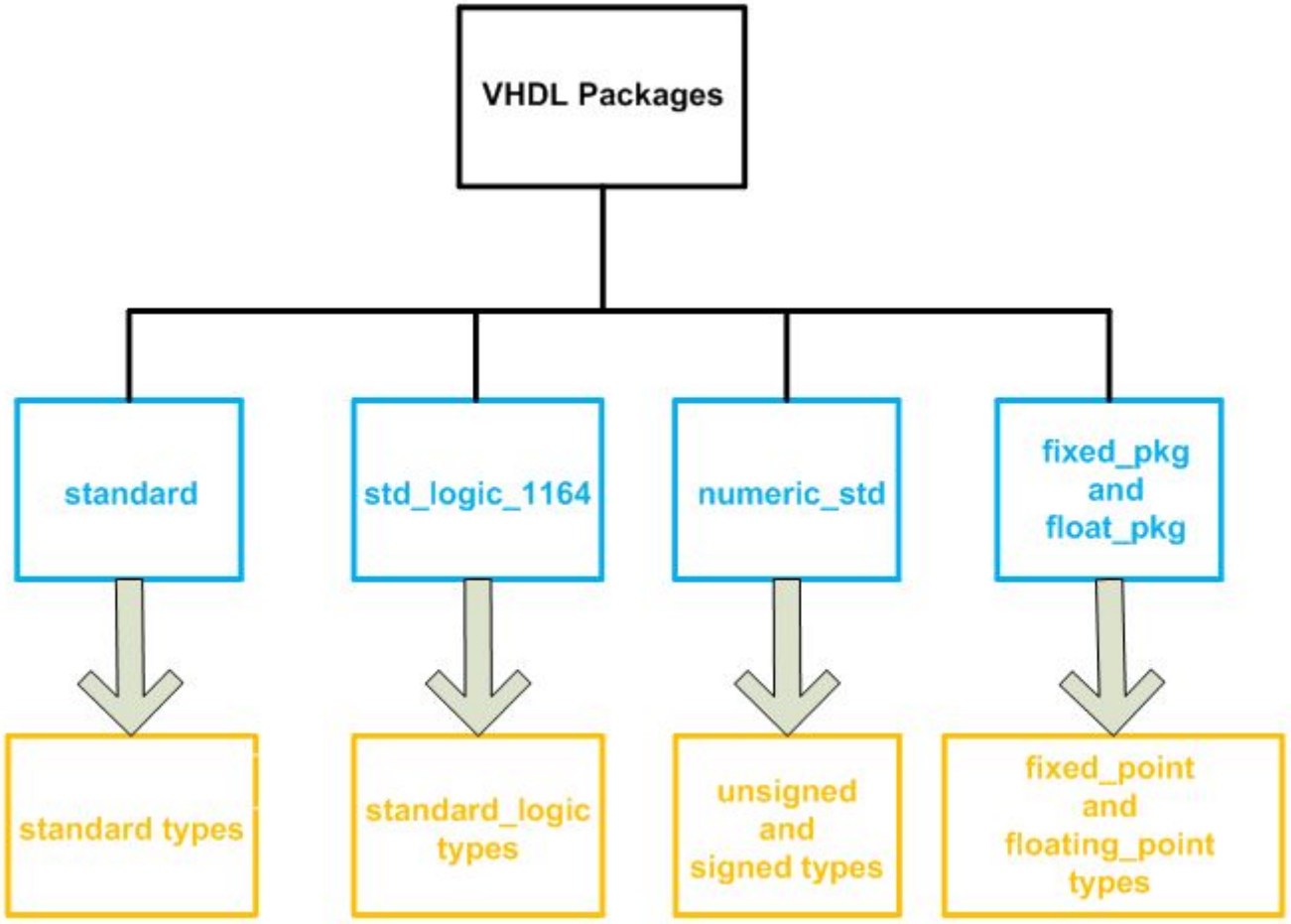
This is, in miniature, how a full VHDL style is constructed. Fairly straightforward, logically self-contained modules of I/O activity are designed up and connected to each to make computational machines capable of performing differing kinds of tasks. A VHDL style might describe a fully-functioning all-purpose pc, or it might write a single rule like the brute-force proof-of-work used for Bitcoin mining.

It's important to appreciate that a VHDL style isn't a program — it's not run or executed. sort of a blueprint, it defines an architecture. Once a design is completed it's typically simulated for testing in a software testbench, and then synthesized, which means that it's translated into a physical style which may be implemented on an actual chip or circuit board.

OBJECT AND DATA SORTS IN VHDL:

VHDL defines many kinds of objects. These include constants, variables, signals and files. {the types|the kinds|the categories} of values which may be allotted to those objects are known as data types. Same data types could also be allotted to totally different object types. for example, a constant, a variable and a signal can all have values that are of data type BIT. Declarations of objects include their object type as well because they are the type of values that they can acquire. For example: signal Enable: BIT;

DATA TYPES:



DESIGN ELEMENTS IN VHDL:

ENTITY: the basic design part in VHDL is named an 'ENTITY'. AN ENTITY represents a guide for a hardware block. It describes simply the outside view of a hardware module – particularly its interface with other modules in terms of input and output signals. The hardware block can be the entire design, a part of it or indeed an entire "test bench". A test bench includes the circuit being designed, blocks which apply test signals to it and those which monitor its output. The inner operation of the entity is described by an architecture related to it.

VHDL MODELLING STYLES: BEHAVIORAL, DATAFLOW, STRUCTURAL

An architecture can be written in one of 3 basic writing styles:

(1) Dataflow (2) behavioural (3) Structural.

The distinction between these styles is based on the type of concurrent statements used:

A dataflow architecture uses only concurrent signal assignment statements.

A behavioral architecture uses only process statements.

A structural architecture uses only component instantiation statements.

Instead of writing an design solely in one in all these styles, we are able to combine 2 or more, leading to a mixed style.

(1) Dataflow style of Modelling:

Dataflow style describes a system in terms of how data flows through the system. data dependencies within the description match those in a typical hardware implementation.

A dataflow description directly implies a corresponding gate-level implementation.

Dataflow descriptions carries with it one or more concurrent signal assignment statements.

E.g. Dataflow vogue half-adder description.

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

entity half_adder is
port (a, b: in std_logic;
sum, carry_out: out std_logic);
finish half_adder;

architecture dataflow of half_adder is
begin
sum <= a xor b;
carry_out <= a and b;
end dataflow;
```

The first assignment statement describes how input data flows from inputs a and b through an XOR function to create sum.

The second assignment statement describes how input data flows through an and function to produce carry_out.

Anytime there's an event on either input, the statements at the same time compute AN updated price for every output.

The coincident signal assignment statements during this description directly imply a hardware implementation consisting of an XOR circuit and an AND gate.

(2) behavioural (Sequential) style of Modelling:

A behavioural description describes a system's behavior or function in an algorithmic fashion.

Behavioral style is that the most abstract style. the description is abstract within the sense that it doesn't directly imply a specific gate-level implementation.

Behavioral style consists of 1 or additional method statements. every method statement is a single concurrent statement that itself contains one or additional sequential statements.

Sequential statements are executed sequentially by a simulator, the same as the execution of sequential statements in a very conventional programming language.

E.g. behavioural style half-adder description.

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

entity half_adder is
port (a, b: in std_logic;
sum, carry_out: out std_logic);
end half_adder;

architecture behavior of half_adder is
begin
ha: process (a, b)
begin
if a = '1' then
sum <= not b;
carry_out <= b;
else
sum <= b;
carry_out <= '0';
end if;
end process ha;
```

```
end behavior;
```

The entity declaration is the same as for the dataflow architecture. However, the architecture body different totally different. This architecture consists of a single process statement.

The process statement starts with the label ha followed by the keyword process. A label on a process is nonmandatory, however is helpful to differentiate processes in styles that contain multiple processes.

Following the keyword process is a list of signals in parentheses, called a sensitivity list. A sensitivity list enumerates exactly which signals cause the process to be executed. Whenever there's an event on a signal in a process's sensitivity list, the process is executed.

Between the second begin keyword and the keywords end process may be a sequential if statement. This if statement is executed whenever the process executes.

(3) Structural style of Modelling:

In structural style of modelling, an entity is described as a collection of interconnected components.

The top-level design entity's architecture describes the interconnection of lower-level design entities. each lower-level design entity will, in turn, be described as an interconnection of design entities at the next-lower level, and so on.

Structural style is most useful and efficient once when|a posh|a fancy} system is described as an interconnection of moderately complex design entities. This approach allows every design entity to be independently designed and verified before being used in the higher-level description.

E.g. Structural style half-adder description.

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

entity half_adder is -- Entity declaration for half adder
port (a, b: in std_logic;
sum, carry_out: out std_logic);
end half_adder;

architecture structure of half_adder is -- architecture body for half adder
```

```
component xor_gate -- xor component declaration
port (i1, i2: in std_logic;
o1: out std_logic);
end component;

component and_gate -- and component declaration
port (i1, i2: in std_logic;
o1: out std_logic);
end component;

begin
u1: xor gate port map (i1 => a, i2 => b, o1 => sum);
u2: and gate port map (i1 => a, i2 => b, o1 => carry_out);
-- we can also use positional Association
-- => u1: xor gate port map (a, b, sum);
-- => u2: and gate port map (a, b, carry_out);
end structure;
```

The half adder is described as an interconnection of an XOR gate design entity and an AND gate design entity.

Design entity half_adder describes how the xor gate and the AND gate are connected to implement a half adder. it's this top-level entity that has a structural style description.

In VHDL, a component is actually a placeholder for a design entity. A structural design that uses components simply specifies the interconnection of the components.

When components are used, every each be declared. A component declaration {is similar|is analogous|is comparable} to an entity declaration in that it provides a listing of the component's name and its ports. component declarations start with the keyword component.

A port map tells how a design entity is connected in the enclosing architecture.

In the statement part of the half-adder architecture are two component instantiation statements. each one creates an instance (copy) of a design entity. component instantiation statements require unique labels.

u1: xor gate port map (a, b, sum);

u2: and gate port map (a, b, carry_out);

(4) RTL Design:

A gate-level logic implementation is sometimes referred to as a register transfer level (RTL) implementation.

This level describes the logic in terms of registers and the Boolean equations for the combinational logic between the registers.

For a combinational system there are no registers and therefore the RTL logic consists solely of combinational logic.

WHAT IS A TESTBENCH?

The testbench is also an HDL code. We write testbenches to inject input sequences to input ports and read values from output ports of the module. The module (or electronic circuit) we are testing is called a DUT or a Device Under Test. Testing of a DUT is very similar to the physical lab testing of digital chips. There we use sequence generators for input and probe the output to a capturing device. And here we do the same thing virtually.

A testbench mainly has three purposes:

- To generate input sequences for DUT.
- After generation, to inject/apply those sequences to input ports.
- To observe output values and compares them with known values.

HOW DOES A TESTBENCH WORK?

A testbench is specific for a DUT. It contains a blank entity and its architecture. A testbench is also a VHDL program. It must have an entity. Also, the entity describes the input and output of the circuit that we are testing.

Inside the architecture of testbench, we declare a component which is actually our DUT. We also initialize some signals because we might need to read the values we've previously assigned. Thus, we use signals for internal calculations and in the end, assign the signal value to the port.

The next step is to generate a stimulus, or you may say sequences for inputs. We have two ways to generate an in-program stimulus.

- Using repetitive patterns
- Using vectors

Types of testbench in VHDL

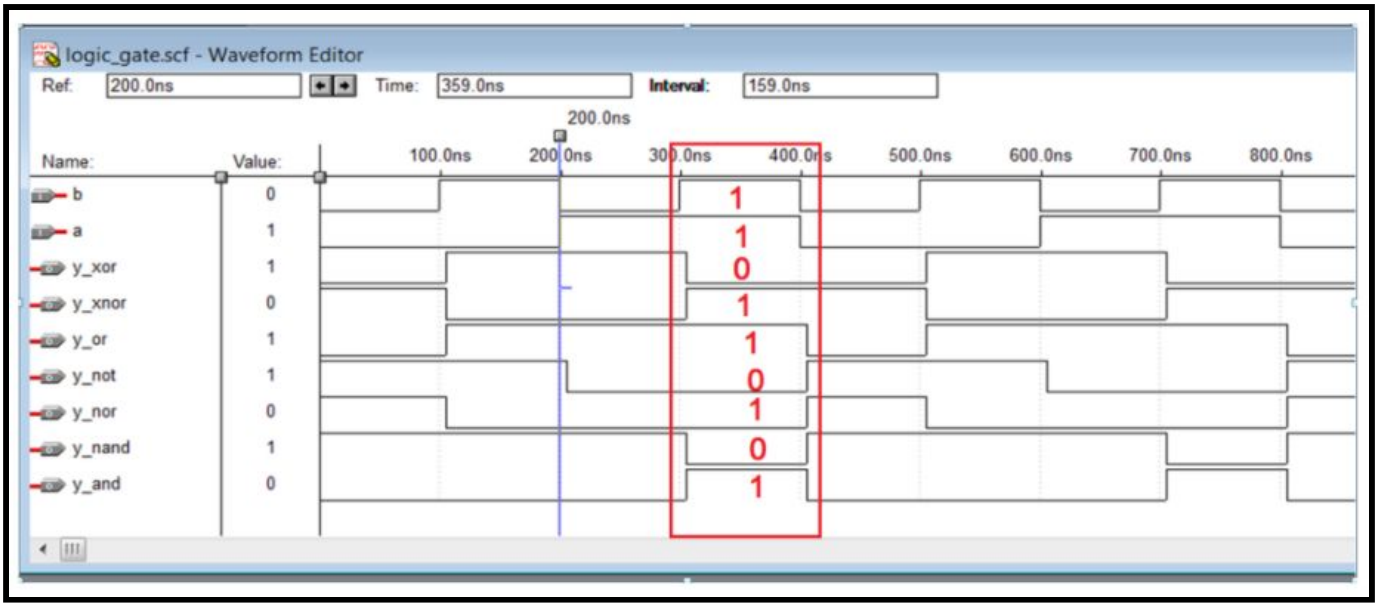
- Simple testbench
- Testbench with a process
- Infinite testbench
- Finite testbench

CONCLUSION:

Thus, we have studied the theory of the VHDL programming language in detail and we have also written code to actually see the output of the code and how VHDL programs work.

VHDL CODE FOR ALL LOGIC GATES:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity logic_gate is
     Port ( A,B : in std_logic;
                y_and,y_or,y_nand,y_nor,y_not,y_xor,y_xnor : out std_logic);
end logic_gate;
architecture all_gates of logic_gate is
begin
          y_and <= a and b;
          y_or <= a or b;
          y_nand <= a nand b;
          y_nor <= a nor b;
          y_not <= not a ;
          y_xor <= a xor b;
          y_xnor <= a xnor b;
end all_gates;
```



A	B	OR Gate op	AND gate op	NAND Gate op	NOR gate op	XOR gate op	XNOR gate op
0	0	0	0	1	1	0	1
0	1	1	0	1	0	1	0
1	0	1	0	1	0	1	0
1	1	1	1	0	0	0	1

A	NOT Gate op
1	0
0	1