PART-A

UNIT 1:

Introduction: Why HDL? , A Brief History of HDL, Structure of HDL Module, Operators, Data types, Types of Descriptions, simulation and synthesis, Brief comparison of VHDL and Verilog

6 Hours

UNIT 2:

Data –Flow Descriptions: Highlights of Data-Flow Descriptions, Structure of Data-Flow Description,DataType- Vectors

6 Hours

UNIT 3:

Behavioral Descriptions: Behavioral Description highlights, structure of HDL behavioral Description, The VHDL variable –Assignment Statement, sequential statements.

7 Hours

UNIT 4:

Structural Descriptions: Highlights of structural Description, Organization of the structural Descriptions, Binding, state Machines, Generate, Generic, and Parameter statements.

7 Hours

PART-B


Advanced HDL Descriptions: File Processing, Examples of File Processing

7 Hours

UNIT 6:

Mixed –Type Descriptions: Why Mixed-Type Description? VHDL User-Defined Types, VHDL Packages, Mixed-Type Description examples

6 Hours
UNIT 7:
Mixed Language Descriptions: Highlights of Mixed-Language Description, How to invoke One language from the Other, Mixed-language Description Examples, Limitations of Mixed-Language Description

7 Hours

UNIT 8:

6 Hours

TEXT BOOKS:
1. HDL Programming (VHDL and Verilog)- Nazeih M Botros- Dreamtech Press

REFERENCE BOOKS:
1. Verilog HDL –Samir Palnitkar-Pearson Education
2. VHDL –Douglas perry-Tata McGraw-Hill
3. A Verilog HDL Primer- J.Bhaskar – BS Publications
# INDEXSHEET

<table>
<thead>
<tr>
<th>SL.NO</th>
<th>TOPIC</th>
<th>PAGE NO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>University syllabus</td>
<td>1-2</td>
</tr>
<tr>
<td></td>
<td>UNIT – 1: Introduction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01 Assignment Questions</td>
<td>4-32</td>
</tr>
<tr>
<td></td>
<td>UNIT - 2: Data –Flow Descriptions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01 Assignment Questions</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>UNIT - 3: Behavioral Descriptions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01 Assignment Questions</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>UNIT - 4: Structural Descriptions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01 Assignment Questions</td>
<td>46-71</td>
</tr>
<tr>
<td></td>
<td>UNIT - 5: Procedures, Tasks, and Functions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01 Assignment Questions</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>UNIT - 6: Mixed –Type Descriptions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01 Assignment Questions</td>
<td>73-121</td>
</tr>
<tr>
<td></td>
<td>UNIT 7: Mixed –Language Descriptions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01 Assignment Questions</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>UNIT 8: Synthesis Basics</td>
<td></td>
</tr>
<tr>
<td></td>
<td>01 Assignment Questions</td>
<td>123-181</td>
</tr>
<tr>
<td></td>
<td></td>
<td>182</td>
</tr>
<tr>
<td></td>
<td></td>
<td>183-228</td>
</tr>
<tr>
<td></td>
<td></td>
<td>229</td>
</tr>
<tr>
<td></td>
<td></td>
<td>230-256</td>
</tr>
<tr>
<td></td>
<td></td>
<td>257</td>
</tr>
<tr>
<td></td>
<td></td>
<td>258-287</td>
</tr>
<tr>
<td></td>
<td></td>
<td>288</td>
</tr>
</tbody>
</table>
UNIT 1: INTRODUCTION

Syllabus of unit 1: Hours: 6
Why HDL?, A Brief History of HDL, Structure of HDL Module, Operators, Data types, Types of Descriptions, simulation and synthesis, Brief comparison of VHDL and Verilog

Recommended readings:

2. **Verilog HDL** - Samir Palnitkar-Pearson Education
4. **A Verilog HDL Primer** - J.Bhaskar – BS Publications
5. **Circuit Design with VHDL** - Volnei A.Pedroni-PHI
UNIT1: INTRODUCTION

Introduction to VHDL:

VHDL stands for VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. 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.

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.

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.

Levels of representation and abstraction
A digital system can be represented at different levels of abstraction [1]. This keeps the description and design of complex systems manageable. Figure 1 shows different levels of abstraction.

Figure 1: Levels of abstraction: Behavioral, Structural and Physical
The highest level of abstraction is the **behavioral** level that describes a system in terms of what it does (or how it behaves) rather than in terms of its components and interconnection between them. A behavioral description specifies the relationship between the input and output signals. This could be a Boolean expression or a more abstract description such as the Register Transfer or Algorithmic level.

As an example, let us consider a simple circuit that warns car passengers when the door is open or the seatbelt is not used whenever the car key is inserted in the ignition lock. At the behavioral level this could be expressed as,

\[
\text{Warning} = \text{Ignition\_on AND ( Door\_open OR Seatbelt\_off)}
\]

The **structural** level, on the other hand, describes a system as a collection of gates and components that are interconnected to perform a desired function. A structural description could be compared to a schematic of interconnected logic gates. It is a representation that is usually closer to the physical realization of a system. For the example above, the structural representation is shown in Figure 2 below.

![Figure 2: Structural representation of a “buzzer” circuit.](image)

VHDL allows to describe a digital system at the **structural** or the **behavioral** level. The behavioral level can be further divided into two kinds of styles: **Data flow** and **Sequential**. The dataflow representation describes how data moves through the system. This is typically done in terms of data flow between registers (Register Transfer level). The data flow model makes use of concurrent statements that are executed in parallel as soon as data arrives at the input. On the other hand, **sequential statements** are executed in the sequence that they are specified.

VHDL allows both **concurrent** and **sequential** signal assignments that will determine the manner in which they are executed.

**Mixed level** design consists both behavioral and structural design in one block diagram.

**Basic Structure of a VHDL file**

(a) **Entity**

A digital system in VHDL consists of a design **entity** that can contain other entities that are then considered components of the top-level entity. Each entity is modeled by an **entity declaration** and an **architecture body**. One can consider the entity declaration as the interface to the outside world that defines the input and output signals, while the architecture body contains the description of the entity and is composed of interconnected entities, processes and components, all operating concurrently, as schematically shown in Figure 3 below. In a typical design there will be many such entities connected together to perform the desired function.
A VHDL entity consisting of an interface (entity declaration) and a body (architectural description).

**a. Entity Declaration**

The entity declaration defines the NAME of the entity and lists the input and output ports. The general form is as follows,

```
entity NAME_OF_ENTITY is [ generic genericdeclarations ];
port (signal_names: mode type;
signal_names: mode type;
  .
signal_names: mode type);
end [NAME_OF_ENTITY ];
```

An entity always starts with the keyword `entity`, followed by its name and the keyword `is`. Next are the port declarations using the keyword `port`. An entity declaration always ends with the keyword `end`, optionally `[]` followed by the name of the entity.

![Block diagram of Full Adder](image)

**Example 1:**

```
entity FULLADDER is
  -- (After a double minus sign (-) the rest of
  -- the line is treated as a comment)
  --
  -- Interface description of FULLADDER
port ( x, y, Cl: in bit;
  S, Co: out bit);
end FULLADDER;
```

The module `FULLADDER` has five interface ports. Three of them are the input ports `x`, `y` and `Cl` indicated by the VHDL keyword `in`. The remaining two are the output ports `S` and `Co` indicated by `out`. The signals going through these ports are chosen to be of the type `bit`. The type `bit` consists of the two characters '0' and '1' and represents the binary logic values of the signals.

- The NAME_OF_ENTITY is a user-selected identifier.
- signal_names consists of a comma separated list of one or more user-selected identifiers that specify external interface signals.
- mode: is one of the reserved words to indicate the signal direction.
- **in** – indicates that the signal is an input
- **out** – indicates that the signal is an output of the entity whose value can only be read by other entities that use it.
- **buffer** – indicates that the signal is an output of the entity whose value can be read inside the entity’s architecture.
- **inout** – the signal can be an input or an output.

**Type**: a built-in or user-defined signal type. Examples of types are bit, bit_vector, Boolean, character, std_logic, and std_ulogic.

- **bit** – can have the value 0 and 1
- **bit_vector** – is a vector of bit values (e.g. bit_vector (0 to 7)
- **std_logic, std_ulogic, std_logic_vector, std_ulogic_vector**: can have 9 values to indicate the value and strength of a signal. Std_ulogic and std_logic are preferred over the bit or bit_vector types.
- **boolean** – can have the value TRUE and FALSE
- **integer** – can have a range of integer values
- **real** – can have a range of real values
- **character** – any printing character
- **time** – to indicate time

**Generic**: generic declarations are optional

**Example 2**:

```vhdl
entity AND3 is
port (in1, in2, in3: in std_logic;
out1: out std_logic);
end AND3;
```

The entity is called AND3 and has 3 input ports, in1, in2, in3 and one output port, out1. The name AND3 is an identifier. Inputs are denoted by the keyword **in**, and outputs by the keyword **out**. Since VHDL is a strongly typed language, each port has a defined type. In this case, we specified the std_logic type. This is the preferred type of digital signals. In contrast to the bit type that can only have the values ‘1’ and ‘0’, the std_logic and std_ulogic types can have nine values. This is important to describe a digital system accurately including the binary values 0 and 1, as well as the unknown value X, the uninitialized value U, “-” for don’t care, Z for high impedance, and several symbols to
indicate the signal strength (e.g. L for weak 0, H for weak 1, W for weak unknown - see section on Enumerated Types). The std_logic type is defined in the std_logic_1164 package of the IEEE library. The type defines the set of values an object can have. This has the advantage that it helps with the creation of models and helps reduce errors. For instance, if one tries to assign an illegal value to an object, the compiler will flag the error.

Example 3:

```
entity mux4_to_1 is
port (I0,I1,I2,I3: in std_logic;
S: in std_logic_vector(1 downto 0);
y: out std_logic);
end mux4_to_1;
```

Figure 5: Block Diagram of 4:1 Multiplexer

Example 4:

D Flip-Flop:

```
entity D_FF is
port (D,CLK, Q, Qb: in std_logic;
S, R: in std_logic);
end D_FF;
```

Figure 6: Block Diagram of D Flip Flop
**entity** dff_sr **is**
**port** (D,CLK,S,R: **in** std_logic;
Q,Qb: **out** std_logic);
**end** dff_sr;

**Architecture body**
The architecture body specifies how the circuit operates and how it is implemented. As discussed earlier, an entity or circuit can be specified in a variety of ways, such as behavioral, structural (interconnected components), or a combination of the above.
The architecture body looks as follows,

```
architecture architecture_name of NAME_OF_ENTITY **is**
-- Declarations
-- components declarations
-- signal declarations
-- constant declarations
-- function declarations
-- procedure declarations
-- type declarations
:
begin
-- Statements
:
end architecture_name;
```

The types of Architecture are:
(a) The behavioral Model
(b) Structure Model
(c) Mixed Model

(a) Behavioral model
The architecture body for the example of Figure 2, described at the behavioral level, is given below,
Example 1:
```
architecture behavioral of BUZZER **is**
begin
WARNING <= (not DOOR and IGNITION) or (not SBELT and IGNITION);
end behavioral;
```

The header line of the architecture body defines the architecture name, e.g. behavioral, and associates it with the entity, BUZZER. The architecture name can be any legal identifier. The main body of the architecture starts with the keyword begin and gives the Boolean expression of the function. We will see later that a behavioral model can be described in several other ways. The “<=” symbol represents an assignment operator and assigns the value of the expression on the right to the signal on the left. The architecture body ends with an end keyword followed by the architecture name.
Example 2:
The behavioral description of a 3 input AND gate is shown below.
```
entity AND3 **is**
**port** (in1, in2, in3: **in** std_logic;
out1: **out** std_logic);
**end** AND3;
architecture behavioral_2 of AND3 **is**
begin
out1 <= in1 and in2 and in3;
end behavioral_2;

Example 3:
entity XNOR2 is
port (A, B: in std_logic;
Z: out std_logic);
end XNOR2;
architecture behavioral_xnor of XNOR2 is
-- signal declaration (of internal signals X, Y)
signal X, Y: std_logic;
begin
X  <= A and B;
Y  <= (not A) and (not B);
Z  <= X or Y;
End behavioral_xnor;

Example 4:
SR Flip Flop:
entity SRFF is
port (S, R: in std_logic;
Q, Qb: out std_logic);
end SRFF;
arquceture behavioral_2 of SRFF is
begin
Q  <= NOT (S and Qb);
Qb <= NOT ( R and Q);
end behavioral_2;

The statements in the body of the architecture make use of logic operators. In addition, other types of operators including relational, shift, arithmetic are allowed as well.

Concurrence
The signal assignments in the above examples are concurrent statements. This implies that the statements are executed when one or more of the signals on the right hand side change their value (i.e, an event occurs on one of the signals).

In general, a change of the current value of a signal is called an event. For instance, when the input S (in SR FF) changes, the first expression gets evaluated, which changes the value of Q, change in Q in turn triggers second expression and evaluates Qb. Thus Q and Qb are updated concurrently.

There may be a propagation delay associated with this change. Digital systems are basically data-driven and an event which occurs on one signal will lead to an event on another signal, etc. Hence, the execution of the statements is determined by the flow of signal values. As a result, the order in which these statements are given does not matter (i.e., moving the statement for the output Z ahead of that for X and Y does not change the outcome). This is in contrast to conventional, software programs that execute the statements in a sequential or procedural manner.

Example 5
architecture CONCURRENT of FULLADDER is
begin
S  <= x xor y xor Ci after 5 ns;
Co <= (x and y) or (y and Ci) or (x and Ci) after 3 ns;
end CONCURRENT;
Two concurrent signal assignment statements describe the model of the entity FULLADDER.

The symbol <= indicates the signal assignment. This means that the value on the right side of the symbol is calculated and subsequently assigned to the signal on the left side. A concurrent signal assignment is executed whenever the value of a signal in the expression on the right side changes. Due to the fact that all signals used in this example are declared as ports in the entity declaration the arch_declarative_part remains empty.

Event Scheduling:
The mechanism of delaying the new value is called scheduling an event. In the above example, assignment to signals S and Co does not happen instantly. The after (keyword) clause delays the assignment of the new value to S and Co by 3 ns.

Example2:
architecture CONCURRENT_VERSION2 of FULLADDER is
signal PROD1, PROD2, PROD3 : bit;
begin
SUM <= A xor B xor C; -- statement 1
CARRY <= PROD1 or PROD2 or PROD3; -- statement 2
PROD1 <= A and B; -- statement 3
PROD2 <= B and C; -- statement 4
PROD3 <= A and C; -- statement 5
end CONCURRENT_VERSION2;

(a)Concurrent statement: In VHDL With select and When else statements are called as concurrent statements and they do not require Process statement

Example 1: VHDL code for 4:1 multiplexor
library ieee;
use ieee.std_logic_1164.all;
entity Mux is
port( I: in std_logic_vector(3 downto 0);
S: in std_logic_vector(1 downto 0);
y: out std_logic);
end Mux;
-- architecture using logic expression
architecture behv1 of Mux is
begin
y<= (not(s(0)) and not(s(1)) and I(0)) or(s(0) and not(s(1))
and I(1)) or (not(s(0)) and s(1) and I(2))or (s(0) and s(1) and I(3));
end behv1;
-- Architecture using when..else:
architecture behv2 of Mux is
begin
y <= I(0) when S="00" else
I(1) when S="01" else
I(2) when S="10" else
I(3) when S="11" else
'Z';
end behv2;
-- architecture using with select statement
architecture behv3 of Mux is
begin
with s select
y<=i(0) when "00",
i(1) when "01",
i(2) when "10",
i(3) when "11",
"Z" when others;
end behv3;
Note: "Z" high impedance state should be entered in capital Z
Example 2: SR flipflop using when else statement
entity SRFF is
port ( S, R: in bit;
Q, QB: inout bit);
end RSFF;
architecture beh of RSFF is
begin
Q <= Q when S= '0' and R = '0' else
'0' when S = '0' and R = '1' else
'1' when S = '1' and R = '0' else
'Z';
QB <= not(Q);
end beh;
The statement WHEN…..ELSE conditions are executed one at a time in sequential order
until the conditions of a statement are met. The first statement that matches the conditions
required assigns the value to the target signal. The target signal for this example is the
local signal Q. Depending on the values of signals S and R, the values Q,1,0 and Z are
assigned to Q.
If more than one statements conditions match, the first statement that matches does
the assign, and the other matching state.
In with ...select statement all the alternatives are checked simultaneously to find a
matching pattern. Therefore the with ... select must cover all possible values of the
selector
Structural Descriptions
A description style where different components of an architecture and their
interconnections are specified is known as a VHDL structural description. Initially, these
components are declared and then components' instances are generated or instantiated. At
the same time, signals are mapped to the components' ports in order to connect them like
wires in hardware. VHDL simulator handles component instantiations as concurrent
assignments.
Syntax:
component declaration:
component component_name
[generic (generic_list: type_name [:= expression] ;;
generic_list: type_name [:= expression]) ];
[port (signal_list: in|out|inout|buffer type_name ;;
signal_list: in|out|inout|buffer type_name) ];
end component;
component instantiation:

component_label: component_name port map (signal_mapping);

The mapping of ports to the connecting signals during the instantiation can be done through the positional notation. Alternatively, it may be done by using the named notation.

If one of the ports has no signal connected to it (this happens, for example, when there are unused outputs), a reserved word open may be used.

Example 1:

signal_mapping: declaration_name => signal_name.

entity RSFF is
port (SET, RESET: in bit;
Q, QBAR: inout bit);
end RSFF;
architecture NETLIST of RSFF is
component NAND2
port (A, B: in bit; C: out bit);
end component;
begi
U1: NAND2 port map (SET, QBAR, Q);
U2: NAND2 port map (Q, RESET, QBAR);
end NETLIST;

Figure 1: Schematic of SR FF using NAND Gate

The lines between the first and the keyword begin are a component declaration. It describes the interface of the entity nand_gate that we would like to use as a component in (or part of) this design. Between the begin and end keywords, the statements define component instances.

There is an important distinction between an entity, a component, and a component instance in VHDL.

The entity describes a design interface, the component describes the interface of an entity that will be used as an instance (or a sub-block), and the component instance is a distinct copy of the component that has been connected to other parts and signals.

In this example the component nand_gate has two inputs (A and B) and an output Q. There are two instances of the nand_gate component in this architecture corresponding to the two nand symbols in the schematic. The first instance refers to the top nand gate in
the schematic and the statement is called the **component instantiation statement**. The first word of the component instantiation statement (u1:nand2) gives instance a name, u1, and specifies that it is an instance of the component `nand_gate`. The next words describes how the component is connected to the set of the design using the **port map** clause. The **port map clause** specifies what signals of the design should be connected to the interface of the component in the same order as they are listed in the component declaration. The interface is specified in order as A, B and then C, so this instance connects `set to A`, `QBAR to B` and `Q to C`. This corresponds to the way the top gate in the schematic is connected. The second instance, named n2, connects `RESET to A`, `Q to A`, and `QBAR to C` of a different instance of the same `nand_gate` component in the same manner as shown in the schematic.

The structural description of a design is simply a textual description of a schematic. A list of components and there connections in any language is also called a netlist. The structural description of a design in VHDL is one of many means of specifying netlists.

**Example 2: Four Bit Adder – Illustrating a structural VHDL model:**

```vhdl
-- Example of a four bit adder
library ieee;
use ieee.std_logic_1164.all;
-- definition of a full adder
entity FULLadder is
  port (x, y, ci: in std_logic;
        s, co: out std_logic);
end FULLadder;
architecture fulladder_behav of FULLadder is
begin
  s  <=  x xor y xor ci ;
  co <= (x and y) or (x and ci)or(y and ci);
end fulladder_behav;
```

Figure 2: 4-bit Adder using four Full Adders.
-- 4-bit adder
library ieee;
use ieee.std_logic_1164.all;
entity FOURBITADD is
port (a, b: in std_logic_vector(3 downto 0);
      Cin : in std_logic;
      sum: out std_logic_vector (3 downto 0);
      Cout: out std_logic);
end FOURBITADD;
architecture fouradder_structure of FOURBITADD is
signal c: std_logic_vector (4 downto 0);
component FULLADDER
port (x, y, ci: in std_logic;
     s, co: out std_logic);
end component;
begin
FA0: FULLADDER
port map (a(0), b(0), Cin, sum(0), c(1));
FA1: FULLADDER
port map (a(1), b(1), C(1), sum(1), c(2));
FA2: FULLADDER
port map (a(2), b(2), C(2), sum(2), c(3));
FA3: FULLADDER
port map (a(3), b(3), C(3), sum(3), c(4));
Cout <= c(4);
end fouradder_structure;

We needed to define the internal signals c (4 downto 0) to indicate the nets that connect the output carry to the input carry of the next full adder. For the first input we used the input signal Cin. For the last carry we defined c (4) as an internal signal. We could not use the output signal Cout since VHDL does not allow the use of outputs as internal signals! For this reason we had to define the internal carry c(4) and assign c(4) to the output carry signal Cout.

(a) VHDL Operators
VHDL supports different classes of operators that operate on signals, variables and constants. The different classes of operators are summarized below.

<table>
<thead>
<tr>
<th>Class</th>
<th>and</th>
<th>or</th>
<th>nand</th>
<th>nor</th>
<th>xor</th>
<th>xnor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Logical operators</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Relational operators</td>
<td>=</td>
<td>/=</td>
<td>&lt;</td>
<td>&lt;=</td>
<td>&gt;</td>
<td>&gt;=</td>
</tr>
<tr>
<td>3. Shift operators</td>
<td>sll</td>
<td>srl</td>
<td>sla</td>
<td>sra</td>
<td>rol</td>
<td>ror</td>
</tr>
<tr>
<td>4. Addition operators</td>
<td>+</td>
<td></td>
<td>&amp;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Unary operators</td>
<td></td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Multiplying op.</td>
<td>*</td>
<td>/</td>
<td>mod</td>
<td></td>
<td>rem</td>
<td></td>
</tr>
<tr>
<td>7. Miscellaneous op.</td>
<td>**</td>
<td>abs</td>
<td>not</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The order of precedence is the highest for the operators of class 7, followed by class 6 with the lowest precedence for class 1. Unless parentheses are used, the operators with the highest precedence are applied first. Operators of the same class have the same precedence and are applied from left to right in an expression. As an example, consider the following std_ulogic_vectors, \( X = \text{‘010’} \), \( Y = \text{‘10’} \), and \( Z = \text{‘10101’} \). The expression
\[
\text{not } X \& Y \text{ xor } Z \text{ rol } 1
\]
is equivalent to \((\text{not } X) \& Y \text{ xor } (Z \text{ rol } 1) = (101) \& 10 \text{ xor } (01011) = (10110) \text{ xor } (01101) = 11101\). The xor is executed on a bit-per-bit basis.

1. Logic operators
The logic operators (and, or, nand, nor, xor and xnor) are defined for the “bit”, “boolean”, “std_logic” and “std_ulogic” types and their vectors. They are used to define Boolean logic expression or to perform bit-per-bit operations on arrays of bits. They give a result of the same type as the operand (Bit or Boolean). These operators can be applied to signals, variables and constants.

Notice that the nand and nor operators are not associative. One should use parentheses in a sequence of nand or nor operators to prevent a syntax error:
\( X \text{ nand } Y \text{ nand } Z \) will give a syntax error and should be written as \((X \text{ nand } Y) \text{ nand } Z\).

2. Relational operators
The relational operators test the relative values of two scalar types and give as result a Boolean output of “TRUE” or “FALSE”.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand Types</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equality</td>
<td>any type</td>
<td>Boolean</td>
</tr>
<tr>
<td>/=</td>
<td>Inequality</td>
<td>any type</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;</td>
<td>Smaller than</td>
<td>scalar or discrete array types</td>
<td>Boolean</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Smaller than or equal</td>
<td>scalar or discrete array types</td>
<td>Boolean</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>scalar or discrete array types</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Notice that symbol of the operator “\(<=\)” (smaller or equal to) is the same one as the assignment operator used to assign a value to a signal or variable. In the following examples the first “\(<=\)” symbol is the assignment operator. Some examples of relational operations are:

```vhdl
variable STS : Boolean;
constant A : integer :=24;
constant B_COUNT : integer :=32;
constant C : integer :=14;
STS <= (A < B_COUNT) ; -- will assign the value “TRUE” to STS
STS <= ((A >= B_COUNT) or (A > C)); -- will result in “TRUE”
STS <= (std_logic (‘1’, ‘0’, ‘1’) < std_logic(‘0’, ‘1’, ‘1’));--makes STS “FALSE”
type new_std_logic is (‘0’, ‘1’, ‘Z’, ‘-’);
```
variable A1: new_std_logic := '1';
variable A2: new_std_logic := 'Z';
STS <= (A1 < A2); will result in “TRUE” since ‘1’ occurs to the left of ‘Z’.
For discrete array types, the comparison is done on an element-per-element basis, starting from the left towards the right, as illustrated by the last two examples.

3. Shift operators
These operators perform a bit-wise shift or rotate operation on a one-dimensional array of elements of the type bit (or std_logic) or Boolean.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sll</strong></td>
<td>Shift left logical (fill right vacated bits with the 0)</td>
<td>Left: Any one-dimensional array type with elements of type bit or Boolean; Right: integer</td>
<td>Same as left type</td>
</tr>
<tr>
<td><strong>srl</strong></td>
<td>Shift right logical (fill left vacated bits with 0)</td>
<td>same as above</td>
<td>Same as left type</td>
</tr>
<tr>
<td><strong>sla</strong></td>
<td>Shift left arithmetic (fill right vacated bits with rightmost bit)</td>
<td>same as above</td>
<td>Same as left type</td>
</tr>
<tr>
<td><strong>sra</strong></td>
<td>Shift right arithmetic (fill left vacated bits with leftmost bit)</td>
<td>same as above</td>
<td>Same as left type</td>
</tr>
<tr>
<td><strong>rol</strong></td>
<td>Rotate left (circular)</td>
<td>same as above</td>
<td>Same as left type</td>
</tr>
<tr>
<td><strong>ror</strong></td>
<td>Rotate right (circular)</td>
<td>same as above</td>
<td>Same as left type</td>
</tr>
</tbody>
</table>

The operand is on the left of the operator and the number (integer) of shifts is on the right side of the operator. As an example,

variable NUM1: bit_vector := "10010110";
NUM1 srl 2;
will result in the number “00100101”.
When a negative integer is given, the opposite action occurs, i.e. a shift to the left will be a shift to the right. As an example
NUM1 srl -2 would be equivalent to NUM1 sll 2 and give the result “01011000”.
Other examples of shift operations are for the bit_vector A = “101001”
variable A: bit_vector := ”101001”;

Addition operators

The addition operators are used to perform arithmetic operation (addition and subtraction) on operands of any numeric type. The concatenation (&) operator is used to concatenate two vectors together to make a longer one. In order to use these operators one has to specify the ieee.std_logic_unsigned.all or std_logic_arith package package in addition to the ieee.std_logic_1164 package.

4. Addition operators

The addition operators are used to perform arithmetic operation (addition and subtraction) on operands of any numeric type. The concatenation (&) operator is used to concatenate two vectors together to make a longer one. In order to use these operators one has to specify the ieee.std_logic_unsigned.all or std_logic_arith package package in addition to the ieee.std_logic_1164 package.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Left Operand Type</th>
<th>Right Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Numeric type</td>
<td>Same as left operand</td>
<td>Same type</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Numeric type</td>
<td>Same as left operand</td>
<td>Same type</td>
</tr>
<tr>
<td>&amp;</td>
<td>Concatenation</td>
<td>Array or element type</td>
<td>Same as left operand</td>
<td>Same array type</td>
</tr>
</tbody>
</table>

An example of concatenation is the grouping of signals into a single bus [4].

```vhdl
signal MYBUS : std_logic_vector (15 downto 0);
signal STATUS : std_logic_vector (2 downto 0);
signal RW, CS1, CS2 : std_logic;
signal MDATA : std_logic_vector (0 to 9);
MYBUS <= STATUS & RW & CS1 & CS2 & MDATA;
```

Other examples are

```vhdl
MYARRAY (15 downto 0) <= "1111_1111" & MDATA (2 to 9);
NEWWORD <= "VHDL" & "93";
```

The first example results in filling up the first 8 leftmost bits of MYARRAY with 1’s and the rest with the 8 rightmost bits of MDATA. The last example results in an array of characters “VHDL93”.

Example:
Signal a: std_logic_vector (3 downto 0);
Signal b: std_logic_vector (3 downto 0);
Signal y:std_logic_vector (7 downto 0);
Y <= a & b;

5. Unary operators
The unary operators “+” and “−” are used to specify the sign of a numeric type.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Identity</td>
<td>Any numeric type</td>
<td>Same type</td>
</tr>
<tr>
<td>−</td>
<td>Negation</td>
<td>Any numeric type</td>
<td>Same type</td>
</tr>
</tbody>
</table>

6. Multiplying operators
The multiplying operators are used to perform mathematical functions on numeric types (integer or floating point).

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Left Operand Type</th>
<th>Right Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Any integer or floating point</td>
<td>Same type</td>
<td>Same type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any physical type</td>
<td>Integer or real type</td>
<td>Same as left</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Any integer or floating point</td>
<td>Any integer or floating point</td>
<td>Same type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any physical type</td>
<td>Any integer or real type</td>
<td>Same as left</td>
</tr>
<tr>
<td>mod</td>
<td>Modulus</td>
<td>Any integer type</td>
<td>Same type</td>
<td>Integer</td>
</tr>
<tr>
<td>rem</td>
<td>Remainder</td>
<td>Any integer type</td>
<td>Same type</td>
<td>Same type</td>
</tr>
</tbody>
</table>

The multiplication operator is also defined when one of the operands is a physical type and the other an integer or real type.
The remainder (rem) and modulus (mod) are defined as follows:

A \text{rem} B = A − (A/B) \times B (in which A/B is an integer)
A \text{mod} B = A − B \times N (in which N is an integer)
The result of the \text{rem} operator has the sign of its first operand while the result of the \text{mod}
operators has the sign of the second operand.
Some examples of these operators are given below.
11 \texttt{rem} 4 results in 3
(-11) \texttt{rem} 4 results in -3
9 \texttt{mod} 4 results in 1
7 \texttt{mod} (-4) results in $-1$ ($7 - 4*2 = -1$).

\textbf{7. Miscellaneous operators}
These are the absolute value and exponentation operators that can be applied to numeric types. The logical negation (not) results in the inverse polarity but the same type.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Left Operand Type</th>
<th>Right Operand Type</th>
<th>Result Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{x*x}</td>
<td>Exponentiation</td>
<td>Integer type</td>
<td>Integer type</td>
<td>Same as left</td>
</tr>
<tr>
<td>\texttt{abs}</td>
<td>Absolute value</td>
<td>Any numeric type</td>
<td>Integer type</td>
<td>Same as left</td>
</tr>
<tr>
<td>\texttt{not}</td>
<td>Logical negation</td>
<td>Any bit or Boolean type</td>
<td></td>
<td>Same type</td>
</tr>
</tbody>
</table>

\textbf{VHDL data types:}
To define new type user must create a type declaration. A type declaration defines the \textbf{name of the type} and the \textbf{range of the type}.
Type declarations are allowed in
(i) Package declaration (ii) Entity Declaration (iii) Architecture Declaration
(iv) Subprogram Declaration (v) Process Declaration

\textbf{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 \texttt{Fourval} \texttt{is} ('x', 'o', 'I', 'z');
Type \texttt{color} \texttt{is} (red, yello, blue, green, orange);
Type \texttt{Instruction} \texttt{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;
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:
Composite types consist of array and record types.
x Array types are groups of elements of same type
x Record allow the grouping of elements of different types
x Arrays are used for modeling linear structures such as ROM, RAM
x Records are useful for modeling data packets, instruction etc.
x 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 (0 to 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 dimensional 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:**
```verilog
Type optype is (add, sub, mpy, div, cmp);
Type instruction is Record
Opcode : optype;
Src : integer;
Dst : integer;
End record;
```

**Structure of Verilog module:**

```verilog
module module_name(signal_names)
Signal_type signal_names;
Signal_type signal_names;
Assign statements
Assign statements
Endmodule_name
```

**Verilog Ports:**
- Input: The port is only an input port. In any assignment statement, the port should appear only on the right hand side of the statement
- Output: The port is an output port. The port can appear on either side of the assignment statement.
- Inout: The port can be used as both an input & output. The inout represents a bidirectional bus.

**Verilog Value Set:**
- **0** represents low logic level or false condition
- **1** represents high logic level or true condition
- **x** represents unknown logic level
- **z** represents high impedance logic level

**Verilog Operators**
Operators in Verilog are the same as operators in programming languages. They take two values and compare or operate on them to yield a new result. Nearly all the operators in Verilog are exactly the same as the ones in the C programming language.
<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>*</td>
<td>Multiply</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>Unary plus</td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>Unary minus</td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>Equality</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>Inequality</td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>Logical Negation</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp;</td>
<td>Logical And</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift</td>
<td>&gt;&gt;</td>
<td>Right Shift</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;</td>
<td>Left Shift</td>
</tr>
<tr>
<td>Conditional</td>
<td>?</td>
<td>Conditional</td>
</tr>
<tr>
<td>Reduction</td>
<td>~</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td></td>
<td>~&amp;</td>
<td>Bitwise nand</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>~</td>
<td></td>
</tr>
<tr>
<td></td>
<td>^</td>
<td>Bitwise xor</td>
</tr>
</tbody>
</table>
Examples:
\[ x = y + z; \] //x will get the value of y added to the value of z
\[ x = 1 >> 6; \] //x will get the value of 1 shifted right by 5 positions
\[ x = !y \] //x will get the value of y inverted. If y is 1, x is 0 and vise versa

**Verilog Data Types:**
Nets (i)
an be thought as hardware wires driven by logic
Equal z when unconnected
Various types of nets
wire
wand (wired-AND)
wor (wired-OR)
tri (tri-state)
In following examples: Y is evaluated, automatically, every time A or B changes

Nets (ii)

```
wire Y; // declaration
assign Y = A & B;
```

```
wire Y; // declaration
assign Y = A;
assign Y = B;
```

```
wand Y; // declaration
assign Y = A;
assign Y = B;
```
```verilog
wor Y; // declaration
assign Y = A;
assign Y = B;

reg A, C; // declaration
// assignments are always done inside a procedure
A = 1;
C = A; // C gets the logical value 1
A = 0; // C is still 1
C = 0; // C is now 0

wire [3:0] busA;
reg [1:4] busB;
reg [1:0] busC;

// Vector assignment (by position!!)
busB[4] = busA[0];
```

- **Variables that store values**
- **Do not represent real hardware but ..**
- .. real hardware can be implemented with registers
- **Only one type: reg**
- **Register values are updated explicitly!!**
- **Vectors:**
  - **Represent buses**
  - **Left number is MS bit**
  - **Slice management**
    - busC[0] = busA[1];
  - **Vector assignment (by position!!)**

**Integer & Real Data Types:**
- **Declaration**
  - integer i, k;
  - real r;
- Use as registers (inside procedures)
i = 1; // assignments occur inside procedure
r = 2.9;
k = r; // k is rounded to 3
- Integers are not initialized!!
- Reals are initialized to 0.0

Parameters:
- Parameters represents global constants. They are declared by the predefined word parameter.

module comp_genr(X,Y,XgtY,XltY,XeqY);
parameter N = 3;
input [ N :0] X,Y;
output XgtY,XltY,XeqY;
wire [N:0] sum,Yb;

Time Data Type:
- Special data type for simulation time measuring
- Declaration
time my_time;
- Use inside procedure
my_time = $time; // get current sim time
- Simulation runs at simulation time, not real time

Arrays (i):
Syntax
integer count[1:5]; // 5 integers
reg var[-15:16]; // 32 1-bit regs
reg [7:0] mem[0:1023]; // 1024 8-bit regs
Accessing array elements
Entire element: mem[10] = 8'b 10101010;
Element subfield (needs temp storage):
reg [7:0] temp;
..temp = mem[10];
var[6] = temp[2];

Strings:
Implemented with regs:
reg [8*13:1] string_val; // can hold up to 13 chars
..string_val = "Hello Verilog";
string_val = "hello"; // MS Bytes are filled with 0
string_val = "I am overflowed"; // "I " is truncated

Escaped chars:
\n newline
\t tab
Styles(Types) of Descriptions:
- Behavioral Descriptions
- Structural Descriptions
- Switch – Level Descriptions
- Data – Flow Descriptions
- Mixed Type Descriptions

Behavioral Descriptions:
VHDL Behavioral description

```vhdl
entity half_add is
  port (I1, I2 : in bit; O1, O2 : out bit);
end half_add;
architecture behave_ex of half_add is
  --The architecture consists of a process construct
  begin
    process (I1, I2)
      --The above statement is process statement
      begin
        O1 <= I1 xor I2 after 10 ns;
        O2 <= I1 and I2 after 10 ns;
      end process;
  end behave_ex;
end behave_ex;
```

Verilog behavioral Description:

```verilog
module half_add (I1, I2, O1, O2);
  input I1, I2;
  output O1, O2;
  reg O1, O2;
  always @(I1, I2)
    //The above abatement is always
    //The module consists of always construct
    begin
      #10 O1 = I1 ^ I2;
      #10 O2 = I1 & I2;
    end
endmodule
```

VHDL Structural Descriptions:

```vhdl
entity system is
  port (a, b : in bit;
        sum, cout : out bit);
end system;
architecture struct_exple of system is
  component xor2
    --The above statement is a component statement
    port(I1, I2 : in bit;
       O1 : out bit);
```
end component;
component and2
    port(I1, I2 : in bit;
        O1 : out bit);
end component;
begin
    X1 : xor2 port map (a, b, sum);
    A1 : and2 port map (a, b, cout);
end struct_exple;

Verilog Structural Description:
module system(a, b, sum, cout);
    input a, b;
    output sum, cout;
    xor X1(sum, a, b);
    //The above statement is EXCLUSIVE-OR gate
    and a1(cout, a, b);
    //The above statement is AND gate
endmodule

Switch Level Descriptions:
VHDL Description:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Inverter is
    Port (y : out std_logic; a: in std_logic);
end Inverter;
architecture Invert_switch of Inverter is
    component nmos
        --nmos is one of the key words for switch-level.
        port (O1: out std_logic; I1, I2 : in std_logic);
    end component;
    component pmos
        --pmos is one of the key words for switch-level.
        port (O1: out std_logic ;I1, I2 : in std_logic);
    end component;
    for all: pmos use entity work. mos (pmos_behavioral);
    for all: nmos use entity work. mos (nmos_behavioral);
    --The above two statements are referring to a package mos
    constant vdd: std_logic := '1';
    constant gnd : std_logic:= '0';
    begin
        p1 : pmos port map (y, vdd, a);
        n1: nmos port map (y, gnd, a);
    end Invert_switch;
Verilog switch – Level Description:
module invert(y,a);
input a;
output y;
supply1 vdd;
supply0 gnd;
pmos p1(y, vdd, a);
nmos n1(y, gnd, a);
--The above two statement are using the two primitives pmos and nmos
endmodule

Data – Flow Descriptions:

VHDL Data – Flow Description:
entity halfadder is
port (a,b: in bit;
    s,c: out bit);
end halfadder;
architecture HA_DtFl of halfadder is
begin
    s <= a xor b;
    c <= a and b;
end HA_DtFl;

Verilog Data – Flow Description:
module halfadder (a,b,s,c);
    input a;
    input b;
    output s;
    output c;
    assign s = a ^ b;
    assign c = a & b;
endmodule

Comparision of VHDL & Verilog:

Data Types
VHDL: Types are in built in or the user can create and define them. User defined types give the user a tool to write the code effectively. VHDL supports multidimensional array and physical type.
Verilog: Verilog data types are simple & easy to use. There are no user defined types.

Ease of Learning
VHDL: Hard to learn because of its rigid type requirements.
Verilog: Easy to learn, Verilog users just write the module without worrying about what Library or package should be attached.

Libraries and Packages
VHDL: Libraries and packages can be attached to the standard VHDL package. Packages can include procedures and functions, & the package can be made available to any module that needs to use it.
Verilog: No concept of Libraries or packages in verilog.
• Operators
VHDL: An extensive set of operators is available in VHDL, but it does not have predefined unary operators.
Verilog: An extensive set of operators is also available in verilog. It also has predefined unary operators.
• Procedures and Tasks
VHDL: Concurrent procedure calls are allowed. This allows a function to be written inside the procedure’s body. This feature may contribute to an easier way to describe a complex system.
Verilog: Concurrent task calls are allowed. Functions are not allowed to be written in the task’s body.
ASSIGNMENT QUESTIONS

1) Explain entity and architecture with an example.
2) Explain structure of verilog module with an example.
3) Explain VHDL operators in detail.
4) Explain verilog operators in detail.
5) Explain how data types are classified in HDL. Mention the advantages of VHDL data types over verilog.
6) Mention the types of HDL descriptions. Explain dataflow and behavioral descriptions.
7) Describe different types of HDL description with suitable example.
8) Mention different styles (types) of descriptions. Explain mixed type and mixed language descriptions.
9) Compare VHDL and Verilog.
10) Write the result of all shift and rotate operations in VHDL after applying them to a 7 bit vector A = 1001010.
11) Explain composite and access data types with an example for each.
12) Discuss different logical operators used in HDL’s.
UNIT 2: DATA-FLOW DESCRIPTIONS

Syllabus of unit 2:  
Highlights of Data-Flow Descriptions, Structure of Data-Flow Description, DataType_Vectors.

Recommended readings:

1. **HDL Programming (VHDL and Verilog)**- Nazeih M.Botros- Dreamtech Press  
2. **Verilog HDL** –Samir Palnitkar-Pearson Education
4. **A Verilog HDL Primer**- J.Bhaskar – BS Publications
5. **Circuit Design with VHDL**- Volnei A.Pedroni-PHI
UNIT 2. DATA FLOW DESCRIPTIONS

Data flow is one type(style) of hardware description.

Facts
x Data – flow descriptions simulate the system by showing how the signal flows from system inputs to outputs.

x Signal – assignment statements are concurrent. At any simulation time, all signal-assignment statements that have an event are executed concurrently.

VHDL Description

entity system is
  port (I1, I2 : in bit; O1, O2 : out bit);
end;
architecture dtfl_ex of system is
begin
  O1 <= I1 and I2; -- statement 1.
  O2 <= I1 xor I2; -- statement 2.

--Statements 1 and 2 are signal-assignment statements
end dtfl_ex;

Verilog Description

module system (I1, I2, O1, O2);
  input I1, I2;
  output O1, O2;
/*by default all the above inputs and outputs are 1-bit signals.*/
  assign O1 = I1&I2; // statement 1
  assign O2 = I1^I2; // statement 2
/*Statements 1 and 2 are continuous signal-assignment statements*/
endmodule

Signal Declaration and Assignment Statements:

Syntax:
signal list_of_signal_names: type [ := initial value];

Examples:
signal SUM, CARRY: std_logic;
signal DATA_BUS: bit_vector (0 to 7);
signal VALUE: integer range 0 to 100;
  ■ Signals are updated after a delta delay.

Example:
SUM <= (A xor B);
  ■ The result of A xor B is transferred to SUM after a delay called simulation Delta which is an infinitesimal small amount of time.

Constant:

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

Examples:
constant RISE\_FALL\_TME: time := 2 ns;
constant DELAY1: time := 4 ns;

**HDL Code for Half Adder—VHDL and Verilog:**

**VHDL Half Adder Description**
entity halfadder is 
port ( 
a : in bit;
b : in bit;
s : out bit;
c : out bit);
end halfadder;
architecture HA\_DtFl of halfadder is 
begin 
s <= a xor b; -- This is a signal assignment statement.
c <= a and b; -- This is a signal assignment statement.
end HA\_DtFl;

**Verilog Half Adder Description**
module halfadder (a, b, s, c);
input a;
input b;
output s;
output c;
/*The default type of all inputs and outputs is a single bit. */
assign s = a ^ b; /* This is a signal assignment statement;
^is a bitwise xor logical operator. */
assign c = a & b; /* This is a signal assignment statement
& is a bitwise logical “and” operator */
endmodule
HDL Code of a 2x1 Multiplexer—VHDL and Verilog:

**VHDL 2x1 Multiplexer Description:**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux2x1 is
port (A, B, SEL, Gbar : in std_logic;
      Y : out std_logic);
end mux2x1;
architecture MUX_DF of mux2x1 is
signal S1, S2, S3, S4, S5 : std_logic;
Begin
  -- Assume 7 nanoseconds propagation delay
  -- for all and, or, and not.
  st1: Y <= S4 or S5 after 7 ns;
  st2: S4 <= A and S2 and S1 after 7 ns;
  st3: S5 <= B and S3 and S1 after 7 ns;
  st4: S2 <= not SEL after 7 ns;
  st5: S3 <= not S2 after 7 ns;
  st6: S1 <= not Gbar after 7 ns;
end MUX_DF;
```

**Verilog Description: 2x1 Multiplexer**

```verilog
module mux2x1 (A, B, SEL, Gbar, Y);
input A, B, SEL, Gbar;
```

Fig: 2x1 Multiplexer (a) Logic diagram (b) Logic symbol
output Y;
wire S1, S2, S3, S4, S5;

/* Assume 7 time units delay for all and, or, not.
In Verilog we cannot use specific time units,
such as nanoseconds. The delay here is
expressed in simulation screen units. */

assign #7 Y = S4 | S5;  //st1
assign #7 S4 = A & S2 & S1;  //st2
assign #7 S5 = B & S3 & S1;  //st3
assign #7 S2 = ~ SEL;  //st4
assign #7 S3 = ~ S2;  //st5
assign #7 S1 = ~ Gbar;  //st6
endmodule

HDL Code for a 2x2 Unsigned Combinational Array Multiplier—VHDL and Verilog:

VHDL 2x2 Unsigned Combinational Array Multiplier Description :

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mult_array is
  port (a, b : in std_logic_vector(1 downto 0);
P : out std_logic_vector (3 downto 0));
end mult_array;

architecture MULT_DF of mult_array is
begin
-- For simplicity propagation delay times are not considered
-- in this example.
P(0) <= a(0) and b(0);
P(1) <= (a(0) and b(1)) xor (a(1) and b(0));
P(2) <= (a(1) and b(1)) xor ((a(0) and b(1)) and (a(1) and b(0)));
P(3) <= (a(1) and b(1)) and ((a(0) and b(1)) and (a(1) and b(0)));
end architecture;
end MULT_DF;

Verilog 2x2 Unsigned Combinational Array Multiplier Description
module mult_arry (a, b, P);
input [1:0] a, b;
output [3:0] P;
/*For simplicity, propagation delay times are not considered in this example.*/
assign P[0] = a[0] & b[0];
assign P[1] = (a[0] & b[1]) ^ (a[1] & b[0]);
assign P[2] = (a[1] & b[1]) ^ ((a[0] & b[1]) & (a[1] & b[0]));
assign P[3] = (a[1] & b[1]) & ((a[0] & b[1]) & (a[1] & b[0]));
endmodule

HDL Code for a D-Latch—VHDL and Verilog:

VHDL D-Latch Description:

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_Latch is
  port (D, E : in std_logic;
        Q, Qbar : buffer std_logic);
-- Q and Qbar are declared as buffer because they act as
-- both input and output, they appear on the right and left
-- hand side of signal assignment statements. inout or
-- linkage could have been used instead of buffer.
end D_Latch;

architecture DL_DtFl of D_Latch is
constant Delay_EorD : Time := 9 ns;
constant Delay_inv : Time := 1 ns;
begin
  --Assume 9-ns propagation delay time between
  -- E or D and Qbar; and 1 ns between Qbar and Q.
  Qbar <= (D and E) nor (not E and Q) after Delay_EorD;
  Q <= not Qbar after Delay_inv;
end
```

Verilog D-Latch Description:
module D_latch (D, E, Q, Qbar);
input D, E;
output Q, Qbar;

/* Verilog treats the ports as internal ports, so Q and Qbar are not considered here as both input and output. If the port is connected externally as bidirectional, then we should use inout. */

time Delay_EorD = 9;
time Delay_inv = 1;
assign #Delay_EorD Qbar = ~(E & D) | (~E & Q);
assign #Delay_inv Q = ~ Qbar;
endmodule

HDL Code of a 2x2 Magnitude Comparator—VHDL and Verilog:

VHDL 2x2 Magnitude Comparator Description:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

description COMPR_2 is
    port (x, y : in std_logic_vector(1 downto 0); xgty, xlty : buffer std_logic; xeqy : out std_logic);
end COMPR_2;

architecture COMPR_DFL of COMPR_2 is
begin
  xgty <= (x(1) and not y(1)) or (x(0) and not y(1) and not y(0)) or x(0) and x(1) and not y(0);
  xlty <= (y(1) and not x(1)) or (not x(0) and y(0) and y(1)) or (not x(0) and not x(1) and y(0));
  xeqy <= xgty nor xlty;
end COMPR_DFL;

Verilog 2x2 Magnitude Comparator Description
module compr_2 (x, y, xgty, xlty, xeqy);
input [1:0] x, y;
output xgty, xlty, xeqy;
assign xgty = (x[1] & ~ y[1]) | (x[0] & ~ y[1]


& ~ y[0]) | (x[0] & x[1] & ~ y[0]);
assign xlt = (y[1] & ~ x[1]) | (~ x[0] & y[0] & y[1]) |
(~ x[0] & ~ x[1] & y[0]);
assign xeq = ~ (xgt | xlt);
endmodule

3-Bit Ripple-Carry Adder Case Study—VHDL and Verilog

VHDL 3-Bit Ripple-Carry Adder Case Study Description

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity adders_RL is
  port (x, y : in std_logic_vector (2 downto 0);
cin : in std_logic;
sum : out std_logic_vector (2 downto 0);
cout : out std_logic);
end adders_RL;

--Ⅰ. RIPPLE-CARRY ADDER

architecture RCarry_DtFl of adders_RL is
--Assume 4.0-ns propagation delay for all gates.
signal c0, c1 : std_logic;
constant delay_gt : time := 4 ns;

begin
sum(0) <= (x(0) xor y(0)) xor cin after 2*delay_gt;

--Treat the above statement as two 2-input XOR.
sum(1) <= (x(1) xor y(1)) xor c0 after 2*delay_gt;

--Treat the above statement as two 2-input XOR.

Verilog 3-Bit Ripple-Carry Adder Case Study Description

module adr_rcla (x, y, cin, sum, cout);
input [2:0] x, y;
input cin;
output [2:0] sum;
output cout;

// I. RIPPLE CARRY ADDER
wire c0, c1;
time delay_gt = 4;
//Assume 4.0-ns propagation delay for all gates.

assign #(2*delay_gt) sum[0] = (x[0] ^ y[0]) ^ cin;
//Treat the above statement as two 2-input XOR.
assign #(2*delay_gt) sum[1] = (x[1] ^ y[1]) ^ c0;
//Treat the above statement as two 2-input XOR.
//Treat the above statement as two 2-input XOR.
assign #(2*delay_gt) c0 = (x[0] & y[0]) | (x[0] & cin) | (y[0] & cin);
assign #(2*delay_gt) c1 = (x[1] & y[1]) | (x[1] & c0) | (y[1] & c0);
endmodule
3-Bit Carry-Lookahead Adder Case Study—VHDL and Verilog

VHDL 3-Bit Carry-Lookahead Adder Case Study Description
--II. CARRY-LOOKAHEAD ADDER

architecture lkh_DtFl of adders_RL is
  --Assume 4.0-ns propagation delay for all gates
  --including a 3-input xor.
signal c0, c1 : std_logic;
signal p, g : std_logic_vector (2 downto 0);
constant delay_gt : time := 4 ns;
begins

g(0) <= x(0) and y(0) after delay_gt;
g(1) <= x(1) and y(1) after delay_gt;
g(2) <= x(2) and y(2) after delay_gt;
p(0) <= x(0) or y(0) after delay_gt;
p(1) <= x(1) or y(1) after delay_gt;
p(2) <= x(2) or y(2) after delay_gt;
c0 <= g(0) or (p(0) and cin) after 2*delay_gt;

  c1 <= g(1) or (p(1) and g(0)) or (p(1) and p(0)
  and cin) after 2*delay_gt;
cout <= g(2) or (p(2) and g(1)) or (p(2) and p(1)
  and g(0)) or
  (p(2) and p(1) and p(0) and cin) after 2*delay_gt;

  sum(0) <= (p(0) xor g(0)) xor cin after delay_gt;
  sum(1) <= (p(1) xor g(1)) xor c0 after delay_gt;
  sum(2) <= (p(2) xor g(2)) xor c1 after delay_gt;
end lkh_DtFl;
Verilog 3-Bit Carry-Lookahead Adder Case Study Description

// II. CARRY-LOOKAHEAD ADDER
module lkahd_adder (x, y, cin, sum, cout);
input [2:0] x, y;
input cin;
output [2:0] sum;
output cout;
/*Assume 4.0-ns propagation delay for all gates
   including a 3-input xor.*/
wire c0, c1; wire
[2:0] p, g; time
delay_gt = 4;
assign #delay_gt g[0] = x[0] & y[0];
assign #delay_gt g[1] = x[1] & y[1];
assign #delay_gt p[0] = x[0] | y[0];
assign #delay_gt p[1] = x[1] | y[1];
assign #(2*delay_gt) c0 = g[0] | (p[0] & cin);
assign #(2*delay_gt) c1 = g[1] | (p[1] & g[0]) | (p[1] & p[0] & cin);
assign #delay_gt sum[0] = (p[0] ^ g[0]) ^ cin;
assign #delay_gt sum[1] = (p[1] ^ g[1]) ^ c0;
   assign #delay_gt sum[2] = (p[2] ^ g[2]) ^ c1;
endmodule
ASSIGNMENT QUESTIONS

1) With illustrations briefly discuss
   i) Signal declaration & assignment statements
   ii) Concurrent signal assignment statements &
   iii) Constant declaration & assignment statements.
2) Explain how an object that has a width of more than 1 bit is declared in HDL using vector data types. Give examples.
3) Explain signal declaration & signal assignment statements with relevant examples.
4) Write a data – flow description (in both VHDL & Verilog) for a full adder with active high enable.
5) Write HDL codes for 2X2 bit combinational array multiplier.
6) How do you assign delay to a signal assignment statement? Explain with an example in VHDL & verilog
7) What is a vector? Give an example for VHDL & verilog vector data types.
8) With the help of a truth table and K – maps write Boolean expression for a 2-bit magnitude comparator, write VHDL/verilog code
9) What are the data types available in VHDL?
10) Write a HDL Code of a 2x1 Multiplexer
UNIT 3: BEHAVIORAL DESCRIPTIONS

Syllabus of unit 3: Hours: 7
Behavioral Description highlights, structure of HDL behavioral Description, The VHDL variable – Assignment Statement, sequential statements.

Recommended readings:

1. **HDL Programming (VHDL and Verilog)**- Nazeih M. Botros- Dreamtech Press
2. **Verilog HDL** – Samir Palnitkar-Pearson Education
4. **A Verilog HDL Primer**- J. Bhaskar – BS Publications
5. **Circuit Design with VHDL**- Volnei A. Pedroni- PHI
UNIT 3. BEHAVIORAL DESCRIPTIONS

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:

\[
\text{signal list_of_signal_names: type [ := initial value] ;}
\]

\[
\text{signal SUM, CARRY: std_logic;}
\]

\[
\text{signal CLOCK: bit;}
\]

\[
\text{signal TRIGGER: integer :=0;}
\]

\[
\text{signal DATA_BUS: bit_vector (0 to 7);}
\]

\[
\text{signal VALUE: integer range 0 to 100;}
\]

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

\[
\text{SUM <= (A xor B);}
\]

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

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

\[
\text{signal wavefrm : std_logic;}
\]

\[
\text{wavefrm <= '0', '1' after 5ns, '0' after 10ns, '1' after 20 ns;}
\]

**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,

\[
\text{constant list_of_name_of_constant: type [ := initial value] ;}
\]

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.

\[
\text{constant RISE_FALL_TME: time := 2 ns;}
\]

\[
\text{constant DELAY1: time := 4 ns;}
\]

\[
\text{constant RISE_TIME, FALL_TIME: time:= 1 ns;}
\]

\[
\text{constant DATA_BUS: integer:= 16;}
\]

**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:

\[
\text{variable list_of_variable_names: type [ := initial value] ;}
\]

A few examples follow:

\[
\text{variable CNTR_BIT: bit :=0;}
\]

\[
\text{variable VAR1: boolean :=FALSE;}
\]
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)

**Comparison between Signal and Variable:**

1. Signal is updated after a certain delay when its signal assignment statement is executed.
   
   The variable is updated as soon as the statement is executed without any delay.

2. Variables take less memory while signals need more memory as they need more information to allow for scheduling and signal attributes.

3. Signals are declared in entity or architecture using <= symbol where as variables are declared inside process or functions using := symbol.

4. Signals have attributes and variables do not have attributes.

**Sequential Statements**

There are several statements that may only be used in the body of a process. These statements are called *sequential statements* because they are executed sequentially. That is, one after the other as they appear in the design from the top of the process body to the bottom.

Sequential behavioral descriptions are based on the process environment To ensure that simulation time can move forward every process must provide a means to get suspended. Thus, a process is constantly switching between the two states: the execution phase in which the process is active and the statements within this process are executed, and the suspended state.

The change of state is controlled by two mutually exclusive implementations:

· With a list of signals in such a manner that an event on one of these signals invokes a process. This can be compared with the mechanism used in conjunction with concurrent signal assignment statements. There, the statement is executed whenever a signal on the right side of the assignment operator <= changes its value. In case of a process, it becomes active by an event on one or more signal belonging to the sensitivity list. All statements between the keywords begin and end process are then executed sequentially.

   ```
   process (sensitivity_list)
   [proc_declarativ_part]
   begin
   [sequential_statement_part]
   end process [proc_label];
   ```

   The sensitivity_list is a list of signal names within round brackets, for example (A, B, C).

   x Process without sensitivity list must contain wait statement. With wait statements, the process is executed until it reaches a wait statement. At this instance it gets explicitly suspended. The statements within the process are handled like an endless loop which is suspended for some time by a wait statement.

   Syntax:

   ```
   process
   [proc_declarativ_part]
   ```
begin
[sequential_statements]
wait ...; -- at least one wait statement
[sequential_statements]
end process [proc_label];

The structure of a process statement is similar to the structure of an architecture.

In the proc_declarativ_part various types, constants and variables can be declared; functions and procedures can be defined. The sequential_statement_part contains the description of the process functionality with ordered sequential statements.

Sensitivity:
The process statement can have an explicit sensitivity list. The list defines the signal that cause the statements inside the process statement to execute whenever one or more elements of the list change value.

When the program flow reaches the last sequential statement, the process becomes suspended, until another event occurs on a signal that is sensitive.

Following are the sequential statements:

if-elsif-else statement:
This branching statement is equivalent to the ones found in other programming languages

Syntax:
if condition then
sequential_statements
{elsif condition then
sequential_statements}
[else
sequential_statements]
end if;

Example1: if statement(without else) and a common use of the VHDL attribute.

count: process (x)
variable cnt : integer :=0 ;
begn
if (x='1' and x'last_value='0') then
cnt:=cnt+1;
end if;
end process;

This if statement has two main parts, the condition and the statement body. A condition is any boolean expression (an expression that evaluates to TRUE and FALSE, such as expressions using relational operators). The condition in the example uses the attribute last_value, which is used to determine the last value that a signal had. Attributes can be used to obtain a lot of auxiliary information about signals.

The execution of the if statement begins by evaluating the condition. If the condition evaluates to the value TRUE then the statements in the statement body will be executed. Otherwise, execution will continue after the end if and the statement body of the if statement is skipped. Thus, the assignment statement in this example is executed every time there is a rising edge on the signal x, counting the number of rising edges.

Example 2: D flip flip model

library ieee ;
use ieee.std_logic_1164.all;
entity dff is
port( data_in: in std_logic;
  clock: in std_logic;
...}
data_out: out std_logic);
end dff;
architecture behv of dff is
begin
process(clock)
begin
-- clock rising edge
if (clock='1' and clock'event) then
data_out <= data_in;
end if;
end process;
end behv;
clock='1' and clock'event – This condition becomes true, when there is an event on the clock and clock state is equal to one i.e. rising edge of the clock.
clock'event – Event is an attribute on the signal to check whether any change in the signal is present.
case statement:
This statement is also identical to switch statement found in C programming language.
Syntax:
case expression is
{when choices => sequential_statements}
[when others => sequential_statements]
end case:
The case statement selects one of the branches of execution based on the value of expression.
Choices may be expressed as single value or as a range of values.
Either all possible values of expression must be covered with choices or the case statement has to be completed with an others branch.
Example1: VHDL code for 4:1 MUX (using case statement)
library ieee;
use ieee.std_logic_1164.all;
entity mux is
Port ( i : in std_logic_vector(3 downto 0);
s : in std_logic_vector(1 downto 0);
y : out std_logic);
end mux;
arhitecture Behavioral of mux is
begin
process(s,i)
begin
case s is
when "00"=> y<=i(0);
when "01"=> y<=i(1);
when "10"=> y<=i(2);
when "11"=> y<=i(3);
when others =>y<='Z';
end case ;
end process;
Loop statement
Loop statement is a conventional loop structure found in other programming languages.
Syntax for while loop:
\[ \text{while condition loop} \quad \text{--controlled by condition} \]
Example:
\[ X<=1; \quad \text{sum}<=1; \]
\[ \text{While (x<10) loop} \]
\[ \text{sum} \text{<=sum*2;} \]
\[ x<=x+1; \]
\[ \text{end loop}; \]
Syntax for for loop:
\[ \text{for identifier in value1 to|downto value2 loop} \quad \text{--with counter} \]
In the for loop the counter identifier is automatically declared. It is handled as a local variable within the loop statement. Assigning a value to identifier or reading it outside the loop is not possible. The for statement is used to execute a list of statements several times.
Example 1: four bit parallel adder using for loop
\[ \text{library ieee;} \]
\[ \text{use ieee.std_logic_1164.all;} \]
\[ \text{entity FOURBITADD is} \]
\[ \text{port (a, b: in std_logic_vector(3 downto 0); Cin : in std_logic; sum: out std_logic_vector (3 downto 0); Cout: out std_logic);} \]
\[ \text{end FOURBITADD;} \]
\[ \text{architecture fouradder_loop of FOURBITADD is} \]
\[ \text{signal ct: std_logic_vector(4 downto 0);} \]
\[ \text{begin} \]
\[ \text{process(a,b,cin)} \]
\[ \text{begin} \]
\[ \text{ct}(0)<= cin; \]
\[ \text{for i in 0 to 3 loop} \]
\[ \text{s}(i)<=a(i) xor b(i) xor ct(i); \]
\[ \text{ct}(i+1)<= (a(i) and b(i)) or (a(i) and ct(i)) or (b(i) and ct(i)); \]
\[ \text{end loop; Cout<= ct}(4); \]
\[ \text{end process;} \]
\[ \text{end fouradder_loop;} \]
Syntax for unconditional loop:
\[ \text{loop} \]
\[ \text{sequential_statements} \]
\[ \text{exit when (condition);} \]
\[ \text{end loop [loop_label];} \]
Exit statement allows the user to terminate the loop.

Next, Exit, Wait and Assert statements
next and exit statement:
With these two statements a loop iteration can be terminated before reaching the keyword
end loop.
With `next` the remaining sequential statements of the loop are skipped and the next
iteration is started at the beginning of the loop.
The `exit` directive skips the remaining statements and all remaining loop iterations.
Syntax:

```plaintext
next [loop_label][when condition];
exit [loop_label][when condition];
```

Example for `next` statement:
```
Process (a, b)
Begin
for i in 0 to 15 loop
if (i = 7 ) then
next;
else
q(i)<=a(i) AND b(i);
end if;
end loop;
end process;
```
The loop statement logically ands array of `a` and `b` bits. And transfers result to `q`. This
behavior continues except for 7th element. When `i=7` the execution starts from `next`
iteration. The statements after `next` are not executed for current iteration.
Example for `exit` statement:
```
Sum:=1;
L3: Loop
Sum:=sum*10;
If sum>100 then
Exit L3;
End if;
End loop L3;
Exit statement provides termination for unconditional loop depending on
condition.
```

`wait` statement:
This statements may only be used in processes without a `sensitivity_list`. The purpose of
the `wait` statement is to control activation and suspension of the process.
Syntax:

```plaintext
wait on signal_names
wait until condition
wait for time_expression;
```
The arguments of the `wait` statement have the following interpretations:

1. `wait on signal_names`: The process gets suspended at this line until there is an event
   on at least one signal in the list `signal_names`. The `signal_names` are separated by
   commas; brackets are not used. It can be compared to the `sensitivity_list` of the
   process statement.
Example 1: D flip flop model using `wait` statement
```
library ieee ;
use ieee.std_logic_1164.all;
entity dff is
port(reset, d: in std_logic;
clock: in std_logic;
q: out std_logic);
end dff;
architecture behv of dff is
```
begin
process
begin
--asynchronous reset input
if (reset="0") then q<=0;
-- clock rising edge
elsif (clock='1' and clock'event) then
q <= d;
end if;
wait on reset,clock;
end process;
end behv;
The statements within the process body are executed only when there is an event on reset or event on clock.

2. **wait until condition:** The process gets suspended until the condition becomes true.

Example (synchronous reset input)
Process
Begin
Wait until clock='1' and clock'event
If (reset='0') then
Q<=0;
Else q<=d;
End if;
End process;

When the rising edge of the clock occurs, the Reset signal is tested first. If Reset is 0, d is assigned to q output.

3. **wait for time_expression:** The process becomes suspended for the time specified by
time_expression.
Process
Begin
A<='0'; Wait for 5ns;
A<='1'; Wait for 5ns;
End process;

In the above statement, it generates a clock for 5ns low state and 5ns high state.

4. **wait** without any argument: The process gets suspended until the end of the simulation.

**assertion statement:**
Generating error or warning messages is possible also within the process environment.
Syntax:
assert condition
[report string_expr]
[severity failure|error|warning|note];

Example:
In JK or D Flip flop, if both asynchronous inputs Set and Reset are at logical 0 state, changes output q and qb both to be at 1 and 1 which is the violation of Boolean law. This condition can be verified by assert statement.
Assert (Set='1' or Reset = '1')
Report “Set and Reset both are 0”
Severity ERROR;
If we wish to check D input has stabilized before the clock input changes, then assert statement can be used as shown.
Assert (Clk = ‘1’ and Clk’Event and D’STABLE(3 ns))
Report “Setup time violation”
Severity WARNING;
If the condition inside the assert statement is false, the statement outputs a user specified
text string (that is written after report) to the standard console and the severity terminates
the program compilation depending on severity level.
The four levels of severity are: (1). Note (2) Warning (3) Error (4) Failure
Similarly hold time of the D Flipfop is defined as the time after a clock edge for which
data must be stable.
Assert (Clk=’1’ and D’EVENT and Clk’STABLE(5 ns))
Report “Hold time violation”
Severity WARNING;
The assert statement is passive, meaning that there is no signal assignment.

Sequential Statements in Verilog:

1. begin
   sequential_statements
end
2. if (expression)
   sequential_statement
   [else
    sequential_statement]
3. case (expression)
   expr: sequential_statement
   .......
   default: sequential_statement
endcase
4. forever
   sequential_statement
5. repeat (expression)
   sequential_statement
6. while (expression)
   sequential_statement
7. for (expr1; expr2; expr3)
   sequential_statement
8. # (time_value)
   • Makes a block suspend for
     “time_value” time units.
9. @ (event_expression)
   • Makes a block suspend until
     event_expression triggers.
HDL Description of a 2x1 Multiplexer Using IF-ELSE—VHDL and Verilog

FIGURE 3.1 2X1 Multiplexer. (a) Flow chart

(b) Logic Symbol.
VHDL 2x1 Multiplexer Using IF-ELSE

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MUX_if is
port (A, B, SEL, Gbar : in std_logic; Y : out std_logic);
end MUX_if;
architecture MUX_bh of MUX_if is
begin
process (SEL, A, B, Gbar)
-- SEL, A, B, and Gbar are the sensitivity list of the process.
variable temp : std_logic;

--It is common practice in behavioral description to use
--variable(s) rather than signal(s). This is done to avoid
--any timing errors that may arise due to the sequential
--execution of signal statements by the behavioral
--description. Execution of variable assignment statements
--is the same as in C language. After calculating the value
--of the variable, it is assigned to the output signal.
--In this example, temp is calculated as the output of the
--multiplexer. After calculation, temp is assigned to
--the output signal Y.

begin
  if Gbar = '0' then
    if SEL = '1' then
      temp := B;
    else
      temp := A;
    end if;
    Y <= temp;
  else
    Y <= 'Z';
  end if;
end process;
end MUX_bh;

Verilog 2x1 Multiplexer Using IF-ELSE
module mux2x1 (A, B, SEL, Gbar, Y);
input A, B, SEL, Gbar;
output Y;
reg Y;
always @ (SEL, A, B, Gbar)
begin
  if (Gbar == 1)


\( Y = \text{if} \) (SEL)
\( Y = B; \)
/* This is a procedural assignment. Procedural assignments
are used to assign values to variables declared as regs
(as Y here in this module). Procedural statements have
to appear inside always, blocks, initial, tasks, or functions*/

else
\( Y = A; \)
end
end

endmodule

**HDL Description of a 2x1 Multiplexer Using ELSE-IF—VHDL and Verilog**

**VHDL 2x1 Multiplexer Using ELSE-IF**
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity MUXBH is
  port (A, B, SEL, Gbar : in std_logic;
  Y : out std_logic);
end MUXBH;
architecture MUX_bh of MUXBH is
begin
process (SEL, A, B, Gbar)
variable temp : std_logic;
begin
  if (Gbar = '0') and (SEL = '1') then
    temp := B;
  elsif (Gbar = '0') and (SEL = '0') then
    temp := A;
  else
    temp := 'Z'; -- Z is high impedance.
  end if;
  Y <= temp;
end process;
end MUX_bh;

**Verilog 2x1 Multiplexer Using ELSE-IF**
module MUXBH (A, B, SEL, Gbar, Y);
input A, B, SEL, Gbar;
output Y;
reg Y; /* since Y is an output and appears inside always,
Y has to be declared as reg (register) */

always @ (SEL, A, B, Gbar)
begin
  if (Gbar == 0 & SEL == 1)
  begin
    Y = B;
  end
  else if (Gbar == 0 & SEL == 0)
   Y = A;
  else
  Y = 1'bz; //Y is assigned to high impedance
end

endmodule

Behavioral Description of a Latch Using Variable and Signal Assignments

FIGURE 3.5 D_Latch (a) Flowchart

(b) Logic Symbol
VHDL Code for Behavioral Description of D-Latch Using Variable-Assignment Statements

entity DLTCH_var is  
  port (d, E : in bit; Q, Qb : out bit);  
-- Since we are using type bit, no need for attaching a Library.  
-- If we use std_logic, we should attach the IEEE Library.  
end DLTCH_var;  
architecture DLCH_VAR of DLTCH_var is  
begin  
VAR : process (d, E)  
variable temp1, temp2 : bit;  
begin  
if E = '1' then  
temp1 := d; -- Variable assignment statement.  
temp2 := not temp1; -- Variable assignment statement.  
end if;  
Qb <= temp2; -- Value of temp2 is passed to Qb  
Q <= temp1; -- Value of temp1 is passed to Q  
end process VAR;  
end DLCH_VAR;  

VHDL Code for Behavioral Description of D-Latch Using Signal-Assignment Statements

entity Dltch_sig is  
  port (d, E : in bit; Q : buffer bit; Qb : out bit);  
--Q is declared as a buffer because it is an input/output  
--signal; it appears on both the left and right  
-- hand sides of assignment  
--statements.  
end Dltch_sig;  
architecture DL_sig of Dltch_sig is  
begin  
process (d, E)  
begin  
if E = '1' then  
  Q <= d; -- signal assignment  
  Qb <= not Q; -- signal assignment  
end if;  
end process;  
end DL_sig;
Behavioral Description of a Positive Edge – Triggered JK Flip-Flop Using the CASE statement

**FIGURE 3.6 JK flip-flop. (a) Logic Symbol**

(b) State diagram

**HDL Code for a Positive Edge-Triggered JK Flip-Flop Using the Case Statement—VHDL and Verilog**

**VHDL Positive Edge-Triggered JK Flip-Flop Using Case**

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity JK_FF is
  port(JK : in bit_vector (1 downto 0);
       clk : in std_logic;
       q, qb : out bit);
end JK_FF;
architecture JK_BEH of JK_FF is
begin
  P1 : process (clk)
  variable temp1, temp2 : bit;
  begin
    if rising_edge (clk) then
      case JK is
```

```vhdl
CITSTUDENTS.IN
```
when "01" => temp1 := '0';
when "10" => temp1 := '1';
when "00" => temp1 := temp1;
when "11" => temp1 := not temp1;
end case;
q <= temp1;
temp2 := not temp1;
qb <= temp2;
end if;
end process P1;

end JK_BEH;

Verilog Positive Edge-Triggered JK Flip-Flop Using Case
module JK_FF (JK, clk, q, qb);
input [1:0] JK;
input clk;
output q, qb;
reg q, qb;

always @ (posedge clk)
begin
  case (JK)
    2'd0 : q = q;
    2'd1 : q = 0;
    2'd2 : q = 1;
    2'd3 : q =~ q;
  endcase
  qb =~ q;
end
endmodule

Behavioral description of a 3-bit binary counter with active high synchronous clear

FIGURE 3.8 Logic symbol of a 3-bit counter with clear.
HDL Code for a 3-Bit Binary Counter Using the Case Statement

**VHDL 3-Bit Binary Counter Case Statement Description**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CT_CASE is
  port (clk, clr : in std_logic;
        q : buffer std_logic_vector (2 downto 0));
end CT_CASE;

architecture ctr_case of CT_CASE is
begin
  ctr : process(clk)
  variable temp : std_logic_vector (2 downto 0) := "101";
  --101 is the initial value, so the counter starts from 110
  begin
    if rising_edge (clk) then
      if clr = '0' then
        case temp is
        when "000" => temp := "001";
        when "001" => temp := "010";
        when "010" => temp := "011";
        when "011" => temp := "100";
        when "100" => temp := "101";
        when "101" => temp := "110";
        when "110" => temp := "111";
        when "111" => temp := "000";
        when others => temp := "000";
        end case;
      else
        temp := "000";
      end if;
    end if;
  end if;
  q <= temp;
end process ctr;
end ctr_case;
```

**Verilog 3-Bit Binary Counter Case Statement Description**

```verilog
module CT_CASE (clk, clr, q);
input clk, clr;
output [2:0] q;
reg [2:0] q;
initial /* The initial procedure is to force the counter
to start from initial count q=110 */
```
q = 3'b101;
always @ (posedge clk)
begin
if (clr == 0)
begin
  case (q)
  3'd0 : q = 3'd1;
  3'd1 : q = 3'd2;
  3'd2 : q = 3'd3;
  3'd3 : q = 3'd4;
  3'd4 : q = 3'd5;
  3'd5 : q = 3'd6;
  3'd6 : q = 3'd7;
  3'd7 : q = 3'd0;
  endcase
end
else
  q = 3'b000;
end
endmodule

Behavioral Description of a 4-bit positive Edge – Triggered counter

HDL Code for a 4-Bit Counter with Synchronous Clear—VHDL and Verilog

VHDL 4-Bit Counter with Synchronous Clear Description
library ieee;
use ieee.std_logic_1164.all;
entity CNTR_LOP is
  port (clk, clr : in std_logic; q : buffer std_logic_vector (3 downto 0));
end CNTR_LOP;
architecture CTR_LOP of CNTR_LOP is
begin
ct : process(clk)
begin
  if rising_edge (clk) then
    if (clr = '0') then
      result := 0;
      -- change binary to integer
      lop1 : for i in 0 to 3 loop
        if temp(i) = '1' then
          result := result + 2**i;
        end if;
      end loop;
      -- increment result to describe a counter
    end if;
  end if;
end
end
result := result + 1;
-- change integer to binary
for j in 0 to 3 loop
  if (result MOD 2 = 1) then
    temp (j) := '1';
  else temp (j) := '0';
  end if;
end loop;
else temp := "0000";
end if;
q <= temp;
end if;
end process ct;
end CTR_LOP;

Verilog 4-Bit Counter with Synchronous Clear Description
module CNTR_LOP (clk, clr, q);
input clk, clr;
output [3:0] q;
reg [3:0] q;
integer i, j, result;
initial
begin
  q = 4'b0000; //initialize the count to 0
end
always @ (posedge clk)
begin
  if (clr == 0)
  begin
    result = 0;
    //change binary to integer
    for (i = 0; i < 4; i = i + 1)
    begin
      if (q[i] == 1)
        result = result + 2**i;
    end
    result = result + 1;
    for (j = 0; j < 4; j = j + 1)
    begin
      if (result %2 == 1)
        q[j] = 1;
      else
        q[j] = 0;
      result = result/2;
    end
  end
end
end
end
else q = 4'b0000;
end
endmodule

Behavioral Description of a 4-bit counter with Synchronous Hold

HDL Code for a 4-Bit Counter with Synchronous Hold—VHDL and Verilog

VHDL 4-Bit Counter with Synchronous Hold Description

library ieee;
use ieee.std_logic_1164.all;
entity CNTR_Hold is
port (clk, hold : in std_logic; q : buffer std_logic_vector (3 downto 0));
end CNTR_Hold;
architecture CNTR_Hld of CNTR_Hold is
begin
cr : process (clk)
variable temp : std_logic_vector (3 downto 0) := "0000";
-- temp is initialized to 0 so count starts at 0
variable result : integer := 0;
begin
if rising_edge (clk) then
result := 0;
-- change binary to integer
lop1 : for i in 0 to 3 loop
if temp(i) = '1' then
result := result + 2**i;
end if;
end loop;
-- increment result to describe a counter
result := result + 1;
-- change integer to binary
lop2 : for i in 0 to 3 loop
-- exit the loop if hold = 1
exit when hold = '1';
-- “when” is a predefined word
if (result MOD 2 = 1) then
temp (i) := '1';
else
temp (i) := '0';
end if;
end
end
end

--Successive division by 2
result := result/2;
end loop;
q <= temp;
end if;
end process ct;
end CNTR_Hld;

Verilog 4-Bit Counter with Synchronous Hold Description
module CT_HOLD (clk, hold, q);
input clk, hold;
output [3:0] q;
reg [3:0] q;
integer i, result;
initial
begin
q = 4'b0000; //initialize the count to 0
end
always @ (posedge clk)
begin
result = 0;
//change binary to integer
for (i = 0; i <= 3; i = i + 1)
begin
if (q[i] == 1)
result = result + 2**i;
end
result = result + 1;
for (i = 0; i <= 3; i = i + 1)
begin
if (hold == 1)
i = 4; //4 is out of range, exit.
else
begin
if (result %2 == 1)
q[i] = 1;
else
q[i] = 0;
result = result/2;
end
end
end
endmodule
Calculating the Factorial Using Behavioral Description with While – Loop
HDL Code for Calculating the Factorial of Positive Integers—VHDL and Verilog

**VHDL: Calculating the Factorial of Positive Integers**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
--The above library statements can be omitted; however no
--error if they are not omitted. The basic VHDL
-- has type "natural."
entity factr is
port(N : in natural; z : out natural);
end factr;
architecture factorl of factr is
begin
  process (N)
  variable y, i : natural;
  begin
    y := 1;
    i := 0;
    while (i < N) loop
      i := i + 1;
      y := y * i;
    end loop;
    z <= y;
  end process;
end factorl;
```

**Verilog: Calculating the Factorial of Positive Integers**

```verilog
module factr (N, z);
input [5:0] N;
output [15:0] z;
reg [15:0] z;
/* Since z is an output, and it will appear inside "always," then
Z has to be declared "reg" */
i integer i;
always @ (N)
begin
  z = 1;
  //z can be written as 16'b0000000000000001 or 16'd1.
  i = 0;
  while (i < N)
  begin
    i = i + 1;
    z = i * z;
  end
end
endmodule
```
Booth Algorithm:

![Booth Algorithm Flowchart]

**FIGURE 3.13** Flowchart of Booth multiplication algorithm. 4x4-Bit Booth Algorithm—VHDL and Verilog

### VHDL 4x4-Bit Booth Algorithm

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity booth is
port (X, Y : in signed (3 downto 0);
Z : buffer signed (7 downto 0));
end booth;
architecture booth_4 of booth is
begin
process (X, Y)
variable temp : signed (1 downto 0);
variable sum : signed (7 downto 0);
variable E1 : unsigned (0 downto 0);
variable Y1 : signed (3 downto 0);
begin
sum := "00000000"; E1 := "0";
for i in 0 to 3 loop
temp := X(i) & E1(0);
Y1 := -Y;
end process (X, Y)
```

```vhdl
```
case temp is
  when "10" => sum (7 downto 4) := sum (7 downto 4) + Y1;
  when "01" => sum (7 downto 4) := sum (7 downto 4) + Y;
  when others => null;
end case;
sum := sum srl 1; --This is a logical
--shift of one position to the right
sum (7) := sum(6);

--The above two statements perform arithmetic shift where
--the sign of the number is preserved after the shift.

E1(0) := x(i);
end loop;
  if (y = "1000") then

--If Y = 1000; then according to our code,
--Y1 = 1000 (-8 not 8 because Y1 is 4 bits only).
--The statement sum = -sum adjusts the answer.

    sum := - sum;
  end if;

z <= sum;
end process;
end booth_4;

Verilog 4x4-Bit Booth Algorithm
module booth (X, Y, Z);
input signed [3:0] X, Y;
output signed [7:0] Z;
reg signed [7:0] Z;
reg [1:0] temp;
integer i;
reg E1;
reg [3:0] Y1;
always @ (X, Y)
begin
  Z = 8'd0;
  E1 = 1'd0;
  for (i = 0; i < 4; i = i + 1)
begin
  temp = {X[i], E1};
end

end
endmodule
//The above statement is catenation

Y1 = - Y;

//Y1 is the 2' complement of Y

case (temp)
    default : begin end
endcase

Z = Z >> 1;

/*The above statement is a logical shift of one position to the right*/

Z[7] = Z[6];

/*The above two statements perform arithmetic shift where the sign of the number is preserved after the shift.*/

E1 = X[i];
end

if (Y == 4'd8)

/*If Y = 1000; then according to our code,
Y1 = 1000 (-8 not 8, because Y1 is 4 bits only).
The statement sum = - sum adjusts the answer.*/

    begin
        Z = - Z;
    end

end

endmodule
ASSIGNMENT QUESTIONS

1) Write behavioral description of half adder in VHDL and verilog with propagation delay of 5nsec. Discuss the important features of their description in VHDL and verilog.

2) Explain the structure of various loop statements in HDL with examples.

3) Explain verilog Repeat and Forever statements with an example.

4) Explain different loop statements in

5) Explain IF and CASE statements with examples.

6) Explain Booth algorithm with a flow chart. Write VHDL or verilog description for 4X4 bit booth algorithm.

7) Write VHDL code for a D-latch using variable assignment & signal assignment statements with simulation waveforms clearly distinguish between the two statements.

8) Write a behavioral description of D-Latch using variable & signal assign

9) What is HDL? Why do you need it
UNIT 4: STRUCTURAL DESCRIPTIONS

Syllabus of unit 4: Hours: 7
Highlights of structural Description, Organization of the structural Descriptions, Binding, state Machines, Generate, Generic, and Parameter statements.

Recommended readings:

1. HDL Programming (VHDL and Verilog)- Nazeih M. Botros- Dreamtech Press
2. Verilog HDL –Samir Palnitkar-Pearson Education
3. VHDL - Douglas Perry-Tata McGraw-Hill
4. A Verilog HDL Primer- J.Bhaskar – BS Publications
5. Circuit Design with VHDL- Volnei A. Pedroni-PHI
UNIT 4. STRUCTURAL DESCRIPTIONS

Highlights of structural Descriptions:
1) Structural description simulates the system by describing its logical components. The components can be gate level, such as AND gates, OR gates or NOT gates or components can be in a higher logical level such as Register Transfer Level (RTL) or processor level.
2) It is more convenient to use structural description rather than behavioral description for systems that required a specific design.
3) All statements in structural description are concurrent. At any simulation time, all statements that have an event are executed concurrently.
4) A major difference between VHDL and Verilog structural description is the availability of components.

HDL Structural Description—VHDL and Verilog

VHDL Structural Description

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity system is
    port (a, b : in std_logic;
    sum, cout : out std_logic);
end system;

architecture struct_exple of system is
    --start declaring all different types of components
    component xor2
        port (I1, I2 : in std_logic;
        O1 : out std_logic);
    end component;
    component and2
        port (I1, I2 : in std_logic;
        O1 : out std_logic);
    end component;
    begin
    --Start of instantiation statements
    X1 : xor2 port map (a, b, sum);
    A1 : and2 port map (a, b, cout);
end struct_exple;

Verilog Structural Description

module system (a, b, sum, cout);
    input a, b;
    output sum, cout;
    xor X1 (sum, a, b);
    /* X1 is an optional identifier; it can be omitted.*/
    and a1 (cout, a, b);
    /* a1 is optional identifier; it can be omitted.*/
end module;
endmodule

Fig. Verilog built-in gates.

Structural Description of a Half Adder

HDL Code of Half Adder—VHDL and Verilog

VHDL Half Adder Description
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity xor2 is
port(I1, I2 : in std_logic; O1 : out std_logic);
end xor2;
architecture Xor2_0 of xor2 is
begin
O1 <= I1 xor I2;
end Xor2_0;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity and2 is
port (I1, I2 : in std_logic; O1 : out std_logic);
end and2;
architecture and2_0 of and2 is
begin
O1 <= I1 and I2;
end and2_0;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity half_add is
  port (a, b : in std_logic; S, C : out std_logic);
end half_add;

architecture HA_str of half_add is
  component xor2
    port (I1, I2 : in std_logic; O1 : out std_logic);
  end component;
  component and2
    port (I1, I2 : in std_logic; O1 : out std_logic);
  end component;
begin
  X1 : xor2 port map (a, b, S);
  A1 : and2 port map (a, b, C);
end HA_str;

Verilog Half Adder Description
module half_add (a, b, S, C);
  input a, b;
  output S, C;
  xor (S, a, b);
  and (C, a, b);
endmodule

BINDING
Binding (linking) segment1 in HDL code to segment2 makes all information in
segment2.
Binding Between a Library and Component in VHDL

--First, write the code that will be bound to another module
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity bind2 is
  port (I1, I2 : in std_logic; O1 : out std_logic);
end bind2;
architecture xor2_0 of bind2 is
begin
  O1 <= I1 xor I2;
end xor2_0;

architecture and2_0 of bind2 is
begin
  O1 <= I1 and I2;
end and2_0;
architecture and2_4 of bind2 is  
begin  
   O1 <= I1 and I2 after 4 ns;  
end and2_4;  

--After writing the above code; compile it and store it in a  
-- known location. Now, open another module  
--where the above information is to be used.  
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
entity half_add is  
   port (a, b : in std_logic; S, C : out std_logic);  
end half_add;  

architecture HA_str of half_add is  
component xor2  
   port (I1, I2 : in std_logic; O1 : out std_logic);  
end component;  
component and2  
   port (I1, I2 : in std_logic; O1 : out std_logic);  
end component;  
for all : xor2 use entity work.bind2 (xor2_0);  
for all : and2 use entity work.bind2 (and2_4);  
begin  
   X1 : xor2 port map (a, b, S);  
   A1 : and2 port map (a, b, C);  
end HA_str;  

Binding Between Two Modules in Verilog

module one (O1, O2, a, b);  
   input [1:0] a;  
   input [1:0] b;  
   output [1:0] O1, O2;  
endmodule  

two M0 (O1[0], O2[0], a[0], b[0]);  
two M1 (O1[1], O2[1], a[1], b[1]);  
endmodule

module two (s1, s2, a1, b1);  
   input a1;  
   input b1;  
   output s1, s2;  
xor (s1, a1, b1);  
and (s2, a1, b1);  
endmodule
Structural Description of a 2X1 Multiplexer with Active Low Enable

FIGURE: Multiplexer  (a) Logic diagram.

(b) Logic symbol

VHDL Code for Several Gates

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

department bind1 is
port (I1 : in std_logic; O1 : out std_logic);
end bind1;
architecture inv_0 of bind1 is
begin
O1 <= not I1; --This is an inverter with zero delay
end inv_0;

architecture inv_7 of bind1 is
begin
O1 <= not I1 after 7 ns; --This is an inverter with a 7-ns delay
end inv_7;
```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bind2 is
port (I1, I2 : in std_logic; O1 : out std_logic);
end bind2;

architecture xor2_0 of bind2 is
begin
O1 <= I1 xor I2; --This is exclusive-or with zero delay.
end xor2_0;

architecture and2_0 of bind2 is
begin
O1 <= I1 and I2; --This is a two input and gate with zero delay.
end and2_0;

architecture and2_7 of bind2 is
begin
O1 <= I1 and I2 after 7 ns; -- This is a two input and gate with 7-ns delay.
end and2_7;

architecture or2_0 of bind2 is
begin
O1 <= I1 or I2; -- This is a two input or gate with zero delay.
end or2_0;

architecture or2_7 of bind2 is
begin
O1 <= I1 or I2 after 7 ns; -- This is a two input or gate with 7-ns delay.
end or2_7;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bind3 is
port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end bind3;

architecture and3_0 of bind3 is
begin
O1 <= I1 and I2 and I3; -- This is a three input and gate with zero delay.
end and3_0;
architecture and3_7 of bind3 is
begin
O1 <= I1 and I2 and I3 after 7 ns; --This is a three input
--and gate with 7-ns
--delay.
end and3_7;

architecture or3_0 of bind3 is
begin
O1 <= I1 or I2 or I3; --This is a three input or gate
--with zero delay.
end or3_0;

architecture or3_7 of bind3 is
begin
O1 <= I1 or I2 or I3 after 7 ns; --This is a three input or gate
--with 7-ns delay.
end or3_7;

**HDL Description of a 2x1 Multiplexer with Active Low Enable—VHDL and Verilog**

**VHDL 2x1 Multiplexer with Active Low Enable**

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity mux2x1 is
    port (A, B, SEL, Gbar : in std_logic;
        Y : out std_logic);
end mux2x1;

architecture mux_str of mux2x1 is
    --Start Components Declaration
    component and3
        port (I1, I2, I3 : in std_logic; O1 : out std_logic);
    end component;

    --Only different types of components need be declared.
    --Since the multiplexer has two identical AND gates,
    --only one is declared.

    component or2
        port (I1, I2 : in std_logic; O1 : out std_logic);
    end component;

    component Inv
        port (I1 : in std_logic; O1 : out std_logic);
    end component;
signal S1, S2, S3, S4, S5 : std_logic;
for all : and3 use entity work.bind3 (and3_7);
for all : Inv use entity work.bind1 (inv_7);
for Or1 : or2 use entity work.bind2 (or2_7);
begin
   --Start instantiation
   A1 : and3 port map (A, S2, S1, S4);
   A2 : and3 port map (B, S3, S1, S5);
   IV1 : Inv port map (SEL, S2);
   IV2 : Inv port map (Gbar, S1);
   IV3 : Inv port map (S2, S3);
   or1 : or2 port map (S4, S5, Y);
end mux_str;

Verilog 2x1 Multiplexer with Active Low Enable
module mux2x1 (A, B, SEL, Gbar, Y);
  input A, B, SEL, Gbar;
  output Y;
  and #7 (S4, A, S2, S1);
  or #7 (Y, S4, S5);
  and #7 (S5, B, S3, S1);
  not #7 (S2, SEL);
  not #7 (S3, S2);
  not #7 (S1, Gbar);
endmodule

Structural Description of a 2x4 decoder with three – state output

FIGURE: Logic diagram of a 2x4 Decoder with tristate output.
FIGURE: Verilog built-in buffers.

VHDL Behavioral Description of a Tri-State Buffer

entity bind2 is
  port (I1, I2 : in std_logic; O1 : out std_logic);
end bind2;

-- Add the following architecture to
-- the entity bind2 of Listing 4.8
architecture bufif1 of bind2 is
begin
  buf : process (I1, I2)
  variable tem : std_logic;
  begin
    if (I2 = '1') then
      tem := I1;
    else
      tem := 'Z';
    end if;
    O1 <= tem;
  end process buf;
end bufif1;

HDL Description of a 2x4 Decoder with Tri-State Output—VHDL and Verilog

VHDL 2x4 Decoder with Tri-State Output
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decoder2x4 is
  port (I : in std_logic_vector(1 downto 0); Enable : in std_logic; D : out std_logic_vector (3 downto 0));
end decoder2x4;

architecture decoder of decoder2x4 is
component bufif1
port (I1, I2 : in std_logic; O1 : out std_logic);
end component;
component inv
    port (I1 : in std_logic; O1 : out std_logic);
end component;
component and2
    port (I1, I2 : in std_logic; O1 : out std_logic);
end component;
for all : bufif1 use entity work.bind2 (bufif1);
for all : inv use entity work.bind1 (inv_0);
for all : and2 use entity work.bind2 (and2_0);
signal s0, s1, s2, s3 : std_logic;
signal Ibar : std_logic_vector (1 downto 0);
-- The above signals have to be declared before they can be used
begin
    B0 : bufif1 port map (s0, Enable, D(0));
    B1 : bufif1 port map (s1, Enable, D(1));
    B2 : bufif1 port map (s2, Enable, D(2));
    B3 : bufif1 port map (s3, Enable, D(3));
    iv0 : inv port map (I(0), Ibar(0));
    iv1 : inv port map (I(1), Ibar(1));
    a0 : and2 port map (Ibar(0), Ibar(1), s0);
    a1 : and2 port map (I(0), Ibar(1), s1);
    a2 : and2 port map (Ibar(0), I(1), s2);
    a3 : and2 port map (I(0), I(1), s3);
end decoder;

Verilog 2x4 Decoder with Tri-State Output
module decoder2x4 (I, Enable, D);
    input [1:0] I;
    input Enable;
    output [3:0] D;
    wire [1:0] Ibar;
    bufif1 (D[0], s0, Enable);
    bufif1 (D[1], s1, Enable);
    bufif1 (D[2], s2, Enable);
    bufif1 (D[3], s3, Enable);
    not (Ibar[0], I[0]);
    not (Ibar[1], I[1]);
    and (s0, Ibar[0], Ibar[1]);
    and (s1, I[0], Ibar[1]);
    and (s2, Ibar[0], I[1]);
    and (s3, I[0], I[1]);
endmodule
F
undamentals of HDL

10EC45

FIGURE: Simulation waveform of a 2X1 decoder with tristate output.

Structural Description of a Full Adder

FIGURE: Full adder as two half adders. (a) Logic Symbol

(b) Logic diagram

VHDL Code for the Half Adder

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
extity bind22 is
  Port (I1, I2 : in std_logic; O1, O2 : out std_logic);
end bind22;

architecture HA of bind22 is
component xor2
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;
component and2
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

for A1 : and2 use entity work.bind2 (and2_0);
for X1 : xor2 use entity work.bind2 (xor2_0);
begin
  X1 : xor2 port map (I1, I2, O1);
  A1 : and2 port map (I1, I2, O2);
end HA;

**HDL Description of a Full Adder (Figures 4.6a and 4.6b)—VHDL and Verilog**

**VHDL Full Adder Description**

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FULL_ADDER is
  Port (x, y, cin : in std_logic; sum, carry : out std_logic);
end FULL_ADDER;
architecture full_add of FULL_ADDER is
  component HA
    Port (I1, I2 : in std_logic; O1, O2 : out std_logic);
  end component;
  component or2
    Port (I1, I2 : in std_logic; O1 : out std_logic);
  end component;
  for all : HA use entity work.bind22 (HA);
  for all : or2 use entity work.bind2 (or2_0);
signal s0, c0, c1 : std_logic;
begin
  HA1 : HA port map (y, cin, s0, c0);
  HA2 : HA port map (x, s0, sum, c1);
  r1 : or2 port map (c0, c1, carry);
end full_add;

**Verilog Full Adder Description**

module FULL_ADDER (x, y, cin, sum, carry);
input x, y, cin;
output sum, carry;
HA H1 (y, cin, s0, c0);
HA H2 (x, s0, sum, c1);
//The above two statements bind module HA
to the present module FULL_ADDER
or (carry, c0, c1);
endmodule
module HA (a, b, s, c);
  input a, b;
  output s, c;
  xor (s, a, b);
  and (c, a, b);
endmodule

**Structural Description of an SR- Latch**

![SR-Latch Diagram](image)

**FIGURE:4.7** SR- LATCH (a) Logic symbol

(b) Logic diagram.

**HDL Description of an SR Latch with NOR Gates**

**VHDL SR Latch with NOR Gates**

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity SR_latch is
  port (R, S : in std_logic;
        Q, Qbar : buffer std_logic);
  --Q, Qbar are declared buffer because
  --they behave as input and output.
end SR_latch;
architecture SR_strc of SR_latch is
--Some simulators would not allow mapping between
--buffer and out. In this
--case, change all out to buffer.
component nor2
port (I1, I2 : in std_logic; O1 : out std_logic);
end component;
for all : nor2 use entity work.bind2 (nor2_0);
begin
  n1 : nor2 port map (S, Q, Qbar);
  n2 : nor2 port map (R, Qbar, Q);
end SR_strc;

Verilog SR Latch with NOR Gates
module SR_Latch (R, S, Q, Qbar);
input R, S;
output Q, Qbar;
nor (Qbar, S,Q);
nor (Q, R, Qbar);
endmodule

FIGURE 4.8 Simulation waveform of an SR- latch.

Structural Description of a D- Latch

HDL Description of a D-Latch—VHDL and Verilog

VHDL D-Latch Description
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_Latch is
  port (D, E : in std_logic; Q, Qbar : buffer std_logic);
end D_Latch;

architecture D_latch_str of D_Latch is
Some simulators will not allow mapping between buffer and out. In this case, change all out to buffer.

component and2
    port (I1, I2 : in std_logic; O1 : out std_logic);
end component;
component nor2
    port (I1, I2 : in std_logic; O1 : out std_logic);
end component;
component inv
    port (I1 : in std_logic; O1 : out std_logic);
end component;
for all : and2 use entity work.bind2 (and2_4);
for all : nor2 use entity work.bind2 (nor2_4);
for all : inv use entity work.bind1 (inv_1);
signal Eb, s1, s2 : std_logic;
begin
    a1 : and2 port map (D, E, s1);
    a2 : and2 port map (Eb, Q, s2);
    in1 : inv port map (E, Eb);
    in2 : inv port map (Qbar, Q);
    n2 : nor2 port map (s1, s2, Qbar);
end D_latch_str;

Verilog D-Latch Description

module D_latch (D, E, Q, Qbar);
    input D, E;
    output Q, Qbar;
    /* assume 4 ns delay for and gate and nor gate, 
    and 1 ns for inverter */
    and #4 gate1 (s1, D, E);

    /* the name "gate1" is optional; we could have 
    written and #4 (s1, D, E) */
    and #4 gate2 (s2, Eb, Q);
    not #1 (Eb, E);
    nor #4 (Qbar, s1, s2);
    not #1 (Q, Qbar);
endmodule
Structural Description of a Pulse-Triggered, Master-Slave D Flip-Flop.

FIGURE 4.9 Logic symbol of the master – slave D flip-flop.

FIGURE 4.10 Logic diagram of a master – slave D flip-flop.

HDL Description of a Master-Slave D Flip-Flop—VHDL and Verilog

VHDL Master-Slave D Flip-Flop

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_FFMaster is
  Port (D, clk : in std_logic; Q, Qbar : buffer std_logic);
end D_FFMaster;

architecture D_FF of D_FFMaster is
  --Some simulators would not allow mapping between
  --buffer and out. In this
  --case, change all out to buffer.
  component inv
    port (I1 : in std_logic; O1 : out std_logic);
  end component;
  component D_latch
    port (I1, I2 : in std_logic; O1, O2 : buffer std_logic);
  end component;
```
end component;
for all : D_latch use entity work.bind22 (D_latch);
for all : inv use entity work.bind1 (inv_1);
signal clkb, clk2, Q0, Qb0 : std_logic;
begin
  D0 : D_latch port map (D, clkb, Q0, Qb0);
  D1 : D_latch port map (Q0, clk2, Q, Qbar);
in1 : inv port map (clk, clkb);
in2 : inv port map (clkb, clk2);
end D_FF;

Verilog Master-Slave D Flip-Flop
module D_FFMaster (D, clk, Q, Qbar);
input D, clk;
output Q, Qbar;
  not #1 (clkb, clk);
  not #1 (clk2, clkb);
  D_latch D0 (D, clkb, Q0, Qb0);
  D_latch D1 (Q0, clk2, Q, Qbar);
endmodule

module D_latch (D, E, Q, Qbar);
input D, E;
output Q, Qbar;
  and #4 gate1 (s1, D, E);
  and #4 gate2 (s2, Eb, Q);
  not #1 (Eb, E);
  nor #4 (Qbar, s1, s2);
  not #1 (Q, Qbar);
endmodule

FIGURE 4.11 Simulation waveform of a master–slave D flip-flop.
Structural Description of a Pulse-Triggered Master-Slave JK Flip–Flop

HDL Description of a Master-Slave JK Flip-Flop—VHDL and Verilog

VHDL Master-Slave JK Flip-Flop

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity JK_FF is
  port (J, K, clk : in std_logic; Q, Qbar : buffer std_logic);

  -- Q and Qbar are declared buffer so they can be input or output

end JK_FF;

architecture JK_Master of JK_FF is

  component and2
    port (I1, I2 : in std_logic; O1 : out std_logic);
  end component;

  component or2
    port (I1, I2 : in std_logic; O1 : out std_logic);
  end component;

  component inv
    port (I1 : in std_logic; O1 : out std_logic);
  end component;

  component D_flip

end component;

port (I1, I2 : in std_logic; O1, O2 : buffer std_logic);
end component;
for all : and2 use entity work.bind2 (and2_4);
for all : or2 use entity work.bind2 (or2_4);
for all : inv use entity work.bind1 (inv_1);
for all : D_flip use entity work.bind22 (D_FFMaster);
signal s1, s2, Kb, DD : std_logic;
begin
  a1 : and2 port map (J, Qbar, s1);
  a2 : and2 port map (Kb, Q, s2);
  in1 : inv port map (K, Kb);
  or1 : or2 port map (s1, s2, DD);
  DFF : D_flip port map (DD, clk, Q, Qbar);
end JK_Master;

Verilog Master-Slave JK Flip-Flop
module JK_FF (J, K, clk, Q, Qbar);
input J, K, clk;
output Q, Qbar;
wire s1, s2;
  and #4 (s1, J, Qbar);
  and #4 (s2, Kb, Q);
  not #1 (Kb, K);
  or #4 (DD, s1, s2);
  D_FFMaster D0 (DD, clk, Q, Qbar);
endmodule

module D_FFMaster (D, clk, Q, Qbar);
  /* We do not have to rewrite the module if the simulator
   * used can attach it to the above module (JK_FF). */
  input D, clk;
  output Q, Qbar;
  wire clkb, clk2, Q0, Qb0;
  not #1 (clkb, clk);
  not #1 (clk2, clkb);
  D_latch D0 ( D, clkb, Q0, Qb0);
  D_latch D1 (Q0, clk2, Q, Qbar);
endmodule

module D_latch (D, E, Q, Qbar);
  input D, E;
  output Q, Qbar;
  wire Eb, s1, s2;
  //assume 4-ns delay for and gate and nor gate,
//and 1 ns for inverter
and #4 gate1 (s1, D, E);
//The name gate1 is optional and could have been omitted.

and (s1, D, E);
and #4 gate2 (s2, Eb, Q);
not #1 (Eb, E);
nor #4 (Qbar, s1, s2);
not #1 (Q, Qbar);
endmodule

Structural Description of a 3-bit Ripple – Carry Adder

VHDL 3-Bit Ripple Carry Adder

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

tity three_bit_adder is
port(x, y : in std_logic_vector (2 downto 0);
cin : in std_logic; sum : out std_logic_vector (2 downto 0);
cout : out std_logic);
end three_bit_adder;

architecture three_bitadd of three_bit_adder is
component full_adder
port (I1, I2, I3 : in std_logic; O1, O2 : out std_logic);
end component;
for all : full_adder use entity work.bind32 (full_add);
signal carry : std_logic_vector (1 downto 0);
begin
M0 : full_adder port map (x(0), y(0), cin, sum(0), carry(0));
M1 : full_adder port map (x(1), y(1), carry(0), sum(1), carry(1));
M2 : full_adder port map (x(2), y(2), carry(1), sum(2), cout);
end three_bitadd;

Verilog 3-Bit Ripple Carry Adder

module three_bit_adder (x, y, cin, sum, cout);
input [2:0] x, y;
input cin;
output [2:0] sum;
output cout;
wire [1:0] carry;
  FULL_ADDER M0 (x[0], y[0], cin, sum[0], carry[0]);
  FULL_ADDER M1 (x[1], y[1], carry[0], sum[1], carry[1]);
  FULL_ADDER M2 (x[2], y[2], carry[1], sum[2], cout);
It is assumed that the module FULL_ADDER (Listing 4.13) is attached by the simulator to the module three_bit_adder so, no need to rewrite the module FULL_ADDER.*/

endmodule

**Structural Description of a 3-bit Magnitude Comparator Using 3-Bit Adder**

![A full adder-based comparator](image)

**FIGURE 4.14** A full adder-based comparator

**HDL Description of a 3-Bit Comparator Using Adders—VHDL and Verilog**

**VHDL 3-Bit Comparator Using Adders**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity three_bit_cmpare is
port (X, Y : in std_logic_vector (2 downto 0);
      xgty, xlty, xeql : out std_logic);
end three_bit_cmpare;

architecture cmpare of three_bit_cmpare is
begin
--Some simulators will not allow mapping between
--buffer and out. In this
--case, change all out to buffer.
```

CITSTUDENTS.IN
component full_adder

    port (I1, I2, I3 : in std_logic; O1, O2 : out std_logic);
end component;
component Inv
    port (I1 : in std_logic; O1 : out std_logic);
end component;
component nor2
    port (I1, I2 : in std_logic; O1 : out std_logic);
end component;
component and3
    port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;
for all : full_adder use entity work.bind32 (full_add);
for all : Inv use entity work.bind1 (inv_0);
for all : nor2 use entity work.bind2 (nor2_0);
for all : and3 use entity work.bind3 (and3_7);
    --To reduce hazards, an AND gate is
    --implemented with a 7-ns delay.
signal sum, Yb : std_logic_vector (2 downto 0);
signal carry : std_logic_vector (1 downto 0);
begin
    in1 : inv port map (Y(0), Yb(0));
in2 : inv port map (Y(1), Yb(1));
in3 : inv port map (Y(2), Yb(2));
F0 : full_adder port map (X(0), Yb(0), '0', sum(0), carry(0));
F1 : full_adder port map (X(1), Yb(1), carry(0),
    sum(1), carry(1));
F2 : full_adder port map (X(2), Yb(2), carry(1),
    sum(2), xgty);

--The current module could have been linked to the 3-bit adders
--designed in Listing 4.18 instead of linking to
--F0, F1, and F2, as was done here.
    a1 : and3 port map (sum(0), sum(1), sum(2), xeqy);
    n1 : nor2 port map (xeqy, xgty, xlty);
end cmpare;

Verilog 3-Bit Comparator Using Adders
module three_bit_cmpare (X, Y, xgty, xlty, xeqy);
input [2:0] X, Y;
output xgty, xlty, xeqy;
wire [1:0] carry;
wire [2:0] sum, Yb;
    not (Yb[0], Y[0]);
    not (Yb[1], Y[1]);
    not (Yb[2], Y[2]);
    FULL_ADDER M0 (X[0], Yb[0], 1'b0, sum[0], carry[0]);
    FULL_ADDER M1 (X[1], Yb[1], carry[0], sum[1], carry[1]);
    FULL_ADDER M2 (X[2], Yb[2], carry[1], sum[2], xgty);

    /* The current module could have been linked to the
       3-bit adders designed in Listing 4.18 instead of
       linking to F0, F1, and F2, as was done here.*/

    and #7 (xeqy, sum[0], sum[1], sum[2]);

    /* To reduce hazard use an AND gate with a delay of 7 units*/

    nor (xlty, xeqy, xgty);
endmodule

**Structural Description of an SRAM Cell**

![Figure 4.15: K-maps](image)

**FIGURE 4.15** K – maps
HDL Description of an SRAM Memory Cell—VHDL and Verilog

**VHDL SRAM Memory Cell Description**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

description memory is
    port (Sel, RW, Din : in std_logic; O1: buffer std_logic );
end memory;

architecture memory_str of memory is
    --Some simulators will not allow mapping between
    --buffer and out. In this
    --case, change all out to buffer.

    component and3
        port (I1, I2, I3 : in std_logic; O1: out std_logic);
    end component;

    component inv
        port (I : in std_logic; O : out std_logic);
    end component;

FIGURE: 4.16 SRAM memory cell. (a) Logic Symbol
```
port (I1 : in std_logic; O1 : out std_logic);
end component;

component or2
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component buffif1
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component SR_Latch
  port (I1, I2 : in std_logic; O1, O2 : buffer std_logic);
end component;
for all : and3 use entity work.bind3 (and3_0);
for all : inv use entity work.bind1 (inv_0);
for all : or2 use entity work.bind2 (or2_0);
for all : buffif1 use entity work.bind2 (buffif1);
for all : SR_Latch use entity work.bind22 (SR_Latch);
signal RWb, Dinb, S, S1, R, O11, Q : std_logic;
begin
  in1 : inv port map (RW, RWb);
in2 : inv port map (Din, Dinb);
a1 : and3 port map (Sel, RWb, Din, S); a2 :
  : and3 port map (Sel, RWb, Dinb, R); SR1 :
  : SR_Latch port map (S, R, Q, open);
--open is a predefined word;
--it indicates that the port is left open.
a3 : and3 port map (Sel, RW, Q, S1);
or1 : or2 port map (S1, S, O11);
buff1 : buffif1 port map (O11, Sel, O1);
end memory_str;

Verilog SRAM Memory Cell Description
module memory (Sel, RW, Din, O1);
input Sel, RW, Din;
output O1;
  not (RWb, RW);
  not (Dinb, Din);
  and (S, Sel, RWb, Din); and (R, Sel, RWb, Dinb); SR_Latch
RS1 (R, S, Q, Qbar); and (S1, Sel, RW, Q);
or (O11, S1, S);
buff1 (O1, O11, Sel);
endmodule
STATE MACHINES

FIGURE 4.17 Logic symbol of a 3-bit counter with active low clear.

FIGURE 4.18 State diagram of a 3-bit counter with active low clear.
Figure 4.19 K-maps
**FIGURE 4.20** Logic diagram of a 3-bit synchronous counter with active low clear using JK master–slave flip–flops.

HDL Description of a 3-Bit Synchronous Counter Using JK Master-Slave Flip-Flops—VHDL and Verilog

***Begin Listing***

**VHDL 3-Bit Synchronous Counter Using JK Master-Slave Flip-Flops**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity countr_3 is
port(clk, clrbar : in std_logic;
   q, qb : buffer std_logic_vector(2 downto 0));
end countr_3;
architecture CNTR3 of countr_3 is
begin
--Start component declaration statements

--Some simulators will not allow mapping between
--buffer and out. In this
--case, change all out to buffer.

component JK_FF
   port (I1, I2, I3 : in std_logic; O1, O2 : buffer std_logic);
end component;

component inv
   port (I1 : in std_logic; O1 : out std_logic);
end component;
```

***END LISTING***
component and2
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component or2
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

for all : JK_FF use entity work.bind32 (JK_Mster);
for all : inv use entity work.bind1 (inv_0);
for all : and2 use entity work.bind2 (and2_0);
for all : or2 use entity work.bind2 (or2_0);
signal J1, K1, J2, K2, clr, clrb1, s1 : std_logic;
begin
  FF0 : JK_FF port map (clrb1, '1', clk, q(0), qb(0));
  -- clrb1 has the same logic as clrbar
  A1 : and2 port map (clrb1, q(0), J1);
  inv1 : inv port map (clr, clrb1);
  inv2 : inv port map (clrbar, clr);
  r1 : or2 port map (q(0), clr, K1);
  FF1 : JK_FF port map (J1, K1, clk, q(1), qb(1));
  A2 : and2 port map (q(0), q(1), s1);
  A3 : and2 port map (clrb1, s1, J2);
  r2 : or2 port map (s1, clr, K2);
  FF2 : JK_FF port map (J2, K2, clk, q(2), qb(2));
end CNTR3;

Verilog 3-Bit Synchronous Counter Using JK Master-Slave Flip-Flops
module countr_3 (clk, clrbar, q, qb);
input clk, clrbar;
output [2:0] q, qb;

  JK_FF FF0(clrb1, 1'b1, clk, q[0], qb[0]);
  // clrb1 has the same logic as clrbar
  and A1 (J1, q[0], clrb1);

/*The name of the and gate “A1” and all other
gates in this code are optional; it can be omitted.*/

  not inv1 (clrb1, clr);
  not inv2 (clr, clrb1);
  or r1 (K1, q[0], clr);
  JK_FF FF1 (J1, K1, clk, q[1], qb[1]);
and A2 (s1, q[0], q[1]);
and A3 (J2, clr_b1, s1);
or or2 (K2, s1, clr);
JK_FF FF2(J2, K2, clk, q[2], qb[2]);
Endmodule

Figure 4.21 simulation waveform of a 3-bit synchronous counter with active low clear.

**Structural Description of a 3-Bit Synchronous Even Counter with active High Hold**

![State diagram of a 3-bit synchronous even counter](image)

FIGURE 4.22 State diagram of an even 3–bit counter.
FIGURE 4.23 K – maps of an even 3-bit counter.

FIGURE 4.24 Three-bit even counter. (a) Logic symbol. (b) Logic diagram
HDL Description of a 3-Bit Synchronous Even Counter with Hold—VHDL and Verilog

**VHDL 3-Bit Synchronous Even Counter with Hold**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity CTR_EVEN is
 port (H, clk : in std_logic;
       Q, Qbar : buffer std_logic_vector (2 downto 0));
end CTR_EVEN;

architecture Counter_even of CTR_EVEN is
--Some simulators will not allow mapping between
--buffer and out. In this
--case, change all out to buffer.

component inv
 port (I1 : in std_logic; O1 : out std_logic);
end component;

component and2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component or2
 port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component and3
 port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;

component or3
 port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;

component D_FF
 port (I1, I2 : in std_logic; O1, O2 : buffer std_logic);
end component;

for all : D_FF use entity work.bind22 (D_FFMaster);
for all : inv use entity work.bind1 (inv_0);
for all : and2 use entity work.bind2 (and2_0);
for all : and3 use entity work.bind3 (and3_0);
for all : or2 use entity work.bind2 (or2_0);
```

for all : or3 use entity work.bind3 (or3_0);
signal Hbar, a1, a2, a3, a4, a5, OR11, OR22 : std_logic;
begin
  DFF0 : D_FF port map ('0', clk, Q(0), Qbar(0));
  inv1 : inv port map (H, Hbar);
  an1 : and2 port map (Hbar, Qbar(1), a1);
  an2 : and3 port map (H, Q(1), Qbar(0), a2);
  r1 : or2 port map (a2, a1, OR11);

  DFF1 : D_FF port map (OR11, clk, Q(1), Qbar(1));
  an3 : and3 port map (Q(2), Qbar(1), Qbar(0), a3);
  an4 : and3 port map (Qbar(0), H, Q(2), a4);
  an5 : and3 port map (Hbar, Qbar(2), Q(1), a5);
  r2 : or3 port map (a3, a4, a5, OR22);

  DFF2 : D_FF port map (OR22, clk, Q(2), Qbar(2));
end Counter_even;

Verilog 3-Bit Synchronous Even Counter with Hold
module CTR_EVEN (H, clk, Q, Qbar);

  input H, clk;
  output [2:0] Q, Qbar;
  D_FFMaster DFF0 (1'b0, clk, Q[0], Qbar[0]);
  not (Hbar, H);
  and (a1, Qbar[1], Hbar);
  and (a2, H, Q[1], Qbar[0]);
  or (OR1, a1, a2);

  D_FFMaster DFF1 (OR1, clk, Q[1], Qbar[1]);
  and (a3, Q[2], Qbar[1], Qbar[0]);
  and (a4, Qbar[0], H, Q[2]);
  and (a5, Hbar, Qbar[2], Q[1]);
  or (OR2, a3, a4, a5);

  D_FFMaster DFF2 (OR2, clk, Q[2], Qbar[2]);
endmodule
FIGURE 4.25 Simulation waveform of even counter with Hold.

Structural Description of a 3 – Bit Synchronous Up/Down Counter

FIGURE 4.26 Symbol logic diagram of an up/down 3-bit counter.
FIGURE 4.27 State diagram of a 3-bit synchronous up/down counter.
FIGURE 4.28  K-maps of a 3-bit synchronous up/down counter.
FIGURE 4.29 Logic diagram of a 3-bit synchronous up/down counter

HDL Description of a 3-Bit Synchronous Up/Down Counter with Clear and Terminal Count—VHDL and Verilog

***Begin Listing***

**VHDL 3-Bit Synchronous Up/Down Counter with Clear and Terminal Count**

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity up_down is
  port (clr, Dir, clk : in std_logic; TC : out std_logic;
  Q, Qbar : buffer std_logic_vector (2 downto 0));
end up_down;

architecture Ctr_updown of up_down is
  --Some simulators will not allow mapping between
  --buffer and out. In this
  --case, change all out to buffer.

  component inv
    port (I1 : in std_logic; O1 : out std_logic);
  end component;

end architecture Ctr_updown;
component and2
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component or2
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component and3
  port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;

component JK_FF
  port (I1, I2, I3 : in std_logic; O1, O2 : buffer std_logic);
end component;

for all : JK_FF use entity work.bind32 (JK_Master);
for all : inv use entity work.bind1 (inv_1);
for all : and2 use entity work.bind2 (and2_4);
for all : and3 use entity work.bind3 (and3_4);
for all : or2 use entity work.bind2 (or2_4);

signal clrbar, Dirbar, J1, K1, J2, K2 : std_logic;
signal s : std_logic_vector (11 downto 0);
begins
  in1 : inv port map (clr, clrbar);
in2 : inv port map (Dir, Dirbar);
an1 : and2 port map (Dirbar, Qbar(0), s(0));
an2 : and2 port map (Dir, Q(0), s(1));
r1 : or2 port map (s(0), s(1), s(2));
an3 : and2 port map (s(2), clrbar, s(3));
r2 : or2 port map (s(3), clr, K1);
r3 : or2 port map (s(2), Q(1), s(4));
an4 : and2 port map (clrbar, s(4), J1);
an5 : and3 port map (Dirbar, Qbar(1), Qbar(0), s(5));
an6 : and3 port map (Dir, Q(1), Q(0), s(6));
r4 : or2 port map (s(6), s(5), s(7));
an7 : and2 port map (s(7), clrbar, J2);
r5 : or2 port map (J2, clr, K2);

JKFF0 : JK_FF port map (clrbar, '1', clk, Q(0), Qbar(0));
JKFF1 : JK_FF port map (J1, K1, clk, Q(1), Qbar(1));
JKFF2 : JK_FF port map (J2, K2, clk, Q(2), Qbar(2));
an8 : and3 port map (clrbar, Qbar(1), Qbar(0), S(8));
an9 : and3 port map (Dirbar, Qbar(2), s(8), S(9));

-- For an8 and an9, we could have used 5-input and gate;
-- but two and gates with a reasonable number of
-- fan-in (three-input) is preferred. Same
-- argument for an10 and an11*/

an10 : and3 port map (clrbar, Q(0), Q(1), S(10));
an11 : and3 port map (Dir, Q(2), s(10), S(11));
r6 : or2 port map (s(9), s(11), TC);
end Ctr_updown;

Verilog 3-Bit Synchronous Up/Down Counter with Clear and Terminal
Count

module up_down (clr, Dir, clk, Q, Qbar, TC);
input clr, Dir, clk;
output [2:0] Q, Qbar;
output TC;

not #1 (clrbar, clr);
not #1 (Dirbar, Dir);
and #4 (s0, Dirbar, Qbar[0]);
and #4 (s1, Dir, Q[0]);
or #4 (s2, s0, s1);
and #4 (s3, s2, clrbar);
or #4 (K1, s3, clr);
or #4 (s4, s2, Q[1]); and
#4 (s5, Dirbar, Qbar[1], Qbar[0]);
or #4 (s6, s5, clr);
or #4 (s7, s6, s5);
and #4 (J2, s7, clrbar);
or #4 (K2, J2, clr);

JK_FF JKFF0 (clrbar, 1'b1, clk, Q[0], Qbar[0]);
JK_FF JKFF1 (J1, K1, clk, Q[1], Qbar[1]);
JK_FF JKFF2 (J2, K2, clk, Q[2], Qbar[2]);

and #4 an8 (s8, clrbar, Qbar[1], Qbar[0]);
and #4 an9 (s9, Dirbar, Qbar[2], s8);

/* For an8 and an9, a five-input and gate could have been used;
but two and gates with a reasonable number of fan-in
(three-input) is preferred. Same argument for an10 and an11*/

and #4 an10 (s10, clrbar, Q[0], Q[1]);
and #4 an11 (s11, Dir, Q[2], s10);
or #4 (TC, s9, s11);
endmodule

FIGURE 4.30 Simulation waveform of an up/down counter.

Structural Description of a 3-Bit Synchronous Decade Counter

FIGURE 4.31 A State diagram of a decade counter.
FIGURE 4.31B  K – maps for a decade counter.
FIGURE 4.32 Logic diagram of a decade counter.

HDL Description of a 3-Bit Synchronous Decade Counter with Terminal Count—VHDL and Verilog

VHDL 3-Bit Synchronous Decade Counter with Terminal Count

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

design entity decade_ctr is
  port (clk : in std_logic;
        Q, Qbar : buffer std_logic_vector (3 downto 0);
        TC : out std_logic);
end design;

design architecture decade_str of decade_ctr is
  --Some simulators will not allow mapping between
  --buffer and out. In this
  --case, change all out to buffer.
  component buf
    port (I1 : in std_logic; O1 : out std_logic);
  end component;
end design;
```
component and2
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

cOMPONENT and3
  port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;

cOMPONENT and4
  port (I1, I2, I3, I4 : in std_logic; O1 : out std_logic);
end component;

cOMPONENT or2
  port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

cOMPONENT or3
  port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;

cOMPONENT D_FF
  port (I1, I2 : in std_logic; O1, O2 : buffer std_logic);
end component;

for all : D_FF use entity work.bind22 (D_FFMaster);
for all : buf use entity work.bind1 (buf_1);
for all : and2 use entity work.bind2 (and2_4);
for all : and3 use entity work.bind3 (and3_4);
for all : and4 use entity work.bind4 (and4_4);
for all : or2 use entity work.bind2 (or2_4);
for all : or3 use entity work.bind3 (or3_4);
signal s : std_logic_vector (6 downto 0);
signal D : std_logic_vector (3 downto 0);
begin
  b1 : buf port map (Qbar(0), D(0));
  DFF0 : D_FF port map (D(0), clk, Q(0), Qbar(0));

-- Assume and gates and or gates have 4 ns propagation
-- delay and invert has 1 ns.
  a1 : and3 port map (Qbar(3), Qbar(1), Q(0), s(0));
  a2 : and2 port map (Q(1), Qbar(0), s(1));
  r1 : or2 port map (s(0), s(1), D(1));
  DFF1 : D_FF port map (D(1), clk, Q(1), Qbar(1));
a3 : and2 port map (Q(2), Qbar(1), s(2));
a4 : and2 port map (Q(2), Qbar(0), s(3));
a5 : and3 port map (Q(1), Q(0), Qbar(2), s(4));
r2 : or3 port map (s(2), s(3), s(4), D(2));
DFF2 : D_FF port map (D(2), clk, Q(2), Qbar(2));

a6 : and2 port map (Q(3), Qbar(0), s(5));
a7 : and4 port map (Q(0), Q(1), Q(2), Qbar(3), s(6));
r3 : or2 port map (s(5), s(6), D(3));
DFF3 : D_FF port map (D(3), clk, Q(3), Qbar(3));
a8 : and4 port map (Q(0), Qbar(1), Qbar(2), Q(3), TC);

end decade_str;

Verilog 3-Bit Synchronous Decade Counter with Terminal Count
module decade_ctr (clk, Q, Qbar, TC);
input clk;
output [3:0] Q, Qbar;
output TC;
wire [3:0] D;
wire [6:0] s;
buf #1 (D[0], Qbar[0]);
D_FFMaster FF0 (D[0], clk, Q[0], Qbar[0]);

/* Assume and gates and or gates have 4 ns propagation
delay and invert has 1 ns */

and #4 (s[0], Qbar[3], Qbar[1], Q[0]);
and #4 (s[1], Q[1], Qbar[0]);
or #4 (D[1], s[0], s[1]);
D_FFMaster FF1 (D[1], clk, Q[1], Qbar[1]);

and #4 (s[2], Q[2], Qbar[1]);
and #4 (s[3], Q[2], Qbar[0]);
and #4 (s[4], Q[1], Q[0], Qbar[2]);
or #4 (D[2], s[2], s[3], s[4]);
D_FFMaster FF2 (D[2], clk, Q[2], Qbar[2]);

and #4 (s[5], Q[3], Qbar[0]);
and #4 (s[6], Q[0], Q[1], Q[2], Qbar[3]);
or #4 (D[3], s[5], s[6]);
D_FFMaster FF3 (D[3], clk, Q[3], Qbar[3]);
and #4 (TC, Q[0], Qbar[1], Qbar[2], Q[3]);
GENERATE (HD), GENERIC (VHDL) AND PARAMETER (VERILOG)

HDL Description of N-Bit Magnitude Comparator Using Generate Statement—
VHDL and Verilog

VHDL N-Bit Magnitude Comparator Using Generate Statement
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity compr_genr is
    generic (N : integer := 3);
    port (X, Y : in std_logic_vector (N downto 0);
          xgty, xlty, xeqy : buffer std_logic);
end compr_genr;

architecture cmpare_str of compr_genr is
    --Some simulators will not allow mapping between
    --buffer and out. In this
    --case, change all out to buffer.

    component full_adder
        port (I1, I2, I3 : in std_logic; O1, O2 : out std_logic);
    end component;

    component inv
        port (I1 : in std_logic; O1 : out std_logic);
    end component;

    component nor2
        port (I1, I2 : in std_logic; O1 : out std_logic);
    end component;

    component and2
        port (I1, I2 : in std_logic; O1 : out std_logic);
    end component;

    signal sum, Yb : std_logic_vector (N downto 0);
    signal carry, eq : std_logic_vector (N + 1 downto 0);

    for all : full_adder use entity work.bind32 (full_add);
    for all : inv use entity work.bind1 (inv_0);
    for all : nor2 use entity work.bind2 (nor2_7);
    for all : and2 use entity work.bind2 (and2_7);
    begin
        carry(0) <= '0';
eq(0) <= '1';
G1 : for i in 0 to N generate
v1 : inv port map (Y(i), Yb(i));
FA : full_adder port map (X(i), Yb(i), carry(i),
    sum(i), carry(i+1));
va : and2 port map (eq(i), sum(i), eq(i+1));
end generate G1;
xgty <= carry(N+1);
xeqy <= eq(N+1);
n1 : nor2 port map (xeqy, xgty, xlty);
end cmpare_str;

Verilog N-Bit Magnitude Comparator Using Generate Statement
module compr_genr (X, Y, xgty, xlty, xeqy);
parameter N = 3;
input [N:0] X, Y;
output xgty, xlty, xeqy;
wire [N:0] sum, Yb;
wire [N+1 : 0] carry, eq;
assign carry[0] = 1'b0;
assign eq[0] = 1'b1;
generate
    genvar i;
    for (i = 0; i <= N; i = i + 1)
        begin : u
            not (Yb[i], Y[i]);
        /* The above statement is equivalent to assign Yb = ~Y if
            outside the generate loop */
        end
        FULL_ADDER FA(X[i], Yb[i], carry[i], sum[i],
            carry[i+1]);
        and (eq[i+1], sum[i], eq[i]);
    end
dengenerate
    assign xgty = carry[N+1];
    assign xeqy = eq[N+1];
    nor (xlty, xeqy, xgty);
endmodule
Structural Description of an N-bit Asynchronous Down Counter Using Generate

![Logic diagram of an n-bit asynchronous down counter when n = 3](image)

**FIGURE 4.33** Logic diagram of an n-bit asynchronous down counter when n = 3

HDL Description of N-Bit Memory Word Using Generate—VHDL and Verilog

**VHDL N-Bit Memory Word Using Generate**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity Memory_word is
  Generic (N : integer := 7);
  port (Data_in : in std_logic_vector (N downto 0); sel, R_W : in std_logic; Data_out : out std_logic_vector (N downto 0));
end Memory_word;

architecture Word_generate of Memory_word is
  component memory_cell
    Port (Sel, RW, Din : in std_logic; O1 : buffer std_logic);
  end component;
  for all : memory_cell use entity work.memory (memory_str);
begin
  G1 : for i in 0 to N generate
    M : memory_cell port map (sel, R_W, Data_in(i), Data_out(i));
  end generate;
end Word_generate;
```

**Verilog N-Bit Memory Word Using Generate**

```verilog
module Memory_Word (Data_in, sel, R_W, Data_out);

parameter N = 7;
input [N:0] Data_in;
input sel, R_W;
output [N:0] Data_out;

generate
  genvar i;
  for (i = 0; i <= N; i = i + 1)
```


begin : u
memory M1 (sel, R_W, Data_in [i], Data_out[i]);
end
endgenerate
endmodule
ASSIGNMENT QUESTIONS

1) Write a HDL Structural Description of full adder
2) What is binding? Explain Binding between a Library and Component in VHDL
3) Explain Binding between Two Modules in Verilog
4) Write a Structural Description of a 2X1 Multiplexer with Active Low Enable
5) Write a Structural Description of a 2x4 decoder with three – state output
6) Write a HDL Description of an SR Latch with NOR Gates
7) Write a Structural Description of a D- Latch
8) Write a Structural Description of a Pulse-Triggered, Master-Slave D Flip-Flop
9) Write a Structural Description of a Master-Slave JK Flip-Flop
10) Write a HDL Structural Description of a 3-bit Ripple – Carry Adder
11) Write a HDL Structural Description of a 3-bit Magnitude Comparator Using 3-Bit adder
12) Write a Structural HDL Description of an SRAM Memory Cell
13) Write a VHDL 3-Bit Synchronous Counter Using JK Master-Slave Flip-Flops
14) Write a Verilog 3-Bit Synchronous Even Counter with Hold
15) Write a Structural HDL Description of a 3-Bit Synchronous Up/Down Counter with Clear and Terminal Count
UNIT 5: PROCEDURES TASKS AND FUNCTIONS

Syllabus of unit 5: Hours: 7
Highlights of Procedures, tasks, and Functions, Procedures and tasks, Functions.
Advanced HDL Descriptions: File Processing, Examples of File Processing

Recommended readings:

1. HDL Programming (VHDL and Verilog) - Nazeih M.Botros- Dreamtech Press
2. Verilog HDL –Samir Palnitkar-Pearson Education
3. VHDL -Douglas perry-Tata McGraw-Hill
4. A Verilog HDL Primer- J.Bhaskar – BS Publications
5. Circuit Design with VHDL-Volnei A.Pedroni-PHI
UNIT 5: PROCEDURES TASKS AND FUNCTIONS

Subprograms

Often the algorithmic model becomes so large that it needs to be split into distinct code segments. And many a times a set of statements need to be executed over and over again in different parts of the model. Splitting the model into subprograms is a programming practice that makes understanding of concepts in VHDL to be simpler. Like other programming languages, VHDL provides subprogram facilities in the form of procedures and functions. The features of subprograms are such that they can be written once and called many times. They can be recursive and thus can be repeated from within the scope. The major difference between procedure and function is that the function has a return statement but a procedure does not have a return statement.

Types Of Subprograms

VHDL provides two sub-program constructs:
Procedure: generalization for a set of statements.
Function: generalization for an expression.

Both procedure and function have an interface specification and body specification.

Declarations of procedures and function
Both procedure and functions can be declared in the declarative parts of:
- entity
- Architecture
- Process
- Package interface
- Other procedure and functions

Formal and actual parameters

The variables, constants and signals specified in the subprogram declaration are called formal parameters.
The variables, constants and signals specified in the subprogram call are called actual parameters.
Formal parameters act as place holders for actual parameters.

Concurrent and sequential programs

Both functions and procedures can be either concurrent or sequential:
- Concurrent functions or procedures exists outside process statement or another subprogram
- Sequential functions or procedures exist only in process statement or another subprogram statement.
Functions

A function call is the subprogram of the form that returns a value. It can also be defined as a subprogram that either defines an algorithm for computing values or describes a behavior. The important feature of the function is that they are used as expressions that return values of specified type. This is the main difference from another type of subprogram: procedures, which are used as statements. The results return by a function can be either scalar or complex type.

Function Syntax

```
< | pure | impure |
  function id | ( parameter_interface_list ) |
  return return_type is
    {declarative part }
  begin
    { sequential statement }
    [label:] return return_value;
  end [ function | [id];

parameter_interface_list <=
( [ constant | signal | identifier[ , , , ]; [in | subtype_indication [ := static_expression ]] )
```

Functions can be either pure (default) or impure. Pure functions always return the same value for the same set of actual parameters. Impure functions may return different values for the same set of parameters. Additionally an impure function may have side effects like updating objects outside their scope, which is not allowed in pure function. The function definition consists of two parts:

1) **Function declaration**: this consists of the name, parameter list and type of values returned by function

2) **Function body**: this contains local declaration of nested subprograms, types, constants, variables, files, aliases, attributes and groups, as well as sequence of statements specifying the algorithm performed by the function.

The function declaration is optional and function body, which contains the copy of it is sufficient for correct specification. However, if a function declaration exists, the function body declaration must exist in the given scope.

**Functional Declaration:**

The function declaration can be preceded by an optional reserved word pure or impure, denoting the character of the function. If the reserved word is omitted it is assumed to be
pure by default.
The function name (id), which appears after the reserved word function can either be an
identifier or an operator symbol. Specification of new functions for existing operators is
allowed in VHDL and is called OPERATOR OVERLOADING.
The parameters of the function are by definition INPUTS and therefore they do not need
to have the mode (direction) explicitly specified. Only constants, signals and files can be
function parameters. The object class is specified by using the reserved words (constant,
signal or file respectively) preceding the parameter name. If no reserved word is used, it
is assumed that the parameter is a CONSTANT.
In case of signal parameters the attributes of the signal are passed into the function,
except for `STABLE, `QUIET, `TRANSACTION and `DELAYED, which may not be
accessed within the function.
Variable class is NOT allowed since the result of operations could be different when
different instantiations are executed. If a file parameter is used, it is necessary to specify
the type of data appearing in the opened file.

Function Body:

Function body contains a sequence of statements that specify the algorithm to be realized
within the function. When the function is called, the sequence of statements is executed.
A function body consists of two parts: declarations and sequential statements. At the end
of the function body, the reserved word END can be followed by an optional reserved
word FUNCTION and the function name.

Pure And Impure Functions

Pure Functions
· Function Does Not Refer to Any Variables or Signals Declared by Parent
· Result of Function Only Depends on Parameters Passed to It
· Always Returns the Same Value for Same Passed Parameters No Matter When It
  Is Called
· If Not Stated Explicitly, a Function Is Assumed to Be Pure

Impure Function
· Can State Explicitly and Hence Use Parents’ Variables and/or Signals for
  Function Computation
· May Not Always Return the Same Value

Function Calling
· Once Declared, Can Be Used in Any Expression
· A Function Is Not a Sequential Statement So It Is Called As Part of an Expression

[ label : ] function_name
[ parameter_association_list ] ;

[ CITSTUDENTS.IN ]
Page 126
Example 1

```haskell
type int_data is file of natural;

function func_1 (a, b, x : real) return real;

function "*" (a, b: integer_new) return integer_new;

function add_signals (signal in1, in2: real) return real;

function end_of_file (file file_name: int_data) return boolean;
```

- The first function name above is called func_1, it has three parameters A, B and X, all of the REAL types and returns a value also of REAL type.
- The second function defines a new algorithm for executing multiplication. Note that the operator is enclosed in double quotes and plays the role of the function name.
- The third is based on the signals as input parameters, which is denoted by the reserved word signal preceding the parameters.
- The fourth function declaration is a part of the function checking for end of file, consisting of natural numbers. Note that the parameter list uses the Boolean type declaration.

Example 2

```haskell
function transcod_1(value: in std_logic_vector (0 to 7))
return
  std_logic_vector is
begin
  case value is
  when "00000000" => return "01010101";
  when "01010101" => return "00000000";
  when others => return "11111111";
  end case;

end transcod_1;
```
The case statement has been used to realize the function algorithm. The formal parameter appearing in the declaration part is the value constant, which is a parameter of the std_logic_vector type. This function returns a value of the same type.

**Example 3**

```plaintext
function func_3 (constant A, B, X: real) return real is
begin
    return A*X**2+B;
end func_3;
```

The formal parameters: A, B and X are constants of the real type. The value returned by this function is a result of calculating the A*X**2+B expression and it is also of the real type.

**Procedure**

A procedure is a subprogram that defined as algorithm for computing values or exhibiting behavior. Procedure call is a statement which encapsulates a collection of sequential statements into a single statement. It may zero or more values. It may execute in zero or more simulation time.

- Procedure declarations can be nested. Allows for recursive calls
- Procedures can call other procedures
- Procedure must be declared before use. It can be declared in any place where declarations are allowed, however the place of declaration determines the scope
- Cannot be used on right side of signal assignment expression since doesn’t return value

**Procedure Syntax.**
**Procedure**

Procedure is a form of subprogram. It contains local declarations and a sequence of statements. Procedure can be called in the place of architecture. The procedure definition consists of two parts:

- The **PROCEDURE DECLARATION**, which contains the procedure name and the parameter list required when the procedure is called.
- The **PROCEDURE BODY**, which consists of local declarations and statements required to execute the procedure.

**Procedure Declaration**

The procedure declaration consists of the procedure name and the formal parameter list. In the procedure specification, the identifier and optional formal parameter list follow the reserved word procedure (example 1).

Objects classes **CONSTANTS**, **VARIABLES**, **SIGNALS**, and files can be used as formal parameters. The class of each parameter is specified by the appropriate reserved word, unless the default class can be assumed. In case of constants, variables, and signals, the parameter mode determines the direction of the information flow and it decides which formal parameters can be read or written inside the procedure. Parameters of the file type have no mode assigned.

There are three modes available: in, out, and inout. When in mode is declared and object class is not defined, then by default it is assumed that the object is a CONSTANT. In case of inout and out modes, the default class is VARIABLE. When a procedure is called formal parameters are substituted by actual parameters, if a formal parameter is a constant, then actual parameter must be an expression. In case of formal parameters such as signal, variable and file, the actual parameters such as class. Example 2 presents several procedure declarations with parameters of different classes and modes. A **procedure can be declared without any parameters**.

**Procedure Body**

Procedure body defines the procedures algorithm composed of **SEQUENTIAL** statements. When the procedure is called it starts executing the sequence of statements declared inside the procedure body.

The procedure body consists of the subprogram declarative part after the reserved word ‘IS’ and the subprogram statement part placed between the reserved words ‘BEGIN’ and ‘END’. The key word procedure and the procedure name may optionally follow.
the END reserve word.
Declarations of a procedure are local to this declaration and can declare subprogram declarations, subprogram bodies, types, subtypes, constants, variables, files, aliases, attribute declarations, attribute specifications, use clauses, group templates and group declarations. (example 3)
A procedure can contain any sequential statements (including wait statements). A wait statement, however, cannot be used in procedure s which are called from process with a sensitivity list or form within a function. Examples 4 and 5 present two sequential statements specifications.

Procedure Parameter List
- Do not have to pass parameters if the procedure can be executed for its effect on variables and signals in its scope. But this is limited to named variables and signals
- Class of object(s) that can be passed are
  - Constant (assumed if mode is in)
  - Variable (assumed if mode is out)
  - Signals are passed by reference (not value) because if wait statement is executed inside a procedure, the value of a signal may change before the rest of the procedure is calculated. If mode is ‘inout’, reference to both signal and driver are passed

In And Out Mode Parameter
- If the mode of a parameter is not specified it is assumed to be “in”.
- A procedure is not allowed to assign a value to a “in” mode parameter.
- A procedure can only read the value of an in mode parameter.
- A procedure can only assign a value to an out mode parameter.
- In the case of a variable “in” mode parameter the value of the variable remains constant during execution of the procedure.
- However, this is not the case for a signal in “in” mode parameter as the procedure can contain a “wait” statement, in which case the value of the signal might change.

Default values
- VHDL permits the specification of default values for constant and “in” mode variable classes only.
- If a default value is specified, the actual parameter can be replaced by the keyword “open” in the call.

Procedure examples
Example 1

```
procedure procedure_1(variable x, y: inout real);
```

The above procedure declaration has two formal parameters: bi-directional X and Y of real type.

Example 2
Procedure proc_1 has two formal parameters: the first one is a constant and it is in the mode in and of the integer type, the second one is an output variable of the integer type.
Procedure proc_2 has only one parameter, which is a bi-directional signal of type std_logic.

Procedure Call

A procedure call is a sequential or concurrent statement, depending on where it is used. A sequential procedure call is executed whenever control reaches it, while a concurrent procedure call is activated whenever any of its parameters of in or inout mode changes its value.

Differences Between Functions And Procedures

<table>
<thead>
<tr>
<th>FUNCTIONS</th>
<th>PROCEDURES</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Returns only one argument.</td>
<td>1) it can return more than one argument</td>
</tr>
<tr>
<td>2) Lists parameters are constant by default but can be overridden by using signal.</td>
<td>2) list parameter s can be in/out/inout, it can be only in for constant and can be out/inout for variable.</td>
</tr>
<tr>
<td>3) function by itself is not complete statement.</td>
<td>3) procedure is a complete statement</td>
</tr>
<tr>
<td>4) function has two parts: function declaration and function call.</td>
<td>4) procedure has two parts: procedure declaration and procedure call.</td>
</tr>
</tbody>
</table>

Tasks (Verilog)

Tasks are Verilog subprograms. They can be implemented to execute specified routines repeatedly. The format in which the task is written can be divided into two parts: the declaration and the body of the task. In the declaration, the name of the task is specified, and the outputs and inputs of the task are listed. An example of task declaration is:
task addr;
output cc,dd;
input aa,bb;

addr is the name(identifier) of the task. The outputs are cc and dd, and the inputs are aa and bb, task is a predefined word. The body of the task shows the relationship between the outputs and the inputs. An example of the body a task is:

begin
  cc = aa ^ bb;
  ................
end
endtask

The body of the task cannot include always or initial. A task must be called within the behavioral statement always or initial. An example of calling the task addr is as follows.

  ................
always @ (a, b)
begin
  addr(c, d, a, b);
end

addr is the name of the task, and inputs a and b are passed to aa and bb. The outputs of the task cc and dd, after execution, are passed to c and d, respectively.

**HDL Description of a Full Adder Using Procedure and Task—VHDL and Verilog**

**VHDL Full Adder Using Procedure**

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_add is
  port (x, y, cin : in std_logic; sum, cout : out std_logic);
end full_add;

architecture two_halfs of full_add is
  -- The full adder is built from two half adders

  procedure Haddr(sh, ch : out std_logic; ah, bh : in std_logic) is
--This procedure describes a half adder
begin
  sh := ah xor bh;
  ch := ah and bh;
end Haddr;

begin

addfull : process (x, y, cin)
  variable sum1, c1, c2, tem1, tem2 : std_logic;
begin
  Haddr (sum1, c1, y, cin);
  Haddr (tem1, c2, sum1, x);
  --The above two statements are calls to the procedure Haddr
  tem2 := c1 or c2;
  sum <= tem1;
  cout <= tem2;
end process;
end two_halfs;

Verilog Full Adder Using Task
module Full_add (x, y, cin, sum, cout);
//The full adder is built from two half adders
input x, y, cin;
output sum, cout;
reg sum, sum1, c1, c2, cout;
always @ (x, y, cin)
begin
  Haddr (sum1, c1, y, cin);
  Haddr (sum, c2, sum1, x);
  //The above two statements are calls to the task Haddr.
  cout = c1 | c2;
end

task Haddr;
//This task describes the half adder
output sh, ch;
input ah, bh;
begin
  sh = ah ^ bh;
  ch = ah & bh;
end
detask
HDL Description of an N-Bit Ripple Carry Adder Using Procedure and Task—VHDL and Verilog

**VHDL N-Bit Ripple Carry Adder Using Procedure**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity adder_ripple is
  generic (N : integer := 3);
  port (x, y : in std_logic_vector (N downto 0);
        cin : in std_logic;
        sum : out std_logic_vector (N downto 0);
        cout : out std_logic);
end adder_ripple;

architecture adder of adder_ripple is
  procedure Faddr (sf, cof : out std_logic;
                   af, bf, cinf : in std_logic) is
  --This procedure describes a full adder
  begin
    sf := af xor bf xor cinf;
    cof := (af and bf) or (af and cinf) or (bf and cinf);
  end Faddr;

  begin
    addrpl : process (x, y, cin)
    variable c1, c2, tem1, tem2 : std_logic;
    variable cint : std_logic_vector (N+1 downto 0);
    variable sum1 : std_logic_vector (N downto 0);
    begin
      cint(0) := cin;
      for i in 0 to N loop
        Faddr (sum1(i), cint(i+1), x(i), y(i), cint(i));
        --The above statement is a call to the procedure Faddr
      end loop;
      sum <= sum1;
      cout <= cint(N+1);
    end process;

  end adder;

Verilog N-Bit Ripple Carry Adder Using Task
module adder_ripple (x, y, cin, sum, cout);
parameter N = 3;
input [N:0] x, y;
input cin;
output [N:0] sum;
output cout;

reg [N+1:0] cint;
reg [N:0] sum;
reg cout;
integer i;
always @ (x, y, cin)
begin
  cint[0] = cin;
  for (i = 0; i <= N; i = i + 1)
  begin
    Faddr (sum[i], cint[i+1], x[i], y[i], cint[i]);
    //The above statement is a call to task Faddr
  end
  cout = cint[N+1];
end

task Faddr;
//The task describes a full adder
output sf, cof;
input af, bf, cinf;
begin
  sf = af ^ bf ^ cinf;
  cof = (af & bf) | (af & cinf) | (bf & cinf);
end
endtask
endmodule

HDL Code for Converting an Unsigned Binary to an Integer Using Procedure and Task—VHDL and Verilog

***Begin Listing***

**VHDL: Converting an Unsigned Binary to an Integer Using Procedure**
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all; --This Library is for type “unsigned”
entity Bin_Int is
generic (N : natural := 3);
port (X_bin : unsigned (N downto 0);
    Y_int : out natural; Z : out std_logic);
--Y is always positive
end Bin_Int;

architecture convert of Bin_Int is

procedure bti (bin : in unsigned; int : out natural;
    signal Z : out std_logic) is
    -- the procedure bti is to change binary to integer
    -- Flag Z is chosen to be a signal rather than a variable
    -- Since the binary vector is always positive,
    -- use type natural for the output of the procedure.
    variable result : natural;
begin

    result := 0;
    for i in bin'Range loop
        --bin'Range represents the range of the unsigned vector bin
        --Range is a predefined attribute
        if bin(i) = '1' then
            result := result + 2**i;
        end if;
    end loop;
    int := result;
    if (result = 0) then
        Z <= '1';
    else
        Z <= '0';
    end if;
begin
    process (X_bin)
        variable tem : natural;

        begin
            bti (X_bin, tem, Z);
            Y_int <= tem;
        end process;
    end convert;

Verilog: Converting an Unsigned Binary to an Integer Using Task
module Bin_Int (X_bin, Y_int, Z);
  parameter N = 3;
  input [N:0] X_bin;
  output integer Y_int;
  output Z;
  reg Z;
  always @(X_bin)
  begin
    bti (Y_int, Z, N, X_bin);
  end

  task bti;
  parameter P = N;
  output integer int;
  output Z;
  input N;
  input [P:0] bin;
  integer i, result;
  begin
    int = 0;
    // change binary to integer
    for (i = 0; i <= P; i = i + 1)
      begin
        if (bin[i] == 1)
          int = int + 2**i;
        end
      if (int == 0)
        Z = 1'b1;
      else
        Z = 1'b0;
  end
  endtask
endmodule
Figure 6.2 simulation output for binary to integer conversion.

**HDL Code for Converting a Fraction Binary to Real Using Procedure and Task—VHDL and Verilog**

**VHDL: Converting a Fraction Binary to Real Using Procedure**

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

entity Bin_real is
  generic (N : integer := 3);
  port (X_bin : in std_logic_vector (0 to N); Y : out real);
end Bin_real;

architecture Bin_real of Bin_real is
  procedure binfloat (a : in std_logic_vector; float : out real) is
    variable flt : real;
    begin
      flt := 0.0;
      rl : for i in N downto 0 loop
        if (a(i) = '1') then
          flt := flt + 1.0 / 2**(i+1);
          --The above statement multiplies each bit by its weight
        end if;
      end loop rl;
      float := flt;
      end binfloat;

    begin
      rel : process (X_bin)
        variable temp : real;
        begin
          binfloat (X_bin, temp);
          Y <= temp;
        end process rel;
    end Bin_real;
```

**Verilog: Converting a Fraction Binary to Real Using Task**

```
module Bin_real (X_bin);
  parameter N = 3;
  input [N:0] X_bin;
  real Z;
```
always @ (X_bin)
begin
  binfloat (X_bin, Z);
end

task binfloat;
parameter P = N;
input [0:P] a;
output real float;
integer i;
begin
  float = 0.0;
  for (i = 0; i <= P; i = i + 1)
  begin
    if (a[i] == 1)
      float = float + 1.0 / 2**(i+1);
  //The above statement multiplies each bit by its weight
  end
end
task
endmodule

<table>
<thead>
<tr>
<th>X_bin</th>
<th>1000</th>
<th>0100</th>
<th>0010</th>
<th>0001</th>
<th>0101</th>
<th>1100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>0.5</td>
<td>0.25</td>
<td>0.125</td>
<td>0.0625</td>
<td>0.3125</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Figure 6.3 Simulation output for fraction binary conversion to real.

HDL Code for Converting an Unsigned Integer to Binary Using Procedure and Task—VHDL and Verilog

**VHDL: Converting an Unsigned Integer to Binary Using Procedure**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Int_Bin is
  generic (N : integer := 3);
  port (X_bin : out std_logic_vector (N downto 0);
        Y_int : in integer;
end entity Int_Bin;
```

```vhdl
architecture Behavioral of Int_Bin is
begin
process (Y_int)
begin
  if (Y_int = 1) then
    Y_int <= 1;
  else
    Y_int <= 0;
  end if;
end process;

process (Y_int)
begin
  for (i = 0; i <= N; i = i + 1)
  begin
    if (Y_int(i) == 1)
      X_bin(i) <= '1';
    else
      X_bin(i) <= '0';
  end
end process;
end architecture Behavioral;
```

**Verilog: Converting an Unsigned Integer to Binary Using Task**

```verilog
module Int_Bin(#N)
  input [N-1:0] X Bin;
  output [N-1:0] Y Int;

  task binfloat; // Task for converting unsigned integer to binary
  parameter P = N;
  input [0:P] a;
  output real float;
  integer i;
  begin
    float = 0.0;
    for (i = 0; i <= P; i = i + 1)
    begin
      if (a[i] == 1)
        float = float + 1.0 / 2**(i+1);
    //The above statement multiplies each bit by its weight
    end
  end
end task

begin
  Y Int <= binfloat(X Bin);
end
```

```verilog
always @ (X Bin)
begin
  binfloat (X Bin, Z);
end
```
flag_even : out std_logic);
end Int_Bin;

architecture convert of Int_Bin is

procedure itb (bin : out std_logic_vector;
    signal flag : out std_logic;
    N : in integer; int : inout integer) is

    -- The procedure itb is to convert the integer to binary
    -- The dimension of bin does not have to be specified
    -- at the above declaration statement; the procedure
    -- can determine the dimension of bin later in its body.

    begin
        if (int MOD 2 = 0) then
            -- The above statement checks int to see if it is even.
            flag <= '1';
        else
            flag <= '0';
        end if;
        for i in 0 to N loop
            if (int MOD 2 = 1) then
                bin (i) := '1';
            else
                bin (i) := '0';
            end if;
        -- perform integer division by 2
        int := int/2;
        end loop;
    end itb;

    begin
        process (Y_int)
            variable tem : std_logic_vector (N downto 0);
            variable tem_int : integer ;

            begin
                tem_int := Y_int;
                itb (tem, flag_even, N, tem_int);
                X_bin <= tem;
            end process;
    end convert;
***Begin Verilog***

Verilog: Converting an Unsigned Integer to Binary Using Task

module Int_Bin (X_bin, flag_even, Y_int);

/* In general Verilog, in contrast to VHDL, does not
strictly differentiate between integers and binaries;
for example, if bin is declared as a binary of width 4,
bin = bin/2 can be written, and the Verilog, but not
VHDL, performs this division as if bin is integer.
In the following, the corresponding VHDL program in
Listing 6.5a is just translated to practice with
the command task */

parameter N = 3;
output [N:0] X_bin;
output flag_even;
input [N:0] Y_int;
reg [N:0] X_bin;
reg flag_even;
always @ (Y_int)
begin
itb (Y_int, N, X_bin, flag_even);
end

task itb;
parameter P = N;
input integer int;
input N;
output [P:0] bin;
output flag;
integer j;
begin
if (int %2 == 0)
// The above statement checks int to see if it is even.
flag = 1'b1;
else
flag = 1'b0;
for (j = 0; j <= P; j = j + 1)
begin
if (int %2 == 1)
bin[j] = 1;
else
  bin[j] = 0;
  int = int/2;
end
end
endtask
endmodule

Figure 6.4 Simulation output for integer conversion to binary.

**HDL Code for Converting a Signed Binary to Integer Using Procedure and Task—VHDL and Verilog.**

**VHDL: Converting a Signed Binary to Integer Using Procedure**
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_btoIn is
  generic (N : integer := 3);
  port (X_bin : in signed (N downto 0); Y_int : out integer;
  even_parity : out std_logic );
end signed_btoIn;

architecture convert of signed_btoIn is

procedure sbti (binsg : in signed; M : in integer;
  int : out integer; signal even : out std_logic) is
The procedure sbti is to change signed binary to integer and also to find whether the parity of the binary is odd or even. The dimension of "sbin" does not have to be specified at the declaration statement; it can be declared later in the body of the procedure.

```verbatim
variable result, parity : integer;
begin
  result := 0;
  for i in 0 to M loop if binsg(i) = '1' then
    result := result + 2**i;
    parity := parity + 1;
  end if;
end loop;

  if (binsg(M) = '1') then
    result := result - 2**(M+1);
  end if;
  int := result;

  if (parity mod 2 = 1) then
    even <= '0';
  else
    even <= '1';
  end if;
end sbti;
begin
  process (X_bin)
  variable tem : integer;
  begin
    sbti (X_bin, N, tem, even_parity);
    Y_int <= tem;
  end process;
end convert;

Verilog: Converting a Signed Binary to Integer Using Task
module signed_btoIn(X_bin, Y_int, even_parity);
/* In general, Verilog (in contrast to VHDL) does not strictly differentiate between integers and binaries; for example if bin is declared as binary of width 4,
write bin = bin/2, and the Verilog (but not VHDL) will perform this division. In the following, just translate the corresponding VHDL counterpart program. */

```vhdl
parameter N = 3;
input signed [N:0] X_bin;
output integer Y_int;
output even_parity;
reg even_parity;

always @ (X_bin)
begin
  sbti (Y_int, even_parity, N, X_bin);
end

task sbti;
parameter P = N;
output integer int;
output even;
input N;
input [P:0] bin;
integer i;
reg parity;

begin
  int = 0;
  parity = 0;
  //change binary to integer
  for (i = 0; i <= P; i = i + 1)
  begin
    if (bin[i] == 1)
      begin
        int = int + 2**i;
        parity = parity + 1;
      end
  end
  if ((parity % 2) == 1)
    even = 0;
  else
    even = 1;

  if (bin [P] == 1)
    int = int - 2**(P+1);
```

This VHDL code performs the same function as the Verilog code but includes the division step to convert the binary input to an integer. It also calculates the even parity of the input binary number and handles the special case of the last bit by subtracting 2**(P+1) when necessary.
end
endtask

endmodule

<table>
<thead>
<tr>
<th>X_bin</th>
<th>0111</th>
<th>1011</th>
<th>1110</th>
<th>1101</th>
<th>0011</th>
<th>1100</th>
</tr>
</thead>
</table>

| Y_int | 7    | -5   | -2   | -3   | 3    | -4   |

Figure 6.5 Simulation output for converting a signed binary to integer.

**VHDL Code for Converting an Integer to Signed Binary Using Procedure**

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity signed_IntToBin is
  generic (N : integer := 3);
  port (X_bin : out signed (N downto 0); Y_int : in integer);
end signed_IntToBin;

architecture convert of signed_IntToBin is

procedure sitb (sbin : out signed; M, int : in integer) is  
  -- The procedure sitb is to convert integer into signed binary.
  -- The dimension of "sbin" does not have to be specified at the 
  -- declaration statement; it can be declared later in the 
  -- body of the procedure
  variable temp_int : integer;
  variable flag : std_logic;
  variable bin : signed (M downto 0);
begin
  if (int < 0) then
    temp_int := - int;
    flag := '1';
    --if flag = 1, the number is negative

    end if;
else
    temp_int := int;
end if;

for i in 0 to M loop
    if (temp_int MOD 2 = 1) then
        bin (i) := '1';
    else
        bin (i) := '0';
    end if;
end loop;

--integer division by 2
temp_int := temp_int/2;

if (flag = '1') then
    sbin := - bin;
else sbin := bin;
end if;
end sitb;

begin
    process (Y_int)
        variable tem : signed (N downto 0);
        begin
            sitb(tem, N, Y_int);
            X_bin <= tem;
        end process;
end convert;

HDL Code for Signed Vector Multiplication Using Procedure and Task—VHDL and Verilog

VHDL: Signed Vector Multiplication Using Procedure
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity Vector_Booth is
    generic (N : integer := 3);
    port (a0, a1, a2, b0, b1, b2 : in signed (N downto 0);
        d : out signed (3*N downto 0));
end Vector_Booth;
architecture multiply of Vector_Booth is
    procedure booth (X, Y : in signed (3 downto 0);
    begin
Z : out signed (7 downto 0)) is
--Booth algorithm here is restricted to 4x4 bits.
--It can be adjusted to multiply any NxN bits.
variable temp : signed (1 downto 0);
variable sum : signed (7 downto 0);
variable E1 : unsigned (0 downto 0);
variable Y1 : signed (3 downto 0);
begin

sum := "00000000"; E1 := "0";
for i in 0 to 3 loop
    temp := X(i) & E1(0);
    Y1 := -Y;
    case temp is
    when "10" => sum (7 downto 4) :=
                sum (7 downto 4) + Y1;
    when "01" => sum (7 downto 4) :=
                sum (7 downto 4) + Y;
    when others => null;
    end case;
    sum := sum srl 1;
    sum (7) := sum(6);
    E1(0) := x(i);
end loop;
if (y = "1000") then
    --If Y = 1000; then according to the code,
    --Y1 = 1000 (-8 not 8 because Y1 is 4 bits only).
    --The statement sum = -sum adjusts the answer.

    sum := -sum;
end if;
Z := sum;
end booth;

procedure sitb (sbin : out signed; M, int : in integer) is
-- The procedure sitb is to convert integer into signed binary.
-- The dimension of "sbin" does not have to be specified
-- at the declaration statement; it can be declared
-- later in the body of the procedure.
variable temp_int : integer;
variable flag : std_logic;
variable bin : signed (M downto 0);
begin
if (int < 0) then
  temp_int := -int;
  flag := '1';
else
  temp_int := int;
end if;

for i in 0 to M loop
  if (temp_int MOD 2 = 1) then
    bin (i) := '1';
  else
    bin (i) := '0';
  end if;
  temp_int := temp_int/2;
end loop;
if (flag = '1') then
  sbin := -bin;
else
  sbin := bin;
end if;
end sitb;

procedure sbti (binsg : in signed; M : in integer;
  int : out integer) is
  -- The procedure sbti is to change signed binary to integer.
  -- We do not have to specify the dimension of "sbin"
  -- at the declaration statement; it can be declared
  -- later in the body of the procedure.

  variable result : integer;
  begin
    result := 0;
    for i in 0 to M loop if
      binsg(i) = '1' then
        result := result + 2**i;
      end if;
    end loop;

    if (binsg(M) = '1') then
      result := result - 2**(M+1);
    end if;
    int := result;
  end sbti;
begin
process (a0, b0, a1, b1, a2, b2)
variable tem0, tem1, tem2 : signed ((2*N + 1) downto 0);
variable d_temp : signed (3*N downto 0);
variable temi0, temi1, temi2, temtotal : integer;

begin
--Find the partial products a0b0, a1b1, a2b2
booth (a0, b0, tem0);
booth (a1, b1, tem1);
booth (a2, b2, tem2);

-- Change the partial products to integers
sbti (tem0, (2*N+1), temi0);
sbti (tem1, (2*N+1), temi1);
sbti (tem2, (2*N+1), temi2);

-- Find the total integer sum of partial products
temtotal := temi0 + temi1 + temi2;

-- Change the integer to binary
sitb (d_temp, 3*N, temtotal);

d <= d_temp;
end process;
end multiply;

Verilog: Signed Vector Multiplication Using Task
module Vector_Booth (a0, a1, a2, b0, b1, b2, d);
parameter N = 3;
input signed [N:0] a0, a1, a2, b0, b1, b2;
output signed [3*N : 0] d;
reg signed [2*N+1 : 0] tem0, tem1, tem2;
reg signed [3*N : 0] d;
always @(a0, b0, a1, b1, a2, b2)
begin
booth (a0, b0, tem0);
//booth is a task to multiply a0 x b0 = tem0
booth (a1, b1, tem1);
booth (a2, b2, tem2);
d = tem0 + tem1 + tem2;
end
task booth;
input signed [3:0] X, Y;
output signed [7:0] Z;
reg signed [7:0] Z;
reg [7:0] Z;
reg [1:0] temp;
integer i;
reg E1;
reg [3:0] Y1;
begin
    Z = 8'd0;
    E1 = 1'd0;

    for (i = 0; i < 4; i = i + 1)
    begin
        temp = {X[i], E1}; //This is catenation
        Y1 = -Y; //Y1 is the 2' complement of Y
        case (temp)
            default : begin end
        endcase
        Z = Z >> 1; /*This is a logical shift of one position to
         *the right*/
        Z[7] = Z[6];
        /*The above two statements perform arithmetic shift
         *where the sign of the number is preserved after
         *the shift.*/
        E1 = X[i];
    end

    if (Y == 4'b1000) Z = -Z;

    /* If Y = 1000, then Y1 = 1000 (should be 8 not -8).
       This error is because Y1 is 4 bits only.
       The statement Z = -Z adjusts the value of Z. */
    end
endtask
endmodule
Figure 6.6 Simulation output for vector multiplication.

**Enzyme – Substrate Activity Using Procedure and Task**

![Diagram]

(a) Strong

(b) Weak

Figure 6.7 Binding between substrate and enzyme.
Figure 6.8 Relationship between the substrate concentration $S$ and rate of reaction $V$.

**HDL Description for Enzyme Activity Using Procedure and Task—VHDL and Verilog**

**VHDL: Enzyme Activity Using Procedure**

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

-- A description of enzyme-substrate binding mechanism.
-- $S= (V_{max} \cdot S)/(S + M)$ where $S$ is the substrate concentration,
-- $M$ is the dissociation constant, and $V_{max}$ is the maximum
-- rate of reaction. In this example, $V_{max} = 1$.
-- The inputs are $S$ and $M$ in binary (integer); the output $v$
-- is in Q4 format. This means that $v$ is always less than
-- one, with the binary point placed to the left of the
-- most significant bit. For example, if $v = 1010$, the decimal
-- equivalent is $0.5 + 0.125 = 0.625$.
-- To calculate $v$, convert $S$ and $M$ to unsigned integers,
-- find the real value of $(S/(S + M))$, convert this real
-- value to Q4 by multiplying it with $2^{4} = 16$, and
-- convert the integer to binary.

entity enzyme_beh is
  port (S : in std_logic_vector (3 downto 0);
       v : out std_logic_vector (3 downto 0);
       M : in std_logic_vector(3 downto 0));
end enzyme_beh;
```

entity enzyme_beh is
  port (S : in std_logic_vector (3 downto 0);
       v : out std_logic_vector (3 downto 0);
       M : in std_logic_vector(3 downto 0));
end enzyme_beh;
architecture enzyme of enzyme_beh is
procedure bti (bin : in std_logic_vector; int : out integer) is
variable result : integer;
begin
result := 0;
for i in 0 to 3 loop if
bin(i) = '1' then result := result + 2**i; end if;
end loop;
int := result;
end bti;
procedure itb (int : in integer; bin : out std_logic_vector) is
-- the procedure itb is to change the integer to binary
variable temp : integer;
begin
temp := int;
for j in 0 to 3 loop
if (temp MOD 2 = 1) then
bin (j) := '1';
else bin (j) := '0';
end if;
temp := temp/2;
end loop;
end itb;
Procedure rti (r : in real; int : out integer) is
-- This procedure converts a real value to an integer;
-- the procedure is effective for small numbers <100.
-- For very large numbers, the procedure is slow, since the
-- execution time of the procedure is proportional to the
-- value of the real number to be converted.
variable temp : real;
variable intg : integer := 0;
begin

temp := r;
while temp >= 0.5 loop
intg := intg + 1;
temp := r - 1.0 * intg ;
end loop;
int := intg;
end rti;

begin
  P1 : process(S, M)
  Variable S1, M1, v1 : integer;
  Variable vr, vq4, vmax : real;
  variable tem : std_logic_vector (3 downto 0);
  begin
    bti (S, s1); bti (M, m1);
    vmax := 1.0;
    vr := vmax*(1.0 *S1) / (S1 * 1.0 + M1 * 1.0);
    vq4 := vr * 2**4;
    rti (vq4, v1);
    itb (v1, tem);
    v <= tem;
  end process P1;
  end enzyme;

Verilog: Enzyme Activity Using Task
/* A description of enzyme-substrate binding mechanism.
S= (Vmax *S)/(S + M), where S is the substrate
concentration, M is the dissociation constant,
and Vmax is the maximum rate of reaction. In this example,
Vmax =1. The inputs are S and M in binary (integer);
the output v is in Q4 format. This means that v is always
less than 1, with the binary point placed to the left of
the most significant bit. For example, if v = 1010, the
decimal equivalent is .5 + .125 = 0.625. To calculate V,
find the real value of (S /(S + M)), convert this real
value to Q4 by multiplying it with 2**4 =16, and convert
the integer to binary. */

module enzyme_beh (S, M, V);
  input [3:0] S, M;
  output [3:0] V;
  integer vmax;
  reg [3:0] V;
  real vr;
  always @(S, M)
  begin
    vmax = 1;
    vr = vmax * (1.0 * S) / (S * 1.0 + M * 1.0);
    vr = vr * 2**4;
    rti (vr, V);
  end

task rti;
/* This task can be replaced by just one statement, v1 = r.
   Verilog, in contrast to VHDL, can handle different
   types of the assignment statement. Verilog finds the
   equivalent integer value v1 for the real r. The task has
   been designed here only to match the VHDL procedure rti. */

input real r;
output [3:0] v1;
real temp;
begin
  temp = r;
  v1 = 4'b0000;
  while (temp >= 0.5)
  begin
    v1 = v1 + 1;
    temp = r - 1.0 * v1;
  end
  end
endtask
endmodule

FIGURE:6.9 Simulation output of the relationship between substrate concentration S and rate of reaction V for M = 3 units.
**Example of a VHDL Function**

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Func_exm is
port (a1, b1 : in std_logic; d1 : out std_logic);
end Func_exm;

architecture Behavioral of Func_exm is
function exp (a, b : in std_logic) return std_logic is
variable d : std_logic;
begin
  d := a xor b;
  return d;
end function exp;

begin
  process (a1, b1)
  begin
    d1 <= exp (a1, b1);
    --The above statement is a function call
  end process;
end Behavioral;

**Verilog Function That Calculates exp = a XOR b**

module Func_exm (a1, b1, d1);
input a1, b1;
output d1;
reg d1;

always @(a1, b1)
begin
  /*The following statement calls the function exp
   and stores the output in d1.*/
  d1 = exp (a1, b1);
end

function exp ;
input a, b;
begin
  exp = a ^ b;
end
HDL Function to Find the Greater of Two Signed Numbers—VHDL and Verilog

**VHDL Function to Find the Greater of Two Signed Numbers**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;

entity greater_2 is
  port (x, y : in signed (3 downto 0); z : out signed (3 downto 0));
end greater_2;

architecture greater_2 of greater_2 is
function grt (a, b : signed (3 downto 0)) return signed is
  -- The above statement declares a function by the name grt.
begin
  if (a >= b) then
    temp := a;
  else
    temp := b;
  end if;
  return temp;
end grt;

begin
  process (x, y)
  begin
    z <= grt (x, y); -- This is a function call.
  end process;
end greater_2;
```

**Verilog Function to Find the Greater of Two Signed Numbers**

```verilog
module greater_2 (x, y, z);
  input signed [3:0] x;
  input signed [3:0] y;
  output signed [3:0] z;
  reg signed [3:0] z;

  always @ (x, y)
```
begin
z = grt (x, y); //This is a function call.
end

function [3:0] grt;

/*The above statement declares a function by the name grt;
grt is also the output of the function*/

input signed [3:0] a, b;
/*The above statement declares two inputs to the function;
both are 4-bit signed numbers.*/

begin
if (a >= b)
grt = a;
else
grt = b; end
endfunction

endmodule

HDL Code for y = 0 < x < 1—VHDL and Verilog

VHDL Floating Sum Description
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity segma is
port (x : in std_logic_vector (0 to 3);
      y: out std_logic_vector (15 downto 0));
end segma;

architecture segm_beh of segma is

procedure flt(a :in std_logic_vector(0 to 3); float: out real) is

--This function converts binary (fraction) to real
variable tem : real;
begin
    tem := 0.0;
for i in 0 to 3 loop
    if (a(i) = '1') then
        tem := tem + 1.0 / (1.0 * 2**(i+1));
    end if;
end loop;
float := tem;
end flt;

Procedure rti (r : in real; int : out integer) is
    --This procedure converts real to integer
    variable temp : real;
    variable intg : integer := 0;
begin
    temp := r;

    while temp >= 0.5 loop
        intg := intg + 1;
        temp := r - 1.0 * intg;
    end loop;
    int := intg;
end rti;

procedure itb (bin : out std_logic_vector;
    N : in integer; int : in integer) is
    variable temp_int : integer := int;
begin
    for i in 0 to N loop
        if (temp_int MOD 2 = 1) then
            bin(i) := '1';
        else bin(i) := '0';
        end if;
        temp_int := temp_int/2;
    end loop;
end itb;

function exp (a : in std_logic_vector (0 to 3))
    return std_logic_vector is
    variable z1 : real;
    variable intgr : integer;
variable tem : std_logic_vector (15 downto 0);

begin

flt (a, z1);
z1 := 1.0 - z1 + z1 * z1 - z1 * z1 * z1;
z1 := z1 * 2**16;
rti (z1, intgr);
itb (tem, 15, intgr);
return tem;
end exp;

begin

g1 : process (x)
variable tem : std_logic_vector (15 downto 0);
begin
tem := exp(x);
y <= tem;
end process g1;
end segm_beh;

Verilog Floating Sum Description
module segma (x, y);
input [0:3] x;
// x is a fraction in Q4 format, 0 < x < 1.
output [15:0] y;
reg [15:0] y;
always @(x)
begin
    y = exp (x);
end

function real float;
//This function is to calculate the real value
//of a fraction in binary
input [0:3] a;

integer i;
begin

```vhdl
float = 0.0;

for (i = 0; i <= 3; i = i + 1)
begin
  if (a[i] == 1)
    float = float + 1.0 / 2**(i+1);
end
endfunction

function [15:0] rti;
input real r;
begin
  rti = r;
end
endfunction

function [15:0] exp;
input [0:3] a;
real z1;
begin
  z1 = float (a); //A call to function "float"
  z1 = 1.0 - z1 + z1**2 - z1**3;
  z1 = z1 * 2**16;
  exp = rti(z1);
end
endfunction
endmodule
```
Advanced HDL Descriptions

EXAMPLES OF FILE PROCESSING
Reading a file consisting of integer numbers

\[
\begin{array}{ccc}
12 & -3 & 5 \\
20 & & \\
\end{array}
\]

FIGURE: 8.1 File file_int.txt

VHDL Code for Reading and Processing a Text File Containing Integers

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

entity FREAD_INTG is
port (START : in std_logic;
z, z1, z2, z3 : out integer);
end FREAD_INTG;

architecture FILE_BEHAVIOR of FREAD_INTG is
begin

process (START)
begin

-- declare the infile as a text file
file infile : text;

--declare the variable fstatus (or any other variable name)
-- as of type file_open_status
variable fstatus : file_open_status;
variable count : integer;

--declare variable temp as of type line
variable temp : line;

begin

--open the file file_int.txt in read mode
file_open (fstatus, infile, "file_int.txt", read_mode);

\]
--Read the first line of the file and store the line in temp
readline (infile, temp);
-- temp now has the data: 12 -3 5

-- Read the first integer (12) from the line temp and store it
-- in the integer variable count.
  read (temp, count);

--count has the value of 12. Multiply by 2 and store in z
  z <= 2 * count;

-- Read the second integer from the line temp and
-- store it in count
  read (temp, count);
--count now has the value of -3

--Multiply by 5 and store in z1
  z1 <= 5 * count;

-- read the third integer in line temp and store it in count
  read (temp, count);

--Multiply by 3 and store in z2
  z2 <= 3 * count;

--Read the second line and store it in temp
readline (infile, temp);
--temp has only the second line

--Read the first integer of the second line and store it in count
read (temp, count);

--Multiply by 4 and store in z3
  z3 <= 4 * count;

--Close the infile
file_close (infile);
end process;

end FILE_BEHAVIOR;
Reading a File Consisting of Real Numbers

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-13.4</td>
<td>-5.654</td>
<td>.023</td>
</tr>
<tr>
<td>-55.32</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE: 8.2** File file_real.txt

VHDL Code for Reading a Text File Containing Real Numbers

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

entity FREAD_REAL is
port (START : in std_logic;
z, z1, z2, z3 : out real);
end FREAD_REAL;

architecture FILE_BEHAVIOR of FREAD_REAL is
begin
  process (START)
  file infile : text;
  variable fstatus : file_open_status;
  variable count : real;
  variable temp : line;
  begin
    --Open the file
    file_open (fstatus, infile, "file_real.txt", read_mode);
    -- Read a line
    readline (infile, temp);
    --Read one number and store it in real variable count
    read (temp, count);
    --multiply by 2
  end process;
end FILE_BEHAVIOR;
```


```
z <= 2.0 * count;

--read another number
read (temp, count);

--multiply by 5
z1 <= 5.0 * count;

--read another number
read (temp, count);

--multiply by 3
z2 <= 3.0 * count;

--read another line
readline (infile, temp);
read (temp, count);

--multiply by 4
z3 <= 4.0 * count;
file_close (infile);
end process;
```

end FILE_BEHAVIOR;

**Reading a File Consisting of ASCII Characters**

```

FIGURE: 8.3 File file_chr.txt.

VHDL Code for Reading an ASCII File

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

entity FREAD_character is
port (START : in std_logic;
    z, z1, z2, z3 : out character);
end FREAD_character;

architecture FILE_BEHAVIOR of FREAD_character is
begin

process (START)
```

A5B
M

```
```
file infile : text;
variable fstatus : file_open_status;
variable count : character;
-- Variable count has to be of type character
variable temp : line;

begin
  file_open (fstatus, infile, "file_chr.txt", read_mode);

--read a line from the file
  readline (infile, temp);

--read a character from the line into count. Count has to be of
--type character
  read (temp, count);

--store the character in z
  z <= count;
  read (temp, count);
  z1 <= count;
  read (temp, count);
  z2 <= count;
  readline (infile, temp);
  read (temp, count);
  z3 <= count;
  file_close (infile);
end process;

end FILE_BEHAVIOR;

VHDL Code for Writing Integers to a File

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;

entity FWRITE_INT is
  port (START : in std_logic;
        z, z1, z2, z3 : in integer);
end FWRITE_INT;

architecture FILE_BEHAVIOR of FWRITE_INT is
begin

  process (START)
  file outfile : text;
  variable fstatus : file_open_status;

  begin

  end process;

end FILE_BEHAVIOR;
begin
  file_open (fstatus, outfile, "Wfile_int.txt", write_mode);
  --The generated file "Wfile_int.txt" is in
  --the same directory as this VHDL module

  --Insert the title of the file Wfile_int.txt.
  --Your simulator should support formatted text;
  --if not, remove all formatted statements " ".
  write (temp, "This is an integer file");

  --Write the line temp into the file
  writeln (outfile, temp);

  --store the first integer in line temp
  write (temp, z);

  -- leave space between the integer numbers.
  write (temp, " ");
  write (temp, z1);

  -- leave another space between the integer numbers.
  write (temp, " ");
  write (temp, z2);
  write (temp, " ");

  writeln (outfile, temp);
  --Insert the fourth integer value on a new line
  write (temp, z3);
  writeln (outfile, temp);

  file_close(outfile);
end process;
end FILE_BEHAVIOR;

OUTPUT

This is an integer file
12  23  -56
-45
VHDL Code for Writing a Package Containing a String of Five Characters

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package array_pkg is
constant N : integer := 4;
--N+1 is the number of elements in the array.
subtype wordChr is character;
type string_chr is array (N downto 0) of wordChr;
end array_pkg;
```

VHDL Code for Reading a String of Characters into an Array

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

--include the package with this module
use work.array_pkg.all;

entity FILE_CHARCTR is
port (START : in std_logic; z : out string_chr);
--string_char is included in the package array_pkg;
--Z is a 5-character array
end FILE_CHARCTR;

architecture FILE_BEHAVIOR of FILE_CHARCTR is
begin
process (START)
file infile : text;
variable fstatus : file_open_status;
variable count : string_chr;
variable temp : line;

begin
file_open (fstatus, infile, "myfile1.txt", read_mode);
readline (infile, temp);
```
read (temp, count);
--Variable count has been declared as an array of five elements,
--each element is a single character
z <= count;

file_close (infile);
end process;

end FILE_BEHAVIOR;

VHDL Code for Finding the Smallest ASCII Value

--The following package needs to be attached to the main module
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package array_pkg is
constant N : integer := 4;
--N+1 is the number of elements in the array.
subtype wordChr is character;
type string_chr is array (N downto 0) of wordChr;
end array_pkg;

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.array_pkg.all;
--Now start writing the code to find the smallest
entity SMALLEST_CHRCTR is
  port (START : in std_logic; z : out string_chr);
end SMALLEST_CHRCTR;

architecture BEHAVIOR_SMALLEST of SMALLEST_CHRCTR is
begin

process (START)
file infile : text;
variable fstatus : file_open_status;
variable count, smallest :
  string_chr := ('z ', 'z ', 'z ', 'z ', 'z ');

-- The above statement assigns initial values (Z’s) to
-- count and smallest.

variable temp : line;

begin
  file_open (fstatus, infile, "f_smallest.txt", read_mode);

  while (count /= ('E', 'N', 'D', ' ', ' ')) loop
    readline (infile, temp);
    read (temp, count);
    if (count < smallest) then
      smallest := count;
    end if;
  end loop;
  z <= smallest;
  file_close (infile);
end process;

end BEHAVIOR_SMALLEST;

After execution, the output Z is equal to “ADA”.

Identifying a Mnemonic Code and its integer Equivalent from a File

<table>
<thead>
<tr>
<th>Mnemonic code</th>
<th>User-assigned integer code</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT</td>
<td>0</td>
</tr>
<tr>
<td>ADD</td>
<td>1</td>
</tr>
<tr>
<td>XOR</td>
<td>4</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
</tr>
<tr>
<td>DIVID</td>
<td>3</td>
</tr>
<tr>
<td>NAND</td>
<td>6</td>
</tr>
<tr>
<td>PRITY</td>
<td>5</td>
</tr>
<tr>
<td>CLA</td>
<td>7</td>
</tr>
</tbody>
</table>

FIGURE 8.6 File cods.txt

VHDL Code for Finding the Integer Code for a Mnemonic Code

--The following package needs to be attached to the main module
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package array_pkg is
constant N : integer := 4;
--N+1 is the number of elements in the array.
subtype wordChr is character;
type string_chr is array (N downto 0) of wordChr;

end array_pkg;

--Start writing the code to find the assigned integer value
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.array_pkg.all;

entity OPCODES is
  port (assmbly_code : in string_chr; z : out string_chr;
        z1 : out integer);

end OPCODES;

architecture BEHAVIOR of OPCODES is
begin

process (assmbly_code)
begin

  file infile : text;
  variable fstatus : file_open_status;
  variable temp : string_chr := (' ', ' ', ' ', ' ', ' ');
  variable tem_bin : integer;
  variable regstr : line;

  begin
    file_open (fstatus, infile, "cods.txt", read_mode);

    for i in 0 to 8 loop

      -- while loop could have been used instead of for loop. See
      -- Exercise 8.3
      readline (infile, regstr);
      read (regstr, temp);

      for j in 0 to N loop

        tem_bin := tem_bin xor temp(j); 

      end loop;

      z1 := tem_bin;

    end loop;

  end begin;

end process;

end BEHAVIOR;
if (temp = asmbly_code) then
z <= temp;

read (regstr, tem_bin);
z1 <= tem_bin;

exit;

else if (i > 7) then
report ("ERROR: CODE COULD NOT BE FOUND");
z <= ('E', 'R', 'R', 'O', 'R', 'E', 'R', 'O');

-- assign -1 to z1 if an error occurs
z1 <= -1;
end if;
end loop;

file_close(infile);

end process;

end BEHAVIOR;

VHDL Code of an Assembler

FIGURE 8.7 The input and output of an assembler.
ORG  200  
CLA   0  
ADD   9  
XOR  10  
MULT  11  
DIVID 12  
XOR  13  
NAND 14  
PRITY 0  
HALT 0  
HLT  5  
END  

**FIGURE 8.8** File asm.txt

**FIGURE 8.9** Flowchart of the assembler.
VHDL Assembler Code

--The following package needs to be attached to the main module
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package array_pkg is
constant N : integer := 4;
--N+1 is the number of elements in the array.
subtype wordChr is character;
type string_chr is array (N downto 0) of wordChr;

end array_pkg;

library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.array_pkg.all;

--Now start the code for the assembly
entity ASSMBLR is

port (START : in bit);

end ASSMBLR;

architecture BEHAVIOR_ASSM of ASSMBLR is

begin

process (START)
file infile : text;
file outfile : text;

variable fstatus, fstatus1 : file_open_status;
variable temp : string_chr := (' ', ' ', ' ', ' ', ' '); 
variable code, addr : integer;
variable regstr, regstw : line;
variable ctr : integer := -1;

begin

file_open (fstatus, infile, "asm.txt", read_mode);

file_open (fstatus1, outfile, "outf.txt", write_mode);

-- Prepare the outfile where the results of the assembler


-- are stored.

write (regstw, "Location Code Address");
write (outfile, regstw);

for i in 0 to 11 loop
--while-loop could have been used instead of for-loop.

readline (infile, regstr);
read (regstr, temp);

if (temp = ('O', 'R', 'I', 'G', ' ')) then
read (regstr, ctr);
elsif (ctr = -1) then
-- If the code of the first line in the file is not ORIG
-- report an error
write (regstw, " ERROR: FIRST OPCODE SHOULD BE ORIG");
write (outfile, regstw);
exit;
else
read (regstr, addr);
write (regstw, ctr);
write (regstw, "");
ctr := ctr + 1;
case temp is
when ('H', 'A', 'L', 'T', ' ') =>
code := 0;
write (regstw, code);
write (regstw, "");
write (regstw, addr);
write (outfile, regstw);

when ('A', 'D', 'D', ' ', ' ') =>
code := 1;

write (regstw, code);
write (regstw, " ");
write (regstw, addr);
writeline (outfile, regstw);

when ('M', 'U', 'L', 'T', ' ') =>
code := 2;
write (regstw, code);
write (regstw, " ");
write (regstw, addr);
writeline (outfile, regstw);

when ('D', 'I', 'V', 'I', 'D') =>
code := 3;
write (regstw, code);
write (regstw, " ");
write (regstw, addr);
writeline (outfile, regstw);

when ('X', 'O', 'R', ' ', ' ') =>
code := 4;
write (regstw, code);
write (regstw, " ");
write (regstw, addr);
writeline (outfile, regstw);

when ('P', 'R', 'I', 'T', 'Y') =>
code := 5;
write (regstw, code);
write (regstw, " ");
write (regstw, addr);
writeline (outfile, regstw);

when ('N', 'A', 'N', 'D', ' ') =>
code := 6;
write (regstw, code);
write (regstw, " ");
write (regstw, addr);
writeline (outfile, regstw);
when ('C', 'L', 'A', ' ', ' ') =>
    code := 7;
    write (regstw, code);
    write (regstw, " ");
    write (regstw, addr);
    writeln (outfile, regstw);
when ('E', 'N', 'D', ' ', ' ') =>
    write (regstw, "END OF FILE ");
    writeln (outfile, regstw);
    exit;
when others =>
    code := -20;
    write (regstw, "ERROR ");
    write (regstw, code);
    writeln (outfile, regstw);
end case;
end if;
end loop;
file_close(infile);
file_close (outfile);
end process;
end BEHAVIOR_ASSM;

Manipulating and Displaying Data in a Verilog File

Verilog Code for Storing $b = 2a$ in file4.txt

module file_test (a, b);
input [1:0] a;
output [2:0] b;
reg [2:0] b;
integer ch1;
always @ (a)
begin
b = 2 * a;
end

initial
begin
  ch1 = $fopen("file4.txt");

  $fdisplay (ch1, "\n\t\t\t This is file4.txt \n");
  $fdisplay (ch1, " Input a in Decimal\tOutput b in
            Decima\t\tOutput b in Binary\n ");
  /*The above statement when entered in the Verilog module should
    be entered in one line without carriage return */
  $fmonitor (ch1,"%d\t%d\t%b \n", a, b);
end
endmodule

**VHDL RECORD TYPE**

**VHDL Code for an Example of Record**

The following is the code of the package weather_fcst

```vhdl
package weather_fcst is
  Type cast is (rain, sunny, snow, cloudy);
  Type weekdays is (Monday, Tuesday, Wednesday,
                 Thursday, Friday, Saturday, Sunday);
  Type forecast is
    Record
      Tempr : real range -100.0 to 100.0;
      unit : string (1 to 3);
      Day : weekdays;
      Cond : cast;
    end record;
end package weather_fcst;
```

-- Now we write the program
library ieee;
use ieee.std_logic_1164.all;
use std.textio.all;
use work.weather_fcst.all;
entity WEATHER_FRCST is
  port (Day_in : in weekdays; unit_in : in string (1 to 3);
       out_temperature : out real;
       out_unit : out string (1 to 3);
       out_day : out weekdays; out_cond : out cast);
-- Type string is a predefined
end WEATHER_FRCST;

--Now we write the code

architecture behaviour_record of WEATHER_FRCST is
begin
process (Day_in, unit_in)
variable temp : forecast;
begin

  case Day_in is

  when Monday =>
      temp.cond := sunny;
      if (unit_in = "CEN") then
          temp.tempr := 35.6;
      elsif (unit_in = "FEH") then
          temp.tempr := 1.2 * 35.6 + 32.0;
      else
          report ("invalid units");
      end if;

  when Tuesday =>
      temp.cond := rain;
      if (unit_in = "CEN") then
          temp.tempr := 30.2;
      elsif (unit_in = "FEH") then
          temp.tempr := 1.2 * 30.2 + 32.0;
      else
          report ("invalid units");
      end if;

  when Wednesday =>
      temp.cond := sunny;
      if (unit_in = "CEN") then
          temp.tempr := 37.2;
elsif (unit_in = "FEH") then
temp.tempr := 1.2 * 37.2 + 32.0;
else
report ("invalid units");
end if;

when Thursday =>
temp.cond := cloudy;
if (unit_in = "CEN") then
temp.tempr := 30.2;
elsif (unit_in = "FEH") then
temp.tempr := 1.2 * 30.2 + 32.0;
else
report ("invalid units");
end if;

when Friday =>
temp.cond := cloudy;
if (unit_in = "FEH") then
temp.tempr := 33.9;
elsif (unit_in = "FEH") then
temp.tempr := 1.2 * 33.9 + 32.0;
else
report ("invalid units");
end if;

when Saturday =>
temp.cond := rain;
if (unit_in = "CEN") then
temp.tempr := 25.1;
elsif (unit_in = "FEH") then
temp.tempr := 1.2 * 25.1 + 32.0;
else
report ("invalid units");
end if;

when Sunday =>
temp.cond := rain;
if (unit_in = "FEH") then
temp.tempr := 27.1;
elsif (unit_in = "FEH") then
temp.tempr := 1.2 * 27.1 + 32.0;
else
report ("invalid units");
end if;
when others =>
temp.tempr := 99.99;
report ("ERROR-NOT VALID DAY");
end case;

out_temperature <= temp.tempr;
out_unit <= unit_in;
out_day <= Day_in;
out_cond <= temp.cond;
end process;
end behaviour_record;

**Figure 8.10** Simulation output
ASSIGNMENT QUESTIONS

1) What are the difference between procedure and function?
2) Explain function syntax with example
3) Explain procedure syntax with example
4) Write a HDL Description of a Full Adder Using Procedure and Task
5) Write a HDL Description of an N-Bit Ripple Carry Adder Using Procedure and Task
6) Write a HDL Code for Converting an Unsigned Binary to an Integer Using Procedure and Task
7) Write a HDL Code for Converting a Fraction Binary to Real Using Procedure and Task
8) Write a HDL Code for Converting an Unsigned Integer to Binary Using Procedure and Task
9) Write a HDL Code for Converting a Signed Binary to Integer Using Procedure and Task
10) Write a VHDL Code for Converting an Integer to Signed Binary Using Procedure

11) Write a HDL Code for Signed Vector Multiplication Using Procedure and Task

12) Write a HDL Description for Enzyme Activity Using Procedure and Task

13) Write a Verilog Function That Calculates \( \text{exp} = \text{a XOR b} \)
14) Write a HDL Function to Find the Greater of Two Signed Numbers
15) Write a VHDL Code for Reading and Processing a Text File Containing Integers

16) Write a VHDL Code for Reading a Text File Containing Real Numbers
17) Write a VHDL Code for Writing Integers to a File
18) Write a VHDL Code for Reading a String of Characters into an Array
19) Write a Verilog Code for Storing \( b = 2a \) in file4.txt
UNIT 6: MIXED TYPE DESCRIPTIONS

Syllabus of unit 6: Hours :6
Why Mixed-Type Description? VHDL User-Defined Types, VHDL Packages, Mixed-Type Description examples

Recommended readings:

2. **Verilog HDL** –Samir Palnitkar-Pearson Education
4. **A Verilog HDL Primer**- J.Bhaskar – BS Publications
5. **Circuit Design with VHDL**-Volnei A.Pedroni-PHI
UNIT 6: MIXED TYPE DESCRIPTIONS

Why mixed – type description?
Mixed- type description is that it is an HDL code that mixes different types of descriptions within the same module. Consider a system that performs two operations addition (Z = X+Y) and division (Z = X/Y). This can be implemented by using behavioral statements. If the behavioral approach is used, we have no control over selecting the components or the methods used to implement the addition and division. The HDL package may contain addition or division algorithms not suitable for our needs. For example, the addition algorithm might need to be as fast as possible, to achieve this we have to use fast adders such as carry-look ahead or carry-save adders. There is no guarantee that behavioral description implements these adders in its addition function. Most likely, it implements ripple – carry addition. If we use data-flow or structural descriptions can also be implemented to describe the specific adder. It is however hard to implement descriptions of complex algorithms, such as division; the logic diagram of divisors is generally complex.

The third option is to use a mixture of two types of descriptions: structural or data-flow for addition and behavioral for division.

VHDL USER – DEFINED TYPES
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;
begi
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;
VHDL PACKAGES
· Packages are useful in organizing the data and the subprograms declared in the model
· VHDL also has predefined packages. These predefined packages include all of the predefined types and operators available in VHDL.
· In VHDL a package is simply a way of grouping a collections of related declarations that serve a common purpose. This can be a set of subprograms that provide operations on a particular type of data, or it can be a set of declarations that are required to modify the design
· Packages separate the external view of the items they declare from the implementation of the items. The external view is specified in the package declaration and the implementation is defined in the separate package body.
· Packages are design unit similar to entity declarations and architecture bodies. They can be put in library and made accessible to other units through use and library clauses
· Access to members declared in the package is through using its selected name
Library_name.package_name.item_name
· Aliases can be used to allow shorter names for accessing declared items
Two Components to Packages
· Package declaration---The visible part available to other modules
· Package body --_The hidden part
Package declaration
The Packages declaration is used to specify the external view of the items. The syntax rule for the package declaration is as follows.
· The identifier provides the name of the package. This name can be used anywhere in the model to identify the model
· The package declarations includes a collection of declaration such as
  _ Type
  _ Subtypes
  _ Constants
  _ Signal
  _ Subprogram declarations etc
  _ Aliases
  _ components
The above declarations are available for the user of the packages.
The following are the advantages of the usage of packages.
· All the declarations are available to all models that use a package.
· Many models can share these declarations. Thus, avoiding the need to rewrite these declarations for every model.
The following are the points to be remembered on the package declaration
· A package is a separate form of design unit, along with entity and architecture bodies.
· It is separately analyzed and placed in their working library.
· Any model can access the items declared in the package by referring the name of the declared item.

Package declaration syntax

```
package identifier is
    { package_declarative_item }
end [ package ] [ identifier ] ;
```

Constants in package declaration
The external view of a constant declared in the package declaration is just the name of the constant and the type of the constant. The value of the constant need not be declared in the package declaration. Such constants are called are called deferred constants. The actual value of these constants will be specified in the package body. If the package declaration contains deferred constants, then a package body is a must. If the value of the constant is specified in the declaration, then the package body is not required.
The constants can be used in the case statement, and then the value of the constant must be logically static. If we have deferred constants in the package declaration then the value of the constant would not be known when the case statement is analyzed. Therefore, it results in an error. In general, the value of the deferred constants is not
logically static.

**Subprograms in the package declaration**
- Procedures and functions can be declared in the package declaration.
- Only the design model that uses the package can access the subprograms declared in the package declarations.
- The subprogram declaration includes only the information contained in the header.
- This does not specify the body of the subprogram. The package declaration provides information regarding the external view of the subprogram without the implementation details. This is called information hiding.
- For every subprogram declaration there must be subprogram body in the package body. The subprograms present in the package body but not declared in the package declaration can’t be accessed by the design models.

**Package body**
Each package declaration that includes a subprogram or a deferred constant must have package body to fill the missing information. But the package body is not required when the package declaration contains only type, subtype, signal or fully specified constants. It may contain additional declarations which are local to the package body but cannot declare signals in body. Only one package body per package declaration is allowed.

```vhdl
package body identifier is
   { package_body_declarative_item }
end [ package body ] identifier ;
```

**Point to remember:**
- The package body starts with the key word package body.
- The identifier of the package body follows the keyword.
- The items declared in the package body must include full declarations of all subprograms declared in the corresponding package declarations. These full declarations must include subprogram headers as it appears in the package declarations. This means that the names, modes typed and the default values of each parameters must be repeated in exactly the same manner. In this regard two variations are allowed:
  - A numerical literal may be written differently for example; in a different base provided it has the same value.
  - A simple name consisting just of an identifier can be replaced by a selected name, provided it refers to the same item.
- A deferred constant declared in the package declaration must have its value specified in the package body by declaration in the package body.
- A package body may include additional types, subtypes, constants and subprograms. These items are included to implement the subprogram defined in the package declaration. The items declared in the package declaration...
can’t be declared in the package body again
· An item declared in the package body has its scope restricted to within the package body, and these items are not visible to other design units.
· Every package declaration can have at most one package body with the name same as that of the package declaration
· The package body can’t include declaration of additional signals. Signals declarations may only be included in the interface declaration of package.

**Examples for package**
Creating a package ‘bit_pack’ to add four bit and eight bit numbers.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
package bit_pack is
function add4(add1, add2: std_logic_vector (3 downto 0); carry: std_logic ) return std_logic_vector;
end package bit_pack;
package body bit_pack is
function add4(add1, add2: std_logic_vector (3 downto 0); carry: std_logic ) return std_logic_vector is
variable cout, cin: std_logic;
variable ret_val : std_logic_vector (4 downto 0);
begin
  cin:= carry;
  ret_val:="00000" ;
  for i in 0 to 3 loop
    ret_val(i) := add1(i) xor add2(i) xor cin;
    cout:= (add1(i) and add2(i)) or (add1(i) and cin) or (add2(i) and cin);
    cin:= cout; end
  loop;
  ret_val(4):=cout;
  return ret_val;
end add4;

1. Program to add two four bit vectors using functions written in package ‘bit_pack’.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.bit_pack.all;
entity addfour is
port ( a: in STD_LOGIC_VECTOR (3 downto 0);
b: in STD_LOGIC_VECTOR (3 downto 0);
cin: in STD_LOGIC;
sum: out STD_LOGIC_VECTOR (4 downto 0) );
end addfour;
architecture addfour of addfour is
begin
  sum<= add4(a, b, cin);
end addfour;
```
end addfour;

**HDL Code for Finding the Greatest Element of an Array—VHDL and Verilog**

**VHDL: Finding the Greatest Element of an Array**

library IEEE;
use IEEE.STD_LOGIC_1164.all;

--Build a package for an array
package array_pkg is
constant N : integer := 4;
--N+1 is the number of elements in the array.

constant M : integer := 3;
--M+1 is the number of bits of each element
--of the array.
subtype wordN is std_logic_vector (M downto 0);
type strng is array (N downto 0) of wordN;
end array_pkg;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.array_pkg.all;
-- The above statement makes the package array_pkg visible in
--this module.
entity array1 is
    generic (N : integer := 4; M : integer := 3);
    --N + 1 is the number of elements in the array; M = 1 is the
    --number of bits of each element.
    Port (a : inout strng; z : out std_logic_vector (M downto 0));
end array1;
architecture max of array1 is
   begin

   com: process (a)
   variable grtst : wordN;
   begin

      --enter the data of the array.
      a <= ("0110", "0111", "0010", "0011", "0001");

      grtst := "0000";

      lop1 : for i in 0 to N loop

         if (grtst <= a(i)) then
            grtst := a(i);
            report " grtst is less or equal than a";
            -- use the above report statement if you want to monitor the
            -- progress of the program
         else
            report "grtst is greater than a";
            -- Use the above report statement to monitor the
            -- progress of the program
         end if;
      end loop lop1;

      z <= grtst;
   end process com;

   end max;
Verilog: Finding the Greatest Element of an Array

module array1 (start, grtst);
parameter N = 4;
parameter M = 3;
input start;
output [3:0] grtst;
reg [M:0] a[0:N];

/*The above statement is declaring an array of N + 1 elements;
each element is M bits. */

reg [3:0] grtst;
integer i;
always @ (start)
begin
  a[0] = 4'b0110;
a[1] = 4'b0111;
a[2] = 4'b0010;
a[3] = 4'b0011;
a[4] = 4'b0001;
grtst = 4'b0000;

  for (i = 0; i <= N; i= i +1)
  begin

    if (grtst <= a[i])
        begin
            grtst = a[i];
            $display (" grtst is less or equal than a");
            // use the above statement to monitor the program
        end

  end
endmodule
else
    $display(" grst is greater than a");

// use the above statement to monitor the program

end
end
endmodule

Multiplication of Two Signed N-Element Vectors—VHDL and Verilog

VHDL: Multiplication of Two Signed N-Element Vectors
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

package booth_pkg is
    constant N : integer := 4;
    --N + 1 is the number of elements in the array.

    constant M: integer := 3;
    --M + 1 is the number of bits of each element
    --of the array.

    subtype wordN is signed (M downto 0);
    type strng is array (N downto 0) of wordN;

    procedure booth (X, Y : in signed (3 downto 0);
        Z : out signed (7 downto 0));

    end booth_pkg;
package body booth_pkg is

procedure booth (X, Y : in signed (3 downto 0); Z : out signed (7 downto 0)) is
-- Booth algorithm here is restricted to 4x4 bits.
-- It can be adjusted to multiply any NxN bits.
variable temp : signed (1 downto 0);
variable sum : signed (7 downto 0);
variable E1 : unsigned (0 downto 0);
variable Y1 : signed (3 downto 0);
begin

sum := "00000000"; E1 := "0";
for i in 0 to 3 loop
  temp := X(i) & E1(0);
  Y1 := -Y;
  case temp is
    when "10" => sum (7 downto 4) :=
      sum (7 downto 4) + Y1;
    when "01" => sum (7 downto 4) :=
      sum (7 downto 4) + Y;
    when others => null;
  end case;
  sum := sum srl 1;
  sum(7) := sum(6);
  E1(0) := x(i);
end loop;
if (y = "1000") then
  sum := -sum;
end if;

-- If Y = 1000; then Y1 is calculated as 1000;
-- that is -8, not 8 as expected. This is because Y1 is
--4 bits only. The statement sum = -sum corrects
--this error.

end if;
Z := sum;
end booth;

end booth_pkg;

-- We start writing the multiplication algorithm using
-- the package booth_pkg
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use work.booth_pkg.all;

entity vecor_multply is
generic (N : integer := 4; M : integer := 3);
--N + 1 is the number of elements in the array; M + 1 is the
--number of bits of each element.
Port (a, b : in string; d : out signed (3*N downto 0));
end vecor_multply;

architecture multiply of vecor_multply is
begin
process (a, b)
variable temp : signed (7 downto 0);
variable temp5 : signed (3*N downto 0) := "0000000000000000";

begin
for i in 0 to 4 loop  
booth(a(i), b(i), temp);

--accumulate the partial products in the product temp5  
temp5 := temp5 + temp;  
end loop;  
d <= temp5;  
end process;  

end multiply;

Verilog: Multiplication of Two Signed N-Element Vectors  
module vecor_multiply (start, d);  
parameter N = 4;  
parameter M = 3;  
input start;  
output signed [3*N:0] d;  
reg signed [M:0] a[0:N];  
reg signed [M:0] b[0:N];  
reg signed [3*N:0] d;  
reg signed [3*N:0] temp;  
integer i;  

always @ (start)  
begin  
a[0] = 4'b1100;  
a[1] = 4'b0000;  
a[2] = 4'b1001;  
a[3] = 4'b0011;  
a[4] = 4'b1111;  
b[0] = 4'b1010;  
b[1] = 4'b0011;  
b[2] = 4'b0111;
b[3] = 4'b1000;
b[4] = 4'b1000;
d = 0;
for (i = 0; i <= N; i = i + 1)
begin
    booth (a[i], b[i], temp);
    d = d + temp;
end
end

task booth;
input signed [3:0] X, Y;
output signed [7:0] Z;
reg signed [7:0] Z;
reg [1:0] temp;
integer i;
reg E1;
reg [3:0] Y1;
begin
    Z = 8'd0;
    E1 = 1'd0;
for (i = 0; i < 4; i = i + 1)
begin
    temp = {X[i], E1}; //This is catenation
    Y1 = -Y; //Y1 is the 2'complement of Y
    case (temp)
    default : begin end
end
endcase
Z = Z >> 1;
/*The above statement is a logical shift of
one position to the right*/

Z[7] = Z[6];
/*The above two statements perform arithmetic shift where
the sign of the number is preserved after the shift. */

E1 = X[i];

end

if (Y == 4'b1000)
/* If Y = 1000, then Y1 = 1000 (should be 8 not -8).
This error is because Y1 is 4 bits only.
The statement Z = -Z adjusts the value of Z. */

Z = -Z;
end

endtask
endmodule

VHDL Two-Dimensional Array

library IEEE;
use IEEE.STD_LOGIC_1164.all;

--Build a package to declare the array
package twodm_array is

constant N : integer := 4;
-- N+1 is the number of elements in the array.
-- this is \([N+1,N+1]\) matrix with N+1 rows and N+1 columns

subtype wordg is integer;
type strng1 is array (N downto 0) of wordg;
type strng2 is array (N downto 0) of strng1;

end twodm_array;

-- use the package to describe a two-dimensional array
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.twodm_array.all;
-- The above statement instantiates the package twodm_array

entity two_array is
  Port (N, M : integer; z : out integer);
end two_array;

architecture Behavioral of two_array is

begin
  com : process (N, M)
  variable t : integer;
  constant y : strng2 := ((7, 6, 5, 4, 3), (6, 7, 8, 9, 10),
    (30, 31, 32, 33, 34), (40, 41, 42, 43, 44),
    (50, 51, 52, 53, 54));

  begin

    t := y (N)(M);
    -- Look at the simulation output to identify
-- the elements of the array
z <= t;
end process com;
end Behavioral;

VHDL Description: Addition of Two [5×5] Matrices

-- First, write a package to declare a two-dimensional
-- array with five elements
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package twodm_array is

constant N : integer := 4;
-- N+1 is the number of elements in the array.
-- This is an N×N matrix with N rows and N columns.
subtype wordg is integer;
type strng1 is array (N downto 0) of wordg;
type strng2 is array (N downto 0) of strng1;

end twodm_array;

--Second, write the code for addition
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.twodm_array.all;
entity matrices is
  Port (x, y : strng2; z : out strng2);
--strng2 type is 5×5 matrix
end matrices;

architecture sum of matrices is
begin
com : process (x, y)
variable t : integer := 0;
begin

for i in 0 to 4 loop
for j in 0 to 4 loop
t := x(i)(j) + y(i)(j);
z(i)(j) <= t;
end loop;
end loop;

end process com;
end sum;

MIXED – TYPE DESCRIPTION EXAMPLES
HDL Description of an ALU—VHDL and Verilog
**FIGURE 7.3** Block diagram of the arithmetic – logic unit.

**VHDL ALU Description**

--Here we write the code for a package for user-defined type and function.

library ieee;
use ieee.std_logic_1164.all;
use IEEE.STD_LOGIC_1164.ALL,IEEE.NUMERIC_STD.ALL;
package codes_Arithm is
type op is (add, mul, divide, none);
-- type op is for the operation codes for the ALU. The operations we need are: addition, multiplication, division, and no operation

function TO_UNSIGN (b : integer) return unsigned;
end;

package body codes_Arithm is
function TO_UNSIGN (b : integer) return unsigned is

--The function converts integer numbers to unsigned. This function can be omitted if it is included in a vendor’s package. The vendor’s -- package, if available, should be attached.

variable temp : integer;
variable bin : unsigned (5 downto 0);
begintemp := b;
for j in 0 to 5 loop

if (temp MOD 2 = 1) then
bin (j) := '1';
else bin (j) := '0';
end if;
temp := temp/2;
end loop;
return bin;
end TO_UNSIGN;
end codes_Arithm;

--Now we write the code for the ALU
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use work.codes_arithm.all;

--The above use statement is to make the user-
--defined package "codes_arithm.all" visible to this module.

entity ALU_mixed is

port (a, b : in unsigned (2 downto 0); cin : in std_logic;
opc : in op; z : out unsigned (5 downto 0));

--opc is of type "op": type op is defined in the
--user-defined package "codes_arithm"
end ALU_mixed;

architecture ALU_mixed of ALU_mixed is

signal c0, c1 : std_logic;
signal p, g : unsigned (2 downto 0);
signal temp1 : unsigned (5 downto 0);

begin
--The following is a data flow-description of a 3-bit lookahead
--adder. The sum is stored in the three least significant bits of
--temp1. The carry out is stored in temp1(3).

\[
\begin{align*}
g(0) &\leq a(0) \text{ and } b(0); \\
g(1) &\leq a(1) \text{ and } b(1); \\
g(2) &\leq a(2) \text{ and } b(2); \\
p(0) &\leq a(0) \text{ or } b(0); \\
p(1) &\leq a(1) \text{ or } b(1); \\
p(2) &\leq a(2) \text{ or } b(2); \\
c_0 &\leq g(0) \text{ or } (p(0) \text{ and } c_{in}); \\
c_1 &\leq g(1) \text{ or } (p(1) \text{ and } g(0)) \text{ or } (p(1) \text{ and } p(0) \text{ and } c_{in}); \\
temp1(3) &\leq g(2) \text{ or } (p(2) \text{ and } g(1)) \text{ or } (p(2) \text{ and } p(1) \\
&\text{ and } g(0)) \text{ or } (p(2) \text{ and } p(1) \text{ and } p(0) \text{ and } c_{in}); \\
\end{align*}
\]

--temp1(3) is the final carryout of the adders

\[
\begin{align*}
temp1(0) &\leq (p(0) \text{ xor } g(0)) \text{ xor } c_{in}; \\
temp1(1) &\leq (p(1) \text{ xor } g(1)) \text{ xor } c_0; \\
temp1(2) &\leq (p(2) \text{ xor } g(2)) \text{ xor } c_1; \\
temp1 (5 \text{ downto } 4) &\leq "00"; \\
\end{align*}
\]

process (a, b, cin, opc, temp1)

--The following is a behavioral description for the multiplication
--and division functions of the ALU.

variable temp : unsigned (5 downto 0);
variable a1, a2, a3 : integer;
begin
a1 := TO_INTEGER (a);
a2 := TO_INTEGER (b);
--The predefined function "TO_INTEGER"
--converts unsigned to integer.
--The function is a member of the VHDL package IEEE.numeric.

case opc is
  when mul =>
    a3 := a1 * a2;
    temp := TO_UNSIGNED(a3);
  when divide =>
    a3 := a1 / a2;
    temp := TO_UNSIGNED(a3);
  when add =>
    temp := temp1;
  when none =>
    null;
end case;

z <= temp;
end process;

end ALU_mixed;

**Verilog ALU Description**

module ALU_mixed (a, b, cin, opc, z);
parameter add = 0;
parameter mul = 1;
parameter divide = 2;
parameter nop = 3;
input [2:0] a, b;
inout cin;
input [1:0] opc;
output [5:0] z;
reg [5:0] z;
wire [5:0] temp1;
wire [2:0] g, p;
wire c0, c1;

// The following is data-flow description
// for 3-bit lookahead adder
assign g[0] = a[0] & b[0];
assign g[1] = a[1] & b[1];
assign p[0] = a[0] | b[0];
assign p[1] = a[1] | b[1];
assign c0 = g[0] | (p[0] & cin);
assign c1 = g[1] | (p[1] & g[0]) | (p[1] & p[0] & cin);
// temp1[3] is the final carryout of the adders
assign temp1[0] = (p[0] ^ g[0]) ^ cin;
assign temp1[1] = (p[1] ^ g[1]) ^ c0;
assign temp1[2] = (p[2] ^ g[2]) ^ c1;
assign temp1[5:4] = 2'b00;

// The following is behavioral description
always @ (a, b, cin, opc, temp1)
begin

case (opc)

    mul : z = a * b;
    add : z = temp1;

endcase

divide : z = a / b;
nop : z = z;
endcase
end
endmodule

FIGURE 7.4 Simulation output of the ALU

HDL Description of 16X8 SRAM

FIGURE 7.5 A block diagram of 16X8 static memory.
VHDL 16x8 SRAM Description

library IEEE;
use IEEE.STD_LOGIC_1164.all;

--Build a package for an array
package array_pkg is
constant N : integer := 15;
--N+1 is the number of elements in the array.
constant M : integer := 7;
--M+1 is the number of bits of each element
--of the array.
subtype wordN is std_logic_vector (M downto 0);
type strng is array (N downto 0) of wordN;
end array_pkg;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use work.array_pkg.all;

entity memory16x8 is
generic (N : integer := 15; M : integer := 7);
--N+1 is the number of words in the memory; M+1 is the
--number of bits of each word.
Port (Memory : inout strng; CS : in std_logic;
    ABUS : in unsigned (3 downto 0);
    Data_in : in std_logic_vector (7 downto 0);
    R_WRbar : in std_logic;
Data_out : out std_logic_vector (7 downto 0));
end memory16x8;

architecture SRAM of memory16x8 is

begin

com : process (CS, ABUS, Data_in, R_WRbar)

variable A : integer range 0 to 15;
begin

if (CS = '1') then

A := TO_INTEGER (ABUS);
-- TO_INTEGER is a built-in function

if (R_WRbar = '0') then

Memory (A) <= Data_in;
else

Data_out <= Memory(A);
end if;
else

Data_out <= "ZZZZZZZZ";
--The above statement describes high impedance.
end if;

end process com;

end SRAM;
Verilog 16x8 SRAM Description

module memory16x8 (CS, ABUS, Data_in, R_WRbar, Data_out);
input CS, R_WRbar;
input [3:0] ABUS;
input [7:0] Data_in;
output [7:0] Data_out;
reg [7:0] Data_out;
reg [7:0] Memory [0:15];

always @ (CS, ABUS, Data_in, R_WRbar)
begin

if (CS == 1'b1)
begin

if (R_WRbar == 1'b0)
begin
    Memory [ABUS] = Data_in;
end
else
    Data_out = Memory [ABUS];
end
else
    Data_out = 8'bZZZZZZZZ;
//The above statement describes high impedance

end
endmodule

endmodule
FIGURE 7.6 Simulation output of 16X8 static memory.

FIGURE 7.7 State – diagram of a finite sequential – state machine.

VHDL State Machine Description

library IEEE;
    use IEEE.STD_LOGIC_1164.all;

    --First we write a package that includes type “states.”
    package types is
type op is (add, mul, divide, none);
type states is (state0, state1, state2, state3);
end;

-- Now we use the package to write the code for the state machine.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use work.types.all;
entity state_machine is
  port (A, clk : in std_logic; pres_st : buffer states;
        Z : out std_logic);
end state_machine;

architecture st_behavioral of state_machine is
begin

  FM : process (clk, pres_st, A)
  variable present : states := state0;
  begin
  if (clk = '1' and clk'event) then
    case pres_st is
      when state0 => if A
                      = '1' then present
                      := state1; Z <=
                      '0';
                    else
                      present := state0;
                    Z <= '1';
                  end if;
      when state1 =>
        if A = '1' then
          present := state2;
        end if;
    end case;
  end if;
end process FM;
end;
Z <= '0';
else
  present := state3;
Z <= '0';
end if;

when state2 => if A
  ='1' then present
  := state3; Z <=
  '1';
else
  present := state0;
Z <= '0';
end if;

when state3 => if A
  ='1' then present
  := state0; Z <=
  '0';
else
  present := state2;
Z <= '0';
end if;
end case:
pres_st <= present;
end if;
end process FM;
end st_behavioral;

Verilog State Machine Description
`define state0 2'b00
`define state1 2'b01
`define state2 2'b10
`define state3 2'b11
module state_machine (A, clk, pres_st, Z);

input A, clk;
output [1:0] pres_st;
output Z;
reg Z;

reg [1:0] present;
reg [1:0] pres_st;

initial
begin
    pres_st = 2'b00;
end

always @ (posedge clk)
begin

    case (pres_st)
    `state0 :
        begin
            if (A == 1)
            begin
                present = `state1;
                Z = 1'b0;
            end
            else
            begin
                present = `state0;
                Z = 1'b1;
            end
        end
    end
\`state1 :
  begin
    if (A == 1)
    begin
      present = `state2;
      Z = 1'b0;
    end
    else
    begin
      present = `state3;
      Z = 1'b0;
    end
  end

\`state2 :
  begin
    if (A == 1)
    begin
      present = `state3;
      Z = 1'b1;
    end
    else
    begin
      present = `state0;
      Z = 1'b0;
    end
  end

\`state3 :
  begin
    if (A == 1)
begin
    present = `state0;
    Z = 1'b0;
end
else
begin
    present = `state2;
    Z = 1'b0;
end
endcase
pres_st = present;
end
endmodule

FIGURE 7.8 Simulation waveform of the state machine
HDL Description of a Basic Computer

FIGURE 7.9 Registers in the basic computer.

FIGURE 7.10 Basic computer instruction format.
FIGURE 7.11 Fetch and execute cycles of the basic computer.
FIGURE 7.12 State diagram of the finite sequential – state machine.

VHDL Basic Computer Memory Program

--Write the code for Package Comp_Pkg

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.numeric_std.all;

package Comp_Pkg is
constant N: integer := 15;
--N+1 is the number of elements in the array.
constant M : integer := 7;
--M+1 is the number of bits of each element
--of the array.

subtype wordN is unsigned (M downto 0);
type strng is array (N downto 0) of wordN;
type states is (state0, state1, state2, state3);

end Comp_Pkg;

--Now write the code for the control unit
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;
use work.Comp_Pkg.all;

entity computer_basic is

generic (N : integer := 15; M : integer := 7);
--N+1 is the number of words in the memory; M+1 is the
--number of bits of each word.

Port (Memory : inout string; PC : buffer unsigned (3 downto 0);
      clk_master : std_logic;
      ACH : buffer unsigned (7 downto 0);
      ACL : buffer unsigned (7 downto 0);
      Reset : buffer std_logic; ON_OFF : in std_logic);

end computer_basic;

architecture Behavioral_comp of computer_basic is

signal z : unsigned (0 downto 0);
signal clk : std_logic;

begin

z(0) <= ACL(6) xor ACL(5) xor ACL(4) xor
       ACL(3) xor ACL(2) xor ACL(1) xor ACL(0);
--Z has to be in vector form to match ACL
clk <= clk_master and ON_OFF;

--The above two statements are data-flow description.
--The following is behavioral description.

cpu : process (Reset, PC, ACL, ACH, clk, Memory, z(0))
variable AR : unsigned (3 downto 0);
variable DR : unsigned (7 downto 0);
variable pres_st, next_st : states;
variable ARI : integer range 0 to 16;
variable IR : unsigned (2 downto 0);
variable PR : unsigned (15 downto 0);

begin

if rising_edge (clk) then
if Reset = '1' then
pres_st := state0;
Reset <= '0';
PC <= "0000";
end if;

case pres_st is
when state0 =>
next_st := state1;
--This is fetch cycle
AR := PC;
when state1 =>
next_st := state2;
ARI := TO_INTEGER(AR);
--This is fetch cycle
DR := Memory (ARI);
when state2 =>
next_st := state3;
--This is fetch cycle
PC <= PC + 1;
IR := DR (7 downto 5);
AR := DR (3 downto 0);

when state3 =>
--This is execute cycle

case IR is
  when "111" =>
    --The op code is CLA
    ACL <= "00000000";
    next_st := state0;
  when "001" =>
    --The op code is ADD
    ARI := TO_INTEGER(AR);
    DR := memory (ARI);
    ACL <= ACL + DR;
    next_st := state0;
  when "010" =>
    --The op code is MULT
    ARI := TO_INTEGER(AR);
    DR := memory (ARI);
    PR := ACL * DR;
    ACL <= PR (7 downto 0);
    ACH <= PR (15 downto 8);
    next_st := state0;
when "011" =>
   -- The op code is DIVID
   ARI := TO_INTEGER(AR);
   DR := memory(ARI);
   ACL <= ACL / DR;
   next_st := state0;

when "100" =>
   -- The op code is XOR
   ARI := TO_INTEGER(AR);
   DR := memory(ARI);
   ACL <= ACL XOR DR;
   next_st := state0;

when "110" =>
   -- The op code is NAND
   ARI := TO_INTEGER(AR);
   DR := memory(ARI);
   ACL <= ACL NAND DR;
   next_st := state0;

when "101" =>
   -- The op code is PRITY
   ACL(7) <= z(0);
   next_st := state0;

when "000" => null;
   -- The op code is HALT
   next_st := state3;

when others => null;
end case;
when others => null;
end case;
pres_st := next_st;
end if;
end process cpu;
end Behavioral_comp;

Verilog Basic Computer Memory Program
module computer_basic (PC, clk_master, ACH, ACL, Reset, ON_OFF);
parameter state0 = 2'b00;
parameter state1 = 2'b01;
parameter state2 = 2'b10;
parameter state3 = 2'b11;

output [3:0] PC;
input clk_master;
output Reset;
input ON_OFF;
output [7:0] ACH;
output [7:0] ACL;
reg [1:0] pres_st;
reg [1:0] next_st;
reg Reset;
reg [3:0] PC;
reg [3:0] AR;
reg [7:0] DR;
reg [2:0] IR;
reg [7:0] ACH;
reg [7:0] ACL;
reg [15:0] PR;
reg [7:0] Memory [0:15];

The above statement can be written using the reduction XOR as:

\[
\text{assign } z = ^\wedge \text{ACL}[6:0];
\]

assign clk = clk_master & ON_OFF;

always @ (Reset, PC, ACL, ACH, posedge(clk), z, pres_st)
begin

if (Reset == 1'b1)
begin
    pres_st = state0;
    Reset = 1'b0;
    PC = 4'd0;
    Memory [0] = 8'hE0; Memory [1] = 8'h29;
    Memory [2] = 8'h8A; Memory [3] = 8'h4B;
    Memory [4] = 8'h6C; Memory [5] = 8'h8D;
    Memory [6] = 8'hCE; Memory [7] = 8'hA0;
    Memory [8] = 8'h00; Memory [9] = 8'h0C;
    Memory [10] = 8'h05; Memory [11] = 8'h04;
    Memory [12] = 8'h09; Memory [13] = 8'h03;
    Memory [14] = 8'h09;
    Memory [15] = 8'h07;
end

end

case (pres_st)

state0 :
begin
    next_st = state1;
    AR = PC;
end
state1 :
//This is fetch cycle
begin
    next_st = state2;
    DR = Memory [AR];
end

state2 :
//This is fetch cycle
begin
    next_st = state3;
    PC = PC + 1;
    IR = DR [7:5];
    AR = DR [3:0];
end

state3 :
//This is execute cycle
begin
    case (IR)
        3'd7 :
            //The op code is CLA
            begin
                ACL = 8'd0;
                next_st = state0;
            end
        3'd1 :
            //The op code is ADD
            begin
                DR = Memory [AR];
                ACL = ACL + DR;
                next_st = state0;
            end
    endcase
end
end

3'd2 :
//The op code is MULT
begin
    DR = Memory [AR];
    PR = ACL * DR;
    ACL = PR [7:0];
    ACH = PR [15:8];
    next_st = state0;
end

3'd3 :
//The op code is DIVID
begin
    DR = Memory [AR];
    ACL = ACL / DR;
    next_st = state0;
end

3'd4 :
//The op code is XOR
begin
    DR = Memory [AR];
    ACL = ACL ^ DR;
    next_st = state0;
end

3'd6 :
//The op code is NAND
begin
    DR = Memory [AR];
    ACL = ~(ACL & DR);
    next_st = state0;
3'd5 :
    //The op code is PRITY
    begin
        ACL[7] = z;
        next_st = state0;
    end

3'd0 :
    //The op code is HALT
    begin
        next_st = state3;
    end

default :
    begin
    end
endcase

default :
    begin
    end
endcase
pres_st = next_st;
endmodule
**FIGURE 7.13** Simulation output of the accumulator register.

<table>
<thead>
<tr>
<th>PC</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
</tr>
</thead>
<tbody>
<tr>
<td>IR</td>
<td>111</td>
<td>001</td>
<td>100</td>
<td>010</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>ACL</td>
<td>0</td>
<td>12</td>
<td>9</td>
<td>36</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ACH</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ASSIGNMENT QUESTIONS

1) Why mixed type description is needed? Explain.
2) Write a VHDL code for finding largest element of an array.
3) Describe the development of HDL code for an Arithmetic Logic unit and write
   VHDL/verilog code for an ALU shown in fig. Assume the following operations:
   addition, multiplication, division, no operation.
4) Write a note on packages in VHDL
5) Write VHDL code for addition of two 5X5 matrices using a package.
6) Write the block diagram and function table of a SRAM. Using these, write a verilog
description for 16X8 SRAM.
UNIT 7: HIGHLIGHTS OF MIXED-LANGUAGE DESCRIPTION

Syllabus of unit 7: Hours : 7
Mixed –Language Descriptions: Highlights of Mixed-Language Description, How to invoke One language from the Other, Mixed-language Description Examples, Limitations of Mixed-Language Description

Recommended readings:

2. Verilog HDL –Samir Palnitkar-Pearson Education
4. A Verilog HDL Primer- J.Bhaskar – BS Publications
5. Circuit Design with VHDL-Volnei A.Pedroni-PHI
UNIT 7: HIGHLIGHTS OF MIXED-LANGUAGE DESCRIPTION

Facts
- To write HDL code in mixed-language, the simulator used with the HDL package should be able to handle a mixed-language environment.
- In the mixed-language environment, both VHDL and Verilog module files are made visible to the simulator.
- In the mixed-language environment, both VHDL and Verilog Libraries are made visible to the simulator.
- Mixed-language environment has many limitations; but the development of simulators that can handle mixed-language environments with minimum constraints is underway. One of these major constraints is that a VHDL module can only invoke the entire Verilog module; and a Verilog module can only invoke a VHDL entity. For example, we cannot invoke a VHDL procedure from a Verilog module.
- Mixed-language description can combine the advantages of both VHDL and Verilog in one module. For example, VHDL has more-extensive file operations than Verilog, including write and read. By writing mixed-language, we can use the VHDL file operations in a Verilog module.

HOW TO INVOKE ONE LANGUAGE FROM THE OTHER

When writing VHDL code, you can invoke (import) a Verilog module; if you are writing Verilog code, you can invoke (import) a VHDL entity. The process is similar in concept to invoking procedures, functions, tasks and packages. For example, by instantiating a VHDL package in a Verilog module, the contents of this package are made visible to the module. Similarly, by invoking a Verilog module in a VHDL module, all information in the Verilog module is made visible to the VHDL module.

How to Invoke a VHDL Entity from a Verilog Module

In Verilog, module instantiates a module with the same name as the VHDL entity; the parameters of the module should match the type and port directions of the entity. VHDL ports that can be mapped to Verilog modules are: in, out and inout; buffer is not allowed. Only the entire VHDL entity can be made visible to the Verilog module.

Invoking a VHDL Entity from a Verilog Module
Module mixed (a,b,c,d);
Input a,b;
Output c,d;
…………
VHD_entity V1(a,b,c,d);
How to Invoke a Verilog Module from a VHDL Module

In the VHDL module, we declare a component with the same name as the Verilog module we want to invoke. The name and port modes of the component should be identical to the name and input/output modes of the Verilog module.

Invoking a Verilog Module from a VHDL Module

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity VHD_enty is
    port( x, y : in std_logic;
         o1,o2 : out std_logic);
end VHD_enty;
architecture VHD_enty of VHD_enty is
begin
    ...........
end VHD_enty;
```

Mixed-Language Description Examples
Invoking a VHDL Entity from a Verilog Module

VHDL entity is invoked in a Verilog module by instantiating the Verilog module with a name that is identical to the entity’s name.

Mixed-Language Description of a Full Adder
**Figure 9.1** Full adder as two half adders.
--This is the Verilog module

```verilog
module Full_Adder1 (x, y, cin, sum, carry);
  input x, y, cin;
  output sum, carry;
  wire c0, c1, s0;

  HA H1 (y, cin, s0, c0);
  HA H2 (x, s0, sum, c1);

  // Description of HA is written in VHDL in the entity HA

  or (carry, c0, c1);
endmodule
```

library IEEE;
use ieee.std_logic_1164.all;
entity HA is

--For correct binding between this VHDL code and the above Verilog
--code, the entity has to be named HA

  port (a, b : in std_logic; s, c : out std_logic);
end HA;
architecture HA_Dtflw of HA is
begin
  s <= a xor b;
  c <= a and b;
end HA_Dtflw;
```
Mixed-Language Description of a 9-Bit Adder

module Nine_bitAdder (a, b, c0, sum_total, carry_out);
  input [8:0] a, b;
  input c0;
  output [8:0] sum_total;
  output carry_out;
  wire cr0, cr1;

  //Invoke the VHDL entity
  adders_RL A1 (a [2:0], b [2:0], c0, sum_total [2:0], cr0);
  adders_RL A2 (a [5:3], b [5:3], cr0, sum_total [5:3], cr1);
  adders_RL A3 (a [8:6], b [8:6], cr1, sum_total [8:6], carry_out);

  //adders_RL is the name of the VHDL entity
endmodule

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

-- This is a VHDL data-flow code for a 3-bit carry-lookahead adder
entity adders_RL is
  port (x, y : in std_logic_vector (2 downto 0);
        cin : in std_logic;
        sum : out std_logic_vector (2 downto 0);
        cout : out std_logic);

  --The entity name is identical to that of the Verilog module.
  --The input and output ports have the same mode as the inputs
  --and outputs of the Verilog module.
end adders\_RL;

architecture lkh\_DtFl of adders\_RL is

signal c0, c1 : std\_logic;
signal p, g : std\_logic\_vector (2 downto 0);
constant delay\_gt : time := 0 ns;
--The gate propagation delay here is equal to 0.
begin

g(0) <= x(0) and y(0) after delay\_gt;
g(1) <= x(1) and y(1) after delay\_gt;
g(2) <= x(2) and y(2) after delay\_gt;
p(0) <= x(0) or y(0) after delay\_gt;
p(1) <= x(1) or y(1) after delay\_gt;
p(2) <= x(2) or y(2) after delay\_gt;
c0 <= g(0) or (p(0) and cin) after 2 * delay\_gt;

c1 <= g(1) or (p(1) and g(0)) or (p(1) and
   p(0) and cin) after 2 * delay\_gt;
cout <= g(2) or (p(2) and g(1)) or (p(2) and p(1) and g(0)) or
   (p(2) and p(1) and p(0) and cin) after 2 * delay\_gt;

sum(0) <= (p(0) xor g(0)) xor cin after delay\_gt;
sum(1) <= (p(1) xor g(1)) xor c0 after delay\_gt;
sum(2) <= (p(2) xor g(2)) xor c1 after delay\_gt;
end lkh\_DtFl;

FIGURE 9.3 Block diagram of a 3-bit adder with a zero flag.
Mixed-Language Description of a 3-Bit Adder with Zero Flag

module three_bitAdd (A, B, cin, Sum_3, Carry_out, Z_flag);
  input [2:0] A, B;
  input cin;
  output [2:0] Sum_3;
  output Carry_out;
  output Z_flag;
  wire cr0, cr1;

  full_add FA0 (A[0], B[0], cin, Sum_3[0], cr0);
  full_add FA1 (A[1], B[1], cr0, Sum_3[1], cr1);
  full_add FA2 (A[2], B[2], cr1, Sum_3[2], Carry_out);

--The above modules invoke the VHDL entity full_add

  assign Z_flag = ~(Sum_3[0] | Sum_3[1] | Sum_3[2] | Carry_out);
endmodule

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity full_add is
  Port (X, Y, cin : in std_logic; sum, cout : out std_logic);
endentity
This is a 1-bit full adder component built from AND-OR-NOT gates; see Figure 9.4.

architecture beh_vhdl of full_add is
--Instantiate the components of a 1-bit adder;
--see Figure 9.4.
component inv
    port(I1 : in std_logic; O1 : out std_logic);
end component;
component and2
    port(I1, I2 : in std_logic; O1 : out std_logic);
end component;
component and3
    port(I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;
component or3
    port(I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;
component or4
    port(I1, I2, I3, I4 : in std_logic; O1 : out std_logic);
end component;
for all : inv use entity work.bind1 (inv_0);
for all : and2 use entity work.bind2 (and2_0);
for all : and3 use entity work.bind3 (and3_0);
for all : or3 use entity work.bind3 (or3_0);
for all : or4 use entity work.bind4 (or4_0);
--The above five “for” statements are to bind the inv, and3, --and2, or3, and or4 with the architecture beh_vhdl.
--See Chapter 4, “Structural Descriptions.”
signal Xbar, Ybar, cinbar, s0, s1, s2, s3, s4, s5, s6 : std_logic;
beg
Iv1 : inv port map (X, Xbar);
Iv2 : inv port map (Y, Ybar);
Iv3 : inv port map (cin, cinbar);
A1 : and3 port map (X, Y, cin, s0);
A2 : and3 port map (Xbar, Y, cinbar, s1);
A3 : and3 port map (Xbar, Ybar, cin, s2);
A4 : and3 port map (X, Ybar, cinbar, s3);
A5 : and2 port map (X, cin, s4);
A6 : and2 port map (X, Y, s5);
A7 : and2 port map (Y, cin, s6);
O1 : or4 port map (s0, s1, s2, s3, sum);
O2 : or3 port map (s4, s5, s6, cout);
end beh_vhdl;

--The following is the behavioral description of the components
--instantiated in the entity full_add.
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity bind1 is
    port (I1 : in std_logic; O1 : out std_logic);
end bind1;
architecture inv_0 of bind1 is
begin
    O1 <= not I1;
end inv_0;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bind2 is
    port (I1, I2 : in std_logic; O1 : out std_logic);
end bind2;

architecture and2_0 of bind2 is
begin
    O1 <= I1 and I2;
end and2_0;

architecture or2_0 of bind2 is
begin
    O1 <= I1 or I2;
end or2_0;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity bind3 is
    port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end bind3;

architecture and3_0 of bind3 is
begin
Mixed-Language Description of a Master-Slave D Flip-Flop

FIGURE 9.5 Logic diagram of master-slave D flip-flop.

module D_Master (D_in, clk, Q_out, Qb_out);
    input D_in, clk;
    output Q_out, Qb_out;
    wire Q0, Qb, clkb; // wire statement here can be omitted.
    assign clkb = ~ clk;
    assign clk2 = ~ clkb;
    D_Latch D0 (D_in, clkb, Q0, Qb);
    D_Latch D1 (Q0, clk2, Q_out, Qb_out);
endmodule

//D_Latch is the name of a VHDL entity describing a D-Latch
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity D_Latch is
-- The entity has the same name as the calling Verilog module
port (D, E : in std_logic;
     Q, Qbar : buffer std_logic);
end D_Latch;

architecture DL_DtFl of D_Latch is
-- This architecture describes a D-latch using
-- data-flow description
constant Delay_EorD : Time := 9 ns;
constant Delay_inv : Time := 1 ns;
begin
  Qbar <= (D and E) nor (not E and Q) after Delay_EorD;
  Q <= not Qbar after Delay_inv;
end DL_DtFl;

FIGURE 9.6 Simulation waveform of a master–slave D flip–flop.
Mixed-Language Description of a 4x4 Comparator

FIGURE 9.7 Logic diagram of the Verilog – generated statements
module compr_genr (X, Y, xgty, xlyt, xeqy);
  parameter N = 3;
  input [N:0] X, Y;
  output xgty, xlyt, xeqy;
  wire [N:0] sum, Yb;
  wire [N+1:0] carry, eq;
  assign carry[0] = 1'b0;
  assign eq[0] = 1'b1;
  assign Yb = ~Y;
  FULL_ADDER FA (X[0], Yb[0], carry[0], sum[0], carry[1]);
  -- The module FULL_ADDER has the same name
  -- as the VHDL entity FULL_ADDER
  FULL_ADDER FA1 (X[1], Yb[1], carry[1], sum[1], carry[2]);
  FULL_ADDER FA2 (X[2], Yb[2], carry[2], sum[2], carry[3]);
  FULL_ADDER FA3 (X[3], Yb[3], carry[3], sum[3], carry[4]);
  generate
    genvar i:
    for (i = 0; i <= N; i = i + 1)
      begin : u
        and (eq[i+1], sum[i], eq[i]);
      end
  endgenerate
  assign xgty = carry [N+1];
  assign xeqy = eq [N+1];
  nor (xlyt, xeqy, xgty);
endmodule
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
entity FULL_ADDER is
  Port (A, B, cin : in std_logic; sum_1, cout : out std_logic);
end FULL_ADDER;

architecture beh_vhdl of FULL_ADDER is

--This architecture is a behavioral description of a full adder
begin
  oneBit : process (A, B, cin)
    variable y : std_logic_vector (2 downto 0);
  begin
    Y := (A & B & Cin);
    --The above statement is a concatenation of
    --three bits A, B, and Cin
    case y is
      when "000" => sum_1 <= '0'; cout <= '0';
      when "110" => sum_1 <= '0'; cout <= '1';
      when "101" => sum_1 <= '0'; cout <= '1';
      when "011" => sum_1 <= '0'; cout <= '1';
      when "111" => sum_1 <= '1'; cout <= '1';
      when others => sum_1 <= '1'; cout <= '0';
    --Others here refer to 100, 001, 010
    end case;
  end process;
end beh_vhdl;

Invoking a Verilog Module from a VHDL Module
We can instantiate a Verilog module from a VHDL module by instantiating a component
in the VHDL module that has the same name and ports as the Verilog module. The
Verilog module should be the only construct that has the same name as the component.

Mixed-Language Description of an AND Gate

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

--This is the VHDL module

entity andgate is
  port (a, b : in std_logic; c : out std_logic);
end andgate;
architecture andgate of andgate is
component and2

--For correct binding with the Verilog module,
--the name of the component should be identical
--to that of the Verilog module.

port (x, y : in std_logic; z : out std_logic);

--The name of the ports should be identical to the name
--of the inputs/outputs of the Verilog module.

end component;

begin
  g1 : and2 port map (a, b, c);
end andgate;

//This is the Verilog module
module and2 (x, y, z);

input x, y;
output z;
assign z = x & y;
endmodule

Mixed-Language Description of a JK Flip-Flop

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity JK_FF is
  Port (Jx, Kx, clk, clx : in std_logic; Qx, Qxbar : out std_logic);
end JK_FF;

architecture JK_FF of JK_FF is

--The JK flip flop is declared as a component
component jk_verilog
  port(j, k, ck, clear : in std_logic; q, qb : out std_logic);
end component;

begin

jk1 : jk_verilog port map (Jx, Kx, clk, clx, Qx, Qxbar);

end JK_FF;

module jk_verilog (j, k, ck, clear, q, qb);
// The module name jk_verilog matches
// the name of the VHDL components
input j, k, ck, clear;
output q, qb;
--The input and output ports match those of the
--VHDL component, jk_verilog
reg q, qb;
reg [1:0] JK;
always @ (posedge ck, clear)
begin
  if (clear == 1)
    begin
      q = 1'b0;
      qb = 1'b1;
    end
  else
    begin
      JK = {j, k};
      case (JK)
        2'd0 : q = q;
        2'd1 : q = 0;
        2'd2 : q = 1;
        2'd3 : q = ~q;
      endcase
      qb = ~q;
    end
end
endmodule
FIGURE 9.9 Simulation waveform of a JK flip-flop with an active high clear.

Mixed-Language Description of 3-Bit Counter with Clear

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity countr_3 is
  port (clk, clrbar : in std_logic;
        q, qb : inout std_logic_vector (2 downto 0));
end countr_3;
```
architecture CNTR3 of countr_3 is

component JK_FF
    port (I1, I2, I3 : in std_logic; O1, O2 : inout std_logic);
end component;

component inv
    port (I1 : in std_logic; O1 : out std_logic);
end component;

component and2
    port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

component or2
    port (I1, I2 : in std_logic; O1 : out std_logic);
end component;

signal J1, K1, J2, K2, clr, clrb1, s1, high : std_logic;
beg

    high <= '1';

FF0 : JK_FF port map (clrb1, High, clk, q(0), qb(0));

A1 : and2 port map (clrb1, q(0), J1);
inv1 : inv port map (clr, clrb1);
inv2 : inv port map (clrb1, clr);
r1 : or2 port map (q(0), clr, K1);
FF1 : JK_FF port map (J1, K1, clk, q(1), qb(1));
A2 : and2 port map (q(0), q(1), s1);
A3 : and2 port map (clrb1, s1, J2);
r2 : or2 port map (s1, clr, K2);
FF2 : JK_FF port map (J2, K2, clk, q(2), qb(2));
end CNTR3;

module and2 (I1, I2, O1);
//This Verilog module represents an AND function

input I1, I2;
output O1;
assign O1 = I1 & I2;
endmodule

module inv (I1, O1);
//This Verilog module represents an INVERT function

input I1;
output O1;
assign O1 = ~I1;
endmodule

module or2 (I1, I2, O1);
//This Verilog module represents an OR function

input I1, I2;
output O1;
assign O1 = I1 | I2;
endmodule

module JK_FF (I1, I2, I3, O1, O2);
//This Verilog module represents a JK flip-flop.

input I1, I2, I3;
output O1, O2;
reg O1, O2;
reg [1:0] JK;
initial
begin
O1 = 1'b0;
O2 = 1'b1;
end
always @ (posedge I3)
begin

JK = {I1, I2};
case (JK)
2'd0 : O1 = O1;
2'd1 : O1 = 0;
2'd2 : O1 = 1;
2'd3 : O1 = ~O1;
endcase
O2 = ~O1;
end
endmodule
Mixed-Language Description of an N-Bit Counter with Ripple – Carry out

![Logic diagram of an n-bit synchronous counter with ripple-carry out.](image)

FIGURE 9.10 Logic diagram of an n-bit synchronous counter with ripple-carry out.

Mixed-Language Description of an N-Bit Asynchronous Counter

```vhdl
--This is a VHDL module
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity asynch_ctrMx is
Generic (N : integer := 3);
port (clk, clear : in std_logic;
     C, Cbar : out std_logic_vector (N-1 downto 0);
     rco : out std_logic);
end asynch_ctrMx;

architecture CT_strgnt of asynch_ctrMx is

component jkff is
--This is a JK flip-flop with a clear bound to Verilog module jkff
    port(j, k, clk, clear : in std_logic; q, qb : out std_logic);
end component;

component andgate is
--This is a three-input AND gate bound to Verilog module andgate
    port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;

signal h, l : std_logic;
signal s : std_logic_vector (N downto 0);
signal s1 : std_logic_vector (N downto 0);
signal C_tem : std_logic_vector (N-1 downto 0);
```

--This is a VHDL module
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity asynch_ctrMx is
Generic (N : integer := 3);
port (clk, clear : in std_logic;
     C, Cbar : out std_logic_vector (N-1 downto 0);
     rco : out std_logic);
end asynch_ctrMx;

architecture CT_strgnt of asynch_ctrMx is

component jkff is
--This is a JK flip-flop with a clear bound to Verilog module jkff
    port(j, k, clk, clear : in std_logic; q, qb : out std_logic);
end component;

component andgate is
--This is a three-input AND gate bound to Verilog module andgate
    port (I1, I2, I3 : in std_logic; O1 : out std_logic);
end component;

signal h, l : std_logic;
signal s : std_logic_vector (N downto 0);
signal s1 : std_logic_vector (N downto 0);
signal C_tem : std_logic_vector (N-1 downto 0);
```
begin
    h <= '1';
    l <= '0';
    s <= (C_tem & clk);

-- s is the concatenation of Q and clk. We need this
-- concatenation to describe the clock of each JK flip-flop.

    s1(0) <= not clear;

Gnlop : for i in (N - 1) downto 0 generate
    G1 : jkff port map (h, h, s(i), clear, C_tem(i), Cbar(i));
end generate GnLop;

C <= C_tem;

rc_gen : for i in (N - 2) downto 0 generate
    --This loop to determine the ripple carry-out
    rc : andgate port map (C_tem(i), C_tem(i+1), s1(i), s1(i+1));
end generate rc_gen;

rco <= s1(N-1);
end CT_strgnt;

module jkff (j, k, clk, clear, q, qb);
// This is a behavioral description of a JK flip-flop

    input j, k, clk, clear;
    output q, qb;
    reg q, qb;
    reg [1:0] JK;
    always @ (posedge clk, clear)
    begin
        if (clear == 1)
            begin
                q = 1'b0;
                qb = 1'b1;
            end
        else
            begin
                JK = {j,k};
                case (JK)
                    2'd0 : q = q;
                    ...
```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2x1_mxd is
    Port (a, b, Sel, E : in std_logic; ybar : out std_logic);
end

architecture mux2x1switch of mux2x1_mxd is

    component nmos_verlg
        --This component, after linking to a Verilog module, behaves as an
        --nmos switch
    port (O1 : out std_logic; I1, I2 : in std_logic);
    end component;

begin

end
```

**FIGURE 9.11** Simulation waveform for an n-bit asynchronous counter.

### Mixed-Language Description of a 2x1 Multiplexer

--This is the VHDL module.

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2x1_mxd is
    Port (a, b, Sel, E : in std_logic; ybar : out std_logic);
end

architecture mux2x1switch of mux2x1_mxd is

    component nmos_verlg
        --This component, after linking to a Verilog module, behaves as an
        --nmos switch
    port (O1 : out std_logic; I1, I2 : in std_logic);
    end component;

begin

end
```

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2x1_mxd is
    Port (a, b, Sel, E : in std_logic; ybar : out std_logic);
end

architecture mux2x1switch of mux2x1_mxd is

    component nmos_verlg
        --This component, after linking to a Verilog module, behaves as an
        --nmos switch
    port (O1 : out std_logic; I1, I2 : in std_logic);
    end component;

begin

end
```
end component;

component pmos_verlg
  --This component, after linking to a Verilog module, behaves as a
  --pmos switch
  port (O1 : out std_logic; I1, I2 : in std_logic);
end component;

--constant vdd : std_logic := '1';
--constant gnd : std_logic := '0';

-- In Chapter 5 we wrote Vdd and gnd as constants.
-- Some VHDL/Verilog simulators do not transfer constants
-- between VHDL and Verilog. So we wrote them as signals.
signal vdd, gnd, Selbar, s0, s1, s2, s3 : std_logic;

begin
  vdd <= '1';
  gnd <= '0';

  --Invert signal Sel. If the complement of Sel is available,
  --then no need for the following pair of transistors.
  v1 : pmos_verlg port map (Selbar, vdd, Sel);
  v2 : nmos_verlg port map (Selbar, gnd, Sel);

  --Write the pull-down combination
  n1 : nmos_verlg port map (s0, gnd, E);
  n2 : nmos_verlg port map (s1, s0, Sel);
  n3 : nmos_verlg port map (ybar, s1, a);
  n4 : nmos_verlg port map (s2, s0, Selbar);
  n5 : nmos_verlg port map (ybar, s2, b);

  --Write the pull-up combination
  p1 : pmos_verlg port map (ybar, vdd, E);
  p2 : pmos_verlg port map (ybar, s3, Sel);
  p3 : pmos_verlg port map (ybar, s3, a);
  p4 : pmos_verlg port map (s3, vdd, Selbar);
  p5 : pmos_verlg port map (s3, vdd, b);
end mux2x1switch;

// This is the Verilog Module

module nmos_verlg (O1, I1, I2);
    input I1, I2;
    output O1;
    nmos (O1, I1, I2);
endmodule

module pmos_verlg (O1, I1, I2);
    input I1, I2;
    output O1;
    pmos (O1, I1, I2);
endmodule

Instantiating CASEX in a VHDL Module

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity P_encodr is
    Port (X : in std_logic_vector (3 downto 0);
          Y : out std_logic_vector (3 downto 0));
end P_encodr;

architecture P_encodr of P_encodr is

component cas_x
    --The name of the component is identical to the name of the
    --Verilog module
    port (a : in std_logic_vector (3 downto 0);
          b : out std_logic_vector (3 downto 0));
end component;

begin
    ax : cas_x port map (X, Y);
end P_encodr;

module cas_x (a, b);
input [3:0] a;
output [3:0] b;
reg [3:0] b;
always @ (a)
beg
    casex (a)
    4'bx11 : b = 4'd1;
    4'bx10 : b = 4'd2;
    4'bxx0 : b = 4'd4;
    4'b1000 : b = 4'd8;
    default : b = 4'd0;
endcase
end
endmodule

Mixed-Language Description of a Simple RC Filter


library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use std.textio.all;
use ieee.numeric_std.all;

entity Filter_draw is
Port (w, w_ctoff : in std_logic_vector (3 downto 0);
    Hw_vhd : out std_logic_vector (7 downto 0));
end Filter_draw;

architecture Filter_draw of Filter_draw is

Function TO_Intgr (a : in std_logic_vector) return integer is
    --This Function converts std_logic_vector type to integer
    variable result : integer;
    begin
    begin
    end

begin
  result := 0;
  lop1 : for i in a' range loop
  if a(i) = '1' then
    result := result + 2**i;
  end if;
  end loop;
return result;
end TO_Intgr;

component flter_RC
  --The name of the component “flter_RC” is the same name as the
  --Verilog module

    port (I1, I2 : in std_logic_vector (3 downto 0); O1 : out
        std_logic_vector (7 downto 0));
end component;
signal Hw_tmp : std_logic_vector (7 downto 0);
begin
  dw : flter_RC port map (w, w_ctoff, Hw_tmp);

  //output the data on a file
  fl : process (w, w_ctoff, Hw_tmp)
  file outfile : text;
  variable fstatus : file_open_status;
  variable temp : line;
  variable Hw_int, w_int, w_ctoff_intg : integer;
begin
  --Files can take integer, real, or character;
  --they cannot take std-logic-vector; so convert to integer.
  Hw_int := TO_Intgr (Hw_tmp);
  w_int := TO_Intgr (w);
  w_ctoff_intg := TO_Intgr (w_ctoff);
  file_open (fstatus, outfile, "Wfile_int.txt", write_mode);
  --The file name is Wfile_int.txt
  --Write headings. Be sure the simulator supports formatted output.
  --otherwise take out all formatted output statements
  write (temp,"
This is a Simple
  R-C Low Pass Filter");
  --The above statement when entered in the VHDL module should be
  --entered in one line without carriage return */
writeline (outfile, temp);
write (temp, " ");
writeline (outfile, temp);
write (temp, " FREQUENCY CUTOFF Amplitude Square");
--The above statement when entered in the VHDL module should be
--entered in one line without carriage return

writeline (outfile, temp);
write (temp, " ");

--write the values of the filter parameters
write (temp, w_int);
write (temp, " ");
write (temp, w_ctoffintg);
write (temp, " ");
write (temp, Hw_int);
writeline (outfile, temp);

file_close (outfile);
Hw_vhd <= Hw_tmp;
end process fl;
end Filter_draw;

// Next we write the Verilog module;
// the module performs a real division

module filter_RC (I1, I2, O1);

//The module performs the real division O1 = 1/[(I1/I2)**2 + 1]
input [3:0] I1, I2;
output [7:0] O1;
reg [7:0] O1;
real S, s1;
always @ (I1, I2)
begin
  s1 = ((1.0*I1)/(1.0 * I2))**2 ;
  //we multiply by 1.0 so the division is done in real format.
  S = 1.0 / (1.0 + s1);
  O1 = 100.00 * S;
end
endmodule
FIGURE 9.13 The file Wfile_int.txt after simulation.

LIMITATIONS OF MIXED-LANGUAGE DESCRIPTION

- Not all VHDL data types are supported in mixed-language description. Only bit, bit_vector, std_logic, std_ulogic, std_logic_vector and std_ulogic_vector are supported.
- The VHDL port type buffer is not supported.
- Only a VHDL component construct can invoke a Verilog module. We cannot invoke a Verilog module from any other construct in the VHDL module.
- A Verilog module can only invoke a VHDL entity. It cannot invoke any other construct in the VHDL module, such as a procedure or function.
ASSIGNMENT QUESTIONS

1) How to invoke a VHDL entity from a verilog module.
2) How to invoke a verilog module from VHDL module? Write a mixed language
description of an ‘OR’ gate where VHDL code invokes the verilog module ‘OR3’(3 input
OR gate)
3) Write a mixed language description of a 4 bit adder with zero flag.
4) Explain mixed language description of a JK flip-flop with a clear pin and write the
simulation waveform.
5) How to invoke a verilog module from a VHDL module? Explain with an example of a
mixed language description for a full adder using 2 half adders.
6) Write a mixed language description of a 9-bit adder consisting of three 3 – bit carry-
look ahead adders to show how a verilog module invokes VHDL entity.

UNIT 8: SYNTHESIS BASICS

Syllabus of unit 8: 

Highlights of Synthesis, Synthesis information from Entity and Module, Mapping
Process and Always in the Hardware Domain

Recommended readings:

1. **HDL Programming (VHDL and Verilog)**- Nazeih M.Botros- Dreamtech Press
2. **Verilog HDL** –Samir Palnitkar-Pearson Education
4. **A Verilog HDL Primer**- J.Bhaskar – BS Publications
5. **Circuit Design with VHDL**- Volnei A.Pedroni-PHi
UNIT 8: SYNTHESIS BASICS

HIGHLIGHTS OF SYNTHESIS

Synthesis converts HDL behavioral code into logical gates or components. These logical gates and components can be downloaded into an electronic chip.

Facts

- Synthesis maps between the simulation (software) domain and the hardware domain.
- Synthesis can be viewed as reverse engineering. The user is provided with the behavioral code and is asked to develop the logic diagram.
- Not all HDL statements can be mapped into the hardware domain. The hardware domain is limited to signals that can take zeros, ones or that are left open. The hardware domain cannot differentiate, for example, between signals and variables, as does the simulation (software) domain.
- To successfully synthesize behavior code into a certain electronic chip, the mapping has to conform to the requirements and constraints imposed by the electronic chip vendor.
- Several synthesis packages are available on the market. These packages can take behavior code, map it, and produce a net list that is downloaded into the chip.
Two synthesizers may synthesize the same code using a different number of the same gates. This is due to the different approaches taken by the two synthesizers to map the code.

**Synthesis steps**
Formulate a flowchart of the system.

Use the flow chart to write a behavioral HDL description.

Simulate the behavioral module and verify that the simulation correctly describes the system.

Map the behavioral statements into components and logic gates. See Chapter 10, "Synthesis Basics".

Verify that the mapping is correct by writing a structural description to simulate the mapped components and logic gates.

Use CAD tools to download the components and logic gates onto an electronic chip such as FPGA.

Test the electronic chip to verify the download.

SYNTHESIS INFORMATION FROM Entity AND Module
Entity (VHDL) or Module in (Verilog) provides information on the inputs and outputs and their types.
Synthesis Information from Entity (VHDL)

**VHDL code for Entity system1**

```vhdl
entity system1 is
    port ( a, b : in bit; d : out bit);
end system1;
```

system1 has two input signals, each of 1 bit and one output signal of 1 bit. Each signal can take 0 (low) or 1 (high).

![Diagram of system1](image)

**VHDL code for Entity system2**

```vhdl
entity system2 is
    port ( a, b : in std_logic; d : out std_logic);
end system2;
```

system2 has two 1-bit input signals and one 1-bit output signal. Each signal can take 0 (low), 1 (high), or high impedance (open).

**VHDL code for Entity system3**

```vhdl
entity system3 is
    port ( a, b : in std_logic_vector (3 downto 0);
        d : out std_logic_vector (7 downto 0));
end system3;
```

system3 has two 4-bit input signals and one 8-bit output signal. The input signals can be binary or left open.

![Diagram of system3](image)

**VHDL code for Entity system4**

```vhdl
entity system4 is
    port ( a, b : in signed (3 downto 0);
```
d : out std_logic_vector( 7 downto 0));
end system4;

system4 has two 4-bit input signals and one 8-bit output signal. The input signal is binary and the output signal can be binary or high impedance.

**VHDL code for Entity system5**

entity system5 is
port (a, b : in unsigned (3 downto 0);
      d : out std_logic_vector( 7 downto 0));
end system5;

**VHDL code for Entity system6**

entity system6 is
port (a, b : in unsigned (3 downto 0);
      d : out integer range -10 to 10);
end system6;

system6 has two 4-bit input signals and one 5-bit output signal. In the hardware domain, the integer is represented by binary.

**FIGURE 10.4** Synthesis information

**VHDL code for Entity system7**

entity system7 is
generic(N: integer := 4; M : integer := 3);
port (a, b : in std_logic_vector (N downto 0);
      d : out std_logic_vector (M downto 0));
end system7;

Since N = 4 and M = 3, system7 has two 5-bit input signals and one 4-bit output signal. All signals are in binary. N and M have no explicit hardware mapping.
FIGURE 10.5 Synthesis information

**MAPPING Process AND always IN THR HARDWARE DOMAIN**

Process (VHDL) and always (Verilog) are the major behavioral statements. These statements are frequently used to model systems with data storage, such as counters, registers and CPUs.

**Mapping the Signal – Assignment Statement to Gate - Level**

**VHDL Code for a Signal-Assignment Statement, Y = X—VHDL and Verilog**

VHDL Signal-Assignment Statement, Y = X

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

entity SIGNA_ASSN is
 port (X : in bit; Y : out bit);
end SIGNA_ASSN;

architecture BEHAVIOR of SIGNA_ASSN is
begin
 P1 : process (X)
 begin
  Y <= X;
 end process P1;
end BEHAVIOR;
```

Verilog Signal-Assignment Statement, Y = X

```verilog
module SIGNA_ASSN (X, Y);
input X;
output Y;
reg y;
always @ (X)
 begin
  Y = X;
 end
endmodule
```
FIGURE 10.14 Gate–level synthesis (a) Logic symbol (b) Gate-level logic diagram.

VHDL Code for a Signal-Assignment Statement, \( Y = 2 * X + 3 \) — VHDL and Verilog

VHDL Signal-Assignment Statement, \( Y = 2 * X + 3 \)

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

entity sign_assn2 is
  port (X : in unsigned (1 downto 0);
        Y : out unsigned (3 downto 0));
end ASSN2;

architecture BEHAVIOR of sign_assn2 is
begin
  P1 : process (X)
  begin
    Y <= 2 * X + 3;
  end process P1;
end BEHAVIOR;
```

Verilog Signal-Assignment Statement, \( Y = 2 * X + 3 \)

```verilog
module sign_assn2 (X, Y);
input [1:0] X;
output [3:0] Y;
reg [3:0] Y;
always @ (X)
begin
  Y = 2 * X + 3;
end
```

endmodule

FIGURE 10.15 Gate – level synthesis (a) Logic symbol (b) Gate-level logic diagram.

Structural Verilog Code for the Logic Diagram in Figure 10.15b.

module sign_struc(X, Y);
input [1:0] X;
output [3:0] Y;
reg [3:0] Y;
always @ (X)
begin
  Y[0] = 1'b1;
  Y[1] = ~ X[0];
  Y[2] = X[0] ^ X[1];
  Y[3] = X[1] & X[0];
end

endmodule
The variable – assignment statement is a VHDL statement. Verilog does not distinguish between signal and variable – assignment statements.

VHDL Variable-Assignment Statement

library ieee;
use ieee.std_logic_1164.all;

entity parity_even is
    port (x : in std_logic_vector (3 downto 0);
         C : out std_logic);
end parity_even;

architecture behav_prti of parity_even is
begin

    P1 : process (x)
    variable c1 : std_logic;
    begin
        c1 := (x(0) xor x(1)) xor (x(2) xor x(3));
        C <= c1;
    end process P1;
end behav_prti;

FIGURE 10.15 Gate – level synthesis (a) Logic symbol (b) Gate-level logic diagram.

FIGURE 10.16 Simulation waveform

Mapping the Variable – Assignment Statement to Gate – Level Synthesis
FIGURE 10.17 Gate – level synthesis (a) Logic symbol (b) Gate-level logic diagram.

Mapping Logical Operators—VHDL and Verilog

**VHDL: Mapping Logical Operators**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity decod_var is
    port (a : in std_logic_vector (1 downto 0);
         D : out std_logic_vector (3 downto 0));
end decod_var;

architecture Behavioral of decod_var is

begin
    dec : process (a)
    variable a0bar, a1bar : std_logic;
    begin
        a0bar := not a(0);
        a1bar := not a(1);
        D(0) <= not (a0bar and a1bar);
        D(1) <= not (a0bar and a(1));
        D(2) <= not (a(0) and a1bar);
        D(3) <= not (a(0) and a(1));
    end process dec;
end Behavioral;
```

**Verilog: Mapping Logical Operators**

```verilog
module decod_var (a, D);
input [1:0] a;
```
output [3:0] D;
reg a0bar, a1bar;
reg [3:0] D;
always @ (a)
begins
    a0bar = ~ a[0];
    a1bar = ~ a[1];
    D[0] = ~ (a0bar & a1bar);
    D[1] = ~ (a0bar & a[1]);
    D[2] = ~ (a[0] & a1bar);
    D[3] = ~ (a[0] & a[1]);
end
endmodule

**FIGURE 10.18** Gate – level synthesis (a) Logic symbol (b) Gate-level logic diagram.
Mapping the If Statement
Example of if-else statement

a) VHDL Description

process (a, x)
begin
   if (a = '1') then
      Y <= X;
   else
      Y <= '0';
   end if;
end process;

b) Verilog description

always @(a, X)
begin
   if (a == 1'b1)
      Y = X;
   else
      Y = 1'b0;
end

FIGURE 10.19 Gate – level synthesis
Example of if-else statement. a) VHDL. b) Verilog

a) VHDL Description

process (a, X, X1)
begin
   if (a = '1') then
      Y <= X;
   else
      Y <= X1;
   end if;
end process;

b) Verilog Description

always @(a, X, X1)
begin
   if (a == 1'b1)
      Y = X;
FIGURE 10.20 Gate – level synthesis
Example of Comparison Using If-else Statement—VHDL and Verilog

**VHDL If-Else Statement**

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity IF_st is
  port (a : in std_logic_vector (2 downto 0); Y : out Boolean);
end IF_st;

architecture IF_st of IF_st is
begin

IfB : process (a)
variable tem : Boolean;
begin
  if (a < "101") then
    tem := true;
  else
    tem := false;
  end if;
  Y <= tem;
end

end IF_st;
```

**Verilog If-Else Statement**

```verilog
module IF_st (a, Y);
input [2:0] a;
output Y;
reg Y;
always @ (a)
begin
  if (a < 3'b101)
    Y = 1'b1;
else
  Y = X1;
end

end IF_st;
```
else
  Y = 'b0;
end
endmodule

FIGURE 10.22 Gate – level synthesis
Example of Elseif and Else-If—VHDL and Verilog

VHDL Elseif Description

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;

entity elseif is
  port (BP : in natural range 0 to 7;
        ADH : out natural range 0 to 15);
  end;

architecture elseif of elseif is
  begin
      ADHP : process(BP)
          variable resADH : natural := 0;
          begin

              if BP <= 2 then resADH := 15;
              elsif BP >= 5 then resADH := 0;
              else
                  resADH := BP * (-5) + 25;
              end if;

end if;
end;
ADH <= resADH;
end process ADHP;
end elseif;

Verilog Else-If Description
module elseif (BP, ADH);
input [2:0] BP;
output [3:0] ADH;
reg [3:0] ADH;
always @ (BP)
begin

    if (BP <= 2) ADH = 15;
    else if (BP >= 5) ADH = 0;
    else
        ADH = BP * (-5) + 25;
    end

endmodule

FIGURE 10.24 Gate – level synthesis
Example of If Statement with Storage-VHDL and Verilog

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity If_store is
port (a, X : in std_logic; Y : out std_logic);
end If_store;

architecture If_store of If_store is
begin
    process (a, X)
    begin
        if (a = '1') then
            Y <= X;
        end if;
    end process;
end If_store;

Verilog If Statement with Storage
module If_store (a, X, Y);
input a, X;
output Y;
reg Y;
always @ (a, X)
begin
    if (a == 1'b1)
        Y = X;
endmodule
**Else-If Statement with Gate-Level Logic**

```vhdl
package weather_fcst is
  Type unit is (cent, half, offset);
end package weather_fcst;

library ieee;
use ieee.std_logic_1164.all;

use work.weather_fcst.all;
entity weather is
  port (a : in unit; tempr : in integer range 0 to 15; z : out integer range 0 to 15);
end weather;

architecture weather of weather is
begin
  T : process (a, tempr)
  variable z_tem : integer range 0 to 15;
  begin
    if ((tempr <= 7) and (a = cent)) then
      z_tem := tempr;
    elsif ((tempr <= 7) and (a = offset)) then
      z_tem := tempr + 4;
    elsif ((tempr <= 7) and (a = half)) then
      z_tem := tempr / 2;
    else
      z_tem := 15;
    end if;
    z <= z_tem;
  end process T;
end weather;
```
FIGURE 10.26 RTL synthesis

<table>
<thead>
<tr>
<th>a(1)a(0)</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>temp(3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Select (0)

Select (1)

FIGURE 10.28 Gate – level synthesis
Mapping the case Statement
Example of Case Mapping

module case_nostr (a, b, ct, d);
  input [3:0] a, b;
  input ct;
  output [4:0] d;
  reg [4:0] d;
  always @ (a, b, ct)
  begin
    case (ct)
      1'b0 : d = a + b;
      1'b1 : d = a - b;
    endcase
  end
endmodule

Case Statement with Storage

module case_str (a, b, ct, d);
  input [3:0] a, b;
  input ct;
  output [4:0] d;
  reg [4:0] d;
  always @ (a, b, ct)
  begin
    case (ct)
      1'b0 : d = a + b;
  end
endmodule
module Encoder_4 (IR, RA);
input [3:0] IR;
output [3:0] RA;
reg [3:0] RA;
always @ (IR)
begin
    casex (IR)
        4'bxxx1 : RA = 4'd1;
        4'bxx10 : RA = 4'd2;
        4'bx100 : RA = 4'd4;
        4'b1000 : RA = 4'd8;
        default : RA = 4'd0;
    endcase
end
endmodule

FIGURE 10.29 RTL synthesis
Verilog Casex
Example of Case with Storage

library IEEE;
use IEEE.STD_LOGIC_1164.all;
package types is
type states is (state0, state1, state2, state3);
end;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.types.all;
entity state_machine is
  port (A, clk : in std_logic; pres_st : buffer states;
       Z : out std_logic);
end state_machine;

architecture st_behavioral of state_machine is
begin

  FM : process (clk, pres_st, A)
  variable present : states := state0;

FIGURE 10.31 Logic diagram
begin
if (clk = '1' and clk'event) then
-- clock’event is an attribute to the signal clk; the above if
-- Boolean expression means the positive edge of clk

case pres_st is
  when state0 =>
    if A = '1' then
      present := state1;
      Z <= '0';
    else
      present := state0;
      Z <= '1';
    end if;

  when state1 => if A = '1' then
    present := state2;
    Z <= '0';
  else
    present := state3;
    Z <= '0';
  end if;

  when state2 => if A = '1' then
    present := state3;
    Z <= '1';
  else
    present := state0;
    Z <= '0';
  end if;

  when state3 => if A = '1' then
    present := state0;
    Z <= '1';
end case;
else
     present := state2;
     Z <= '1';
     end if;
end case;

pres_st <= present;
end if;
end process FM;
end st_behavioral;

FIGURE 10.33 RTL logic diagram
Mapping the Loop Statement
Loop in HDL description is an essential tool for behavioral modeling. It is, however, easier to code than it is to synthesize in the hardware domain.

A For-Loop Statement. a) VHDL. b) Verilog.

a) VHDL Description
for i in 0 to 63 loop
    temp(i) := temp(i) + b(i);
end loop;

b) Verilog Description
for i=0; i<=63;i=i+1
begin
    temp[i] = temp[i] + b[i];
end

Synthesis of the Loop statement
VHDL Code Includes For-Loop

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity listing10_32 is

port (a : in std_logic_vector (3 downto 0);
     c : in integer range 0 to 15;
     b : out std_logic_vector (3 downto 0));

end listing10_32;

architecture listing10_32 of listing10_32 is
begin
    shfl : process (a, c)
    variable result, j : integer;

end
variable temp : std_logic_vector (3 downto 0);
begin

result := 0;
lop1 : for i in 0 to 3 loop
  if a(i) = '1' then
    result := result + 2**i;
  end if;
end loop;
if result > c then
  lop2 : for i in 0 to 3 loop
    j := (i + 2) mod 4;
    temp (j) := a(i);
  end loop;
else
  lop3 : for i in 0 to 3 loop
    j := (i + 1) mod 4;
    temp (j) := a(i);
  end loop;
end if;
b <= temp;
end process shfl;
end listing10_32;
FIGURE 10.35 Simulation output

FIGURE 10.36 RTL synthesis
**Mapping Procedure or Task**

Procedures, Tasks and Functions are code constructs that optimize HDL module writing. In the hardware domain, we do not have a logic for procedures or tasks, they are incorporated in the entity or the module that calls them.

**A Verilog Example of Task**

```verilog
module example_task (a1, b1, d1);
input a1, b1;
output d1;
reg d1;
always @(a1, b1)
begin
    xor_synth (d1, a1, b1);
end

task xor_synth;
output d;
input a, b;
begin
    d = a ^ b;
end
endtask
endmodule
```

**An Example of a Procedure**

```verilog
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Int_Bin is
generic (N : integer := 3);
port (X_bin : out std_logic_vector (N downto 0);
    Y_int : in integer;
    flag_even : out std_logic);
```
end Int_Bin;

architecture convert of Int_Bin is

procedure itb (bin : out std_logic_vector;
               signal flag : out std_logic;
               N : in integer; int : inout integer) is

begin
  if (int MOD 2 = 0) then
    flag <= '1';
  else
    flag <= '0';
  end if;
  for i in 0 to N loop
    if (int MOD 2 = 1) then
      bin (i) := '1';
    else
      bin (i) := '0';
    end if;
    int := int / 2;
  end loop;
end itb;

begin
  process (Y_int)
  variable tem : std_logic_vector (N downto 0);
  variable tem_int : integer;

  begin
    tem_int := Y_int;
    itb (tem, flag_even, N, tem_int);
  end process;
end


X_bin <= tem;
end process;
end convert;

FIGURE 10.38 Synthesis

Mapping the Function Statement

Functions like procedures, are simulation constructs, they optimize the HDL module writing style.

Verilog Example of a Function

module Func_synth (a1, b1, d1);
input a1, b1;
output d1;
reg d1;

always @(a1, b1)
begin

d1 = andopr (a1, b1);
end

function andopr;
input a, b;
begin
andopr = a ^ b;
end
endfunction

endmodule
FIGURE 10.39 Synthesis

Example of Function Synthesis

module Function_Synth2 (x, y);

input [2:0] x;
output [3:0] y;
reg [3:0] y;

always @ (x)
begin
y = fn (x);
end

function [3:0] fn;
input [2:0] a;
begin
if (a <= 4)
fn = 2 * a + 5;
end
endfunction
endmodule

FIGURE 10.40 Simulation output

<table>
<thead>
<tr>
<th>x</th>
<th>011</th>
<th>000</th>
<th>100</th>
<th>001</th>
<th>101</th>
<th>0111</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>1011</td>
<td>0101</td>
<td>1101</td>
<td>0111</td>
<td>0111</td>
<td>0111</td>
</tr>
</tbody>
</table>
ASSIGNMENT QUESTIONS

1) Discuss some of the important facts related to synthesis. **7 Marks**
2) Discuss synthesis information from entity with examples. **8 Marks**
3) Describe synthesis information extraction from entity and module with examples. **10 Marks**
4) Explain mapping the signal – assignment and variable assignment statements to Gate-level with suitable examples. **10 Marks**
5) Explain extraction of synthesis information from entity. **4 Marks**
6) With an example explain verilog synthesis information extraction from module inputs and outputs. **4 Marks**
7) Write VHDL/verilog code for signal assignment statement \( Y = (2 \times X + 3) \) for an entity with one input \( X \) of 2-bits and one output \( Y \) of 4-bits. Show mapping of this signal assignment to gate level. **12 Marks**