MODULE -5

INTRODUCTION TO VHDL

5.1: Objectives

- To understand use of VHDL in design synthesis process
- To understand the entity and architecture of VHDL with simple designs
- To learn different data types, data objects and attributes on VHDL

5.2: Introduction

VHDL stands for **V**HSIC (Very High Speed Integrated Circuits) **H**ardware **D**escription **L**anguage. In the mid-1980's the U.S. Department of Defense and the IEEE sponsored the development of this hardware description language with the goal to develop very high-speed integrated circuit. It has become now one of industry's standard languages used to describe digital systems.

5.2.1 Why use VHDL?

The other widely used hardware description language is Verilog. Both are powerful languages that allow you to describe and simulate complex digital systems. A third HDL language is ABEL (Advanced Boolean Equation Language) which was specifically designed for Programmable Logic Devices (PLD). ABEL is less powerful than the other two languages and is less popular in industry

Power and Flexibility

VHDL not only gives you the power to describe circuits quickly using powerful language constructs but also permits other levels of design description including Boolean equations and structural netlists. Figure 5-1 illustrates four ways to describe a 2-bit comparator.

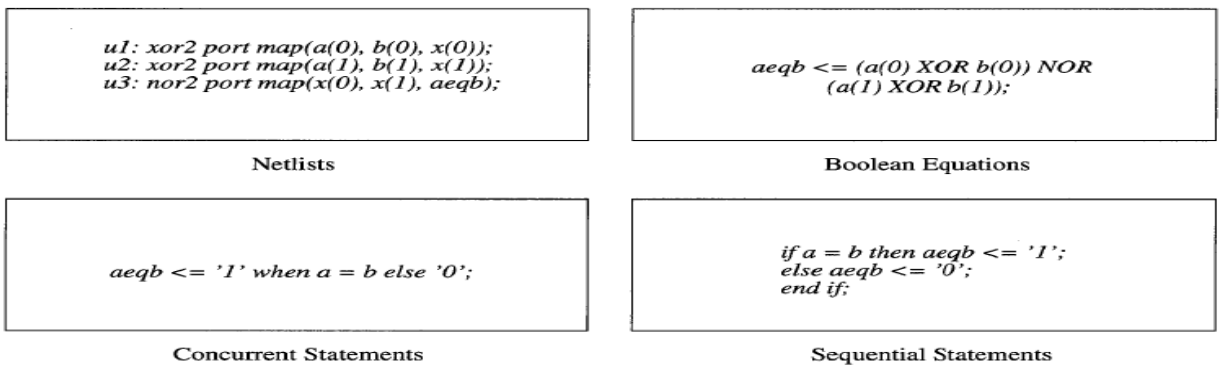

Figure 5-1 VHDL permits several classes of design description.

Device- Independent Design

VHDL permits you to create your design without having to first choose a device for implementation. With one design description, you can target many device architectures: You do not have to become intimately familiar with a device's architecture in order to optimize your design for resource utilization or performance. Instead, you can concentrate on creating your design.

Portability

VHDL's portability permits you to take the same design description that you used for synthesis and use it for simulation. For designs that require thousands of gates, being able to simulate the design description before synthesis and fitting (or place and route) can save you valuable time. Because VHDL is a standard, your design description can be taken from one simulator to another, one synthesis tool to another, and one platform to another. Figure 1-2 illustrates that the source code for a design can be used with any synthesis tool and that the design can be implemented in any device that is supported by the chosen synthesis tool.

5.2.2 VHDL versus conventional programming languages

(1) A hardware description language is inherently parallel, i.e. commands, which correspond to logic gates, are executed (computed) in parallel, as soon as a new input arrives.
(2) A HDL program mimics the behavior of a physical, usually digital, system.
(3) It also allows incorporation of timing specifications (gate delays) as well as to describe a system as an interconnection of different components

5.2.3. VHDL Application

VHDL is used mainly for the development of Application Specific Integrated Circuits (ASICs). Tools for the automatic transformation of VHDL code into a gate-level net list were developed already at an early point of time. This transformation is called synthesis and is an integral part of current design flows.

For the use with Field Programmable Gate Arrays (FPGAs) several problems exist. In the first step, Boolean equations are derived from the VHDL description, no matter, whether an ASIC or a FPGA is the target technology. But now, this Boolean code has to be partitioned into the configurable logic blocks (CLB) of the FPGA. This is more difficult than the mapping onto an ASIC library. Another big problem is the routing of the CLBs as the available resources for interconnections are the bottleneck of current FPGAs.

While synthesis tools cope pretty well with complex designs, they obtain usually only suboptimal results. Therefore, VHDL is hardly used for the design of low complexity Programmable Logic Devices (PLDs).

VHDL can be applied to model system behavior independently from the target technology. This is either useful to provide standard solutions, e.g. for micro controllers, error correction (de-)coders, etc, or behavioral models of microprocessors and RAM devices are used to simulate a new device in its target environment.

5.3 VHDL Program Structure

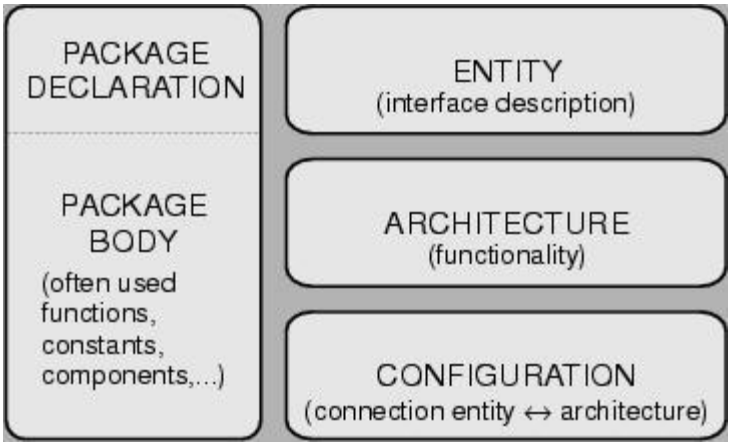

Figure 5-2 VHDL program structure.

entity entity-name **is**

    [port(interface-signal-declaration);]

end [entity] [entity-name];

architecture architecture-name **of** entity-name **is** [declarations]

begin

    architecture body

end [architecturee] [architecture-name];

5.4 Entity Block

An **entity block** is the beginning building block of a VHDL design. Each design has only one entity block which describes the interface signals into and out of the design unit. The syntax for an entity declaration is:

entity entity_name is

port (signal_name,signal_name : mode type;

signal_name,signal_name : mode type); end entity_name;

An entity block starts with the reserve word entity followed by the entity_name. Names and identifiers can contain letters, numbers, and the under score character, but must begin with an alpha character. Next is the reserved word is and then the port declarations. The indenting shown in the entity block syntax is used for documentation purposes only and is not required since VHDL is insensitive to white spaces.

A single PORT declaration is used to declare the interface signals for the entity and to assign MODE and data TYPE to them. If more than one signal of the same type is declared, each identifier name is separated by a comma. Identifiers are followed by a colon (:), mode and data type selections.

In general, there are five types of modes, but only three are frequently used. These three will be addressed here. They are in, out, and inout setting the signal flow direction for the ports as input, output, or bidirectional. Signal declarations of different mode or type are listed individually and separated by semicolons (;). The last signal declaration in a port statement and the port statement itself are terminated by a semicolon on the outside of the port's closing parenthesis.

The entity declaration is completed by using an **end** operator and the entity name. Optionally, you can also use an **end entity** statement**.** In VHDL, all statements are terminated by a semicolon.

Here is an example of an entity declaration for a set/reset (**SR**)

latch:

entity latch is

port (s,r : in std_logic;

q,nq : out std_logic); end latch;

The set/reset latch has input control bits **s** and **r** which are define d as single input bits and output bits **q** and **nq**. Notice that the declaration does not define the operation yet, just the interfacing input and output logic signals of the design. A design circuit's operation will be defined in the architecture block.

We can define a **literal constant** to be used within an entity with the **generic** declaration, which is placed before the port declaration within the entity block. Generic literals than can be used in port and other declarations. This makes it easier to modify or update designs. For instance if you declare a number of bit _vector bus signals, each eight bits in length, and at some future time you want to change them all to 16-bits, you would have to change each of the bit_vector range. However, by using a generic to define the range value, all you have to do is change the generic's value and the change will be reflected in each of the bit_vectors defined by that generic. The syntax to define a generic is:

generic (name : type := value);

The reserved word **generic** defines the declaration statement. This is followed by an identifier name for the generic and a colon. Next is the data type and a literal assignment value for the identifier. **:=** is the assignment operator that allows a literal value to be assigned to the generic identifier name. This operator is used for other assignment functions as we will see later.

For example, here is the code to define a bus width size using a generic literal.

entity my processor is generic (busWidth : integer := 7);

Presently, bus Width has the literal value of 7. This makes the documentation more descriptive for a vector type in a port declaration:

port(data_bus : in std_logic_vector (busWidth downto 0);

q-out : out std_logic_vector (busWidth downto 0));

In this example, data_ bus and q _out have a width of eight (8) bits (**7 down to 0**). When the design is updated to a larger bus size of sixteen (16)bits, the only change is to the literal assignment in the generic declaration from **7 to 15.**

5.5 Architecture Block

The architecture block defines how the entity operates. This may be described in many ways, two of which are most prevalent: STRUCTURE and DATA FLOW or BEHAVIOR formats. The BEHAVIOR approach describes the actual logic behavior of the circuit. This is generally in the form of a Boolean expression or process. The STRUCTURE approach defines how the entity is structured - what logic devices make up the circuit or design. The general syntax for the architecture block is:

architecture arch_name of entity_name is declarations;

begin

statements defining operation;

end arch_name;

example, we will use the set/reset NOR latch of figure 1. In VHDL code listings, -- (double dash) indicates a comment line used for documentation and ignored by the compiler.

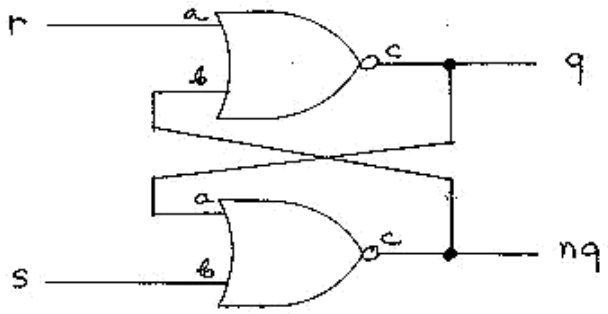

Figure 5-3 SR-Latch

```
library ieee;
use ieee.std_logic_1164.all;
                    -- entity block
entity latch is
                    -- interface signal declarations port (s,r : in std_logic;
q,nq : out std_logic);
end latch;
                    -- architecture block architecture flipflop of latch is begin
                    -- assignment statements q <= r nor nq;
nq <= s nor q;
end flipflop;
```

The first two lines imports the IEEE standard logic library std_logic_1164 which contains predefined logic functions and data types such as std_logic and std_logic_vector. The use statement determines which portions of a library file to use. In this example we are selecting all of the items in the 1164 library. The next block is the entity block which declares the latch's interface inputs, r and s and outputs q and nq. This is followed by the architecture block which begins by identifying itself with the name flipflop as a description of entity latch

Within the architecture block's body (designated by the begin reserved word) are two assignment statements. Signal assignment statements follow the general syntax of:

signal_identifier_name <= expression;

The **<=** symbol is the assignment operator for assigning a value to a signal. This differs from the **:=** assignment operator used to assign an initial literal value to generic identifier used earlier.

In our latch example, the state of the signal **q** is assigned the logic result of the nor function using input signals **r** and **nq**. The nor operator is defined in the IEEE std_logic_1164 library as a standard VHDL function to perform the nor logic operation. Through the use of Boolean expressions, the operation of the NOR latch's behavior is described and translated by a VHDL compiler into the hardware function appearing in figure 5.3.

5.5 Data Objects: Signals, Variables and Constants

A data object is created by an object declaration and has a value and type associated with it. An object can be a Constant, Variable or a Signal.

Signals can be considered wires in a schematic that can have a current value and future values, and that are a function of the signal assignment statements.

Variables and Constants are used to model the behavior of a circuit and are used in processes, procedures and functions.

Signal

Signals are declared with the following statement: **signal**

list_of_signal_names: type [:= initial value] ; **signal** SUM,

CARRY: std_logic;

signal CLOCK: bit;

signal TRIGGER: integer :=0;

signal DATA_BUS: bit_vector (0 to 7);

signal VALUE: integer **range** 0 **to** 100;

Signals are updated when their signal assignment statement is executed, after a certain delay, as illustrated below,

SUM <= (A **xor** B);

The result of A xor B is transferred to SUM after a delay called simulation Delta which is a infinitesimal small amount of time.

One can also specify multiple waveforms using multiple events as illustrated below,

signal wavefrm : std_logic;

wavefrm <= '0', '1' after 5ns, '0' after 10ns, '1' after 20 ns;

Constant

A constant can have a single value of a given type and cannot be changed during the simulation. A constant is declared as follows,

constant *list_of_name_of_constant*: type [:= initial value] ;

where the initial value is optional. Constants can be declared at the start of an architecture and can then be used anywhere within the architecture. Constants declared within a process can only be used inside that specific process.

constant RISE_FALL_TME: time := 2 ns;

constant DELAY1: time := 4 ns;

constant RISE_TIME, FALL_TIME: time:= 1 ns;

constant DATA_BUS: integer:= 16;

Variable

A variable can have a single value, as with a constant, but a variable can be updated using a variable assignment statement.

(1) The variable is updated without any delay as soon as the statement is executed. (2) Variables must be declared inside a process.

The variable declaration is as follows:

variable *list_of_variable_names*: type [:= initial value] ; A few examples follow:

variable CNTR_BIT: bit :=0;

variable VAR1: boolean :=FALSE;

variable SUM: integer **range** 0 **to** 256 :=16;

variable STS_BIT: bit_vector (7 **downto** 0);

The variable SUM, in the example above, is an integer that has a range from 0 to 256 with initial value of 16 at the start of the simulation.

A variable can be updated using a variable assignment statement such as

Variable_name := expression;

Example of a process using Variables: **architecture**

VAR **of** EXAMPLE is **signal** TRIGGER,

RESULT: integer := 0; **begin**

process

variable x1: integer :=1; **variable**

x2: integer :=2; **variable** x3:

integer :=3; **begin**

wait on TRIGGER;

x1 := x2;

x2 := x1 + x3;

x3 := x2;

RESULT <= x1 + x2 + x3;

end process;

end VAR;

Example of a process using Signals: **architecture**

SIGN **of** EXAMPLE is **signal** TRIGGER,

RESULT: integer := 0; **signal** s1: integer :=1;

signal s2: integer :=2; **signal**

s3: integer :=3; **begin**

process begin

wait on TRIGGER;

s1 <= s2;

s2 <= s1 + s3;

s3 <= s2;

RESULT <= s1 + s2 + s3;

end process;

end SIGN;

In the first case, the variables "x1, x2 and x3" are computed sequentially and their values updated instantaneously after the TRIGGER signal arrives. Next, the RESULT is computed using the new values of these variables. This results in the following values

(after a time TRIGGER): x1 = 2, x2 = 5 (ie2+3), x3= 5. Since RESULT is a signal it will be computed at the

time TRIGGER and updated at the time TRIGGER + Delta. Its value will be RESULT=12.

On the other hand, in the second example, the signals will be computed at the time TRIGGER. All of these signals are computed at the same time, using the old values of s1, s2 and s3. All the signals will be updated at Delta time after the TRIGGER has arrived. Thus the signals will have these values: s1= 2, s2= 4 (ie 1(old value of s1) +3), s3=2(old value of s2) and RESULT=6 ie (1+2+3).

5.6 Data types and Attributes

To define new type user must create a type declaration. A type declaration defines the **name of the type** and the **range of the type**. Type declarations are allowed in (i) Package declaration (ii) Entity Declaration (iii) Architecture Declaration (iv)Subprogram Declaration (v) Process Declaration

Enumerated Types:

An Enumerated type is a very powerful tool for abstract modeling. All of the values of an enumerated type are user defined. These values can be identifiers or single character literals.

An identifier is like a name, for examples: day, black, x Character literals are single characters enclosed in quotes, for example: ‘x’, ‘I’, ‘o’

Type Fourval **is** (‘x’, ‘o’, ‘I’, ‘z’);

Type color **is** (red, yello, blue, green, orange);

Type Instruction **is** (add, sub, lda, ldb, sta, stb, outa, xfr);

Real type example:

Type input level **is** range -10.0 to +10.0

Type probability **is** range 0.0 to 1.0;

Type W_Day **is** (MON, TUE, WED, THU, FRI, SAT, SUN);

type dollars **is range** 0 **to** 10;

variable day: W_Day;

variable Pkt_money:Dollars;

Case Day **is**

When TUE => pkt_money:=6;

When MON OR WED=> Pkt_money:=2;

When others => Pkt_money:=7;

End case;

Example for enumerated type - Simple Microprocessor model:

Package instr is

Type instruction is (add, sub, lda, ldb, sta, stb, outa, xfr); End

instr;

Use work.instr.all;

Entity mp **is**

PORT (instr: **in** Instruction; Addr: **in**

Integer;

Data: **inout** integer);

End mp;

Architecture mp **of** mp **is**

Begin

Process (instr)

type reg **is array**(0 to 255) **of** integer;

variable a,b: integer; **variable**

reg: reg; **begin**

case instr **is**

when lda => a:=data; **when** ldb

=> b:=data; **when** add =>

a:=a+b; **when** sub => a:=a-b;

when sta => reg(addr) := a; **when** stb

=> reg(addr):= b; **when** outa => data

:= a; **when** xfr => a:=b;

end case;

end process;

end mp;

Physical types:

These are used to represent real world physical qualities such as length, mass, time and current.

Type _____ is range _____ to _____

Units identifier;

{(identifier=physical literal;)} **end**

units identifier; **Examples:**

(1) **Type** resistance **is range** 0 to 1E9

units

ohms;

kohms = 1000ohms; Mohms =

1000kohms; **end units**;

(2) **Type** current **is range** 0 **to** 1E9

units

na;

ua = 1000na; ma =

1000ua; a = 1000ma;

end units;

Composite types consist of array and record types.

· Array types are groups of elements of same type

· Record allow the grouping of elements of different types

· Arrays are used for modeling linear structures such as ROM, RAM

· Records are useful for modeling data packets, instruction etc.

· A composite type can have a value belonging to either a scalar type, composite type or an access type.

Array Type:

Array type groups are one or more elements of the same type together as a single object. Each element of the array can be accessed by one or more array indices.

Type data-bus **is array** (0to 31) **of** BIT;

Variable x: data-bus;

Variable y: bit; Y :=x(0);

Y := x(15);

Type address_word **is array**(0 to 63) **of** BIT;

Type data_word **is array**(7 downto 0) **of** std_logic;

Type ROM **is array**(0 to 255) **of** data_word;

We can declare array objects of type mentioned above as follows:

Variable ROM_data: ROM;

Signal Address_bus: Address_word;

Signal word: data_word;

Elements of an array can be accessed by specifying the index values into the array. X<= Address_bus(25); transfers 26th element of array Address_bus to X.

Y := ROM_data(10)(5); transfers the value of 5th element in 10th row.

Multi dimentional array types may also be defined with two or more dimensions. The following example defines a two-dimensional array variable, which is a matrix of integers with four rows and three columns:

Type matrix4x3 **is array** (1 to 4, 1 to 3) **of** integer;

Variable matrixA: matrix4x3 := ((1,2,3), (4,5,6), (7,8,9), (10,11,12));

Variable m:integer;

The viable matrixA, will be initialized to

1 2 3

4 5 6

7 8 9

10 11 12

The array element matrixA(3,2) references the element in the third row and second column,

which has a value of 8.

m := matrixA(3,2); m gets the value 8

Record Type:

Record Types group objects of many types together as a single object. Each element of the record can be accessed by its field name. Record elements can include elements of any type including arrays and records. Elements of a record can be of the same type or different types.

Example:

Type optype **is** (add, sub, mpy, div, cmp);
Type instruction **is**
Record
Opcode : optype; Src :
integer;
Dst : integer; End
record;

5.7: Outcomes

After completion of the module the students are able to:

- Understand use of VHDL in design synthesis process
- Understand the entity and architecture of VHDL with simple designs
- Learn different data types, data objects and attributes on VHDL.

5.8 : Recommended Questions

1. Discuss the evolution of VHDL.
2. List and explain the advantages and benefits of using VHDL?
3. Explain the digital system synthesis using VHDL in detail.
4. Explain the components of VHDL program.
5. Define entity. Explain different types of ports used in VHDL entity.
6. Explain different description styles supported by VHDL with example.
7. Compare different description styles of VHDL with examples.
8. Explain the different data objects, data types of VHDL.
9. What are attributes? Explain with examples.
10. Differentiate between signal assignment and variable assignment statements.
11. Write the entity declaration 2-bit magnitude comparator.