1 Introduction to VHDL

VHDL, or **Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language**, was developed in the early 1980s under the VHSIC program initiated by the U.S. Department of Defense. Its primary goal was to provide a standardized way to describe and simulate the behavior of digital circuits before their actual implementation. VHDL became an IEEE standard in 1987 (IEEE 1076) and has since been widely adopted in academia and industry.

VHDL plays a crucial role in embedded systems design due to its ability to describe complex digital logic at various levels of abstraction. This flexibility is particularly significant in the development of hardware for embedded systems, where custom logic designs often need to interface with processors, memory, and I/O devices. Key benefits include:

- **High-Level Abstraction**: VHDL allows designers to model digital circuits behaviorally, structurally, or at a dataflow level, enabling flexibility in design and verification.
- **Simulation Before Implementation**: Designers can verify functionality and performance through simulation, reducing costly hardware errors.
- **Hardware Portability**: VHDL designs can be synthesized for different target devices like FPGAs or ASICs, promoting reusability and adaptability.
- **Concurrent Design**: It supports concurrent operations, aligning well with the parallel nature of digital circuits.

Designing digital systems can be a daunting task, especially when dealing with the intricate interplay of processors, memory, and I/O devices in modern embedded systems. The stakes are high—errors in the design phase can lead to costly revisions in hardware, delaying time-to-market and impacting overall system performance. This is where VHDL emerges as a game-changer, offering designers a powerful toolset to translate complex ideas into reliable hardware solutions with precision and confidence.

Once the design is complete, it undergoes **simulation** and **synthesis** to ensure correctness and readiness for hardware implementation. Simulation involves testing the logical behavior of the design under various input conditions to verify its alignment with the

intended functionality. This step focuses solely on the conceptual logic without considering hardware-specific factors. On the other hand, Synthesis converts the VHDL description into a hardware-specific representation, accounting for timing constraints, physical resources, and other real-world considerations of the target FPGA or PLD. This step ensures the design's feasibility in actual hardware deployment. By combining these strengths, VHDL enables designers to create robust and efficient digital circuits while minimizing errors even before hardware implementation.

1.1 Application of VHDL

VHDL is a powerful hardware description language widely used for designing digital circuits and systems. Its primary application lies in the development of **Application Specific Integrated Circuits (ASICs)**, where it enables designers to describe complex digital systems at a high level of abstraction. The transformation of VHDL code into a gate-level netlist, known as synthesis, is a crucial part of the ASIC design flow. This process automates the mapping of high-level designs into physical hardware components, making VHDL an integral tool for efficient and precise ASIC development.

In addition to ASICs, VHDL is extensively employed in the design of **Field Programmable Gate Arrays (FPGAs)**. However, FPGA design introduces specific challenges. Initially, Boolean equations are derived from the VHDL description, regardless of the target technology. These equations must then be partitioned into the configurable logic blocks (CLBs) of the FPGA, a step more complex than mapping onto an ASIC library due to the fixed architecture of FPGAs. Furthermore, routing the interconnections between CLBs often presents significant constraints, as the available resources for routing can become a bottleneck in FPGA designs. Despite these difficulties, modern synthesis tools are capable of handling complex FPGA designs, though the results are sometimes suboptimal.

For simpler digital systems implemented on **Programmable Logic Devices (PLDs)**, VHDL is less commonly used. The potential for suboptimal synthesis results and the overhead associated with the design process make it less suitable for low-complexity designs where simpler tools and methods might suffice.

Overall, VHDL remains a foundational tool in the realm of digital design, particularly for ASICs and FPGAs. Its ability to facilitate the description, simulation, and synthesis of intricate hardware systems ensures its continued relevance and importance in modern electronic design workflows.

1.2 Basic Structure of a VHDL Program

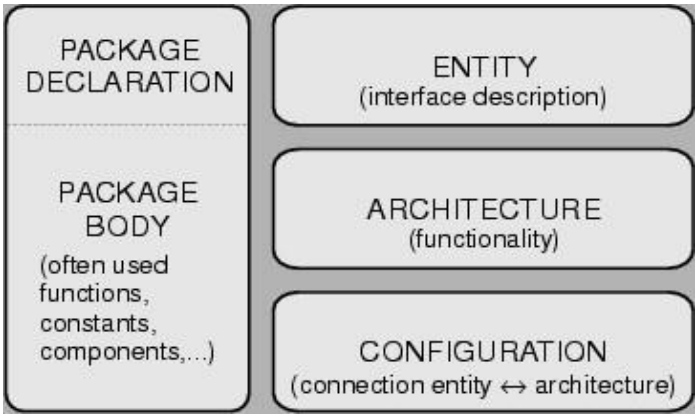
The foundational structure of a VHDL program revolves around the concept of blocks, which act as the basic units of design. These blocks enable designers to describe the functionality, data flow, or structural behavior of a logic circuit with clarity, flexibility, and modularity. In VHDL, digital systems are typically designed as a hierarchy of modules, each representing a different component or function. This modular approach helps break down complex systems into manageable, reusable parts.

The main building blocks of a VHDL design are **Entity**, **Architecture**, **Package**, **Configuration**, and **Library**. Each component has a distinct role in defining and organizing the design. The **Entity** serves as the external interface, specifying the input and output ports that link the component to other parts of the system. In contrast, the **Architecture** defines the internal workings of the entity, describing how the system operates. A digital system often consists of multiple entities, each with its corresponding architecture, working together to form a cohesive design.

Packages in VHDL are used to define global elements such as data types, constants, and functions, which can be shared across multiple entities. This promotes reusability and modularity, ensuring that common elements are defined only once and can be used wherever needed. **Configurations** allow designers to bind entities with their corresponding architectures, offering the flexibility to experiment with different design implementations and configurations. This feature is particularly useful when optimizing designs or exploring multiple architectural options.

Lastly, VHDL designs are typically organized into **Libraries**, which store and manage various design units. Libraries provide a structured and efficient way to compile and access components, ensuring that large designs remain organized and easy to navigate. Collectively, these basic structures offer a powerful framework for creating, simulating,

and synthesizing digital systems, promoting clarity, scalability, and maintainability throughout the design process.



1. Entity Declaration

An **entity declaration** is the fundamental building block of a VHDL design, defining the external interface of a design unit. It outlines the input and output signals that connect the design to its environment, essentially forming a "black box" representation of the hardware module. The entity block specifies the names, types, and directions of the signals that interact with the outside world, offering a clear blueprint for the system's functionality.

Structure of an Entity Declaration

The **entity declaration** begins with the reserved word `entity`, followed by the **entity name**, which must adhere to VHDL naming rules. Identifiers can include letters, numbers, and underscores but must start with an alphabetic character. The entity name is followed by the reserved word `is`, which introduces the port declarations, where input and output signals are defined.

General Syntax of Entity

A typical entity block structure looks like this:

```
entity entity_name is
  port (
    signal_name1, signal_name2   : mode type;
    signal_name3                 : mode type
  );
end entity_name;
```

- **entity_name**: Represents the name of the entity.
- **port**: Declares the interface signals, specifying their modes and data types.
- **mode**: Indicates the direction of signal flow.
- **type**: Defines the data type of the signal.

The entity declaration concludes with the reserved word `end`, optionally followed by the entity name for better readability.

Port Declaration and Modes

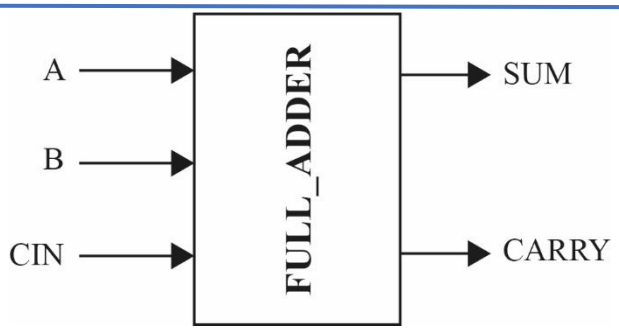
The **port declaration** within an entity block specifies the direction and type of signals that form the interface. Each signal can belong to one of the following modes:

- **in**: Signals that serve as inputs to the design. These signals can only be read and are typically used for receiving data or control inputs.
- **out**: Signals used for outputs from the design. They can only be assigned values within the design and are generally written to drive external devices.
- **inout**: Signals that are bidirectional, allowing both reading and writing. These are commonly used in bus architectures.
- **buffer**: A specialized unidirectional mode allowing read-back capability within the design. Unlike `inout`, it is restricted to a single driver.

Example: Full Adder

The following example illustrates an entity declaration for a **one-bit full adder**. Figure-1 shows the interface of the component. The entity is named `FULL_ADDER`, with input ports `A`, `B`, and `CIN` of data type `BIT`, and output ports `SUM` and `CARRY`, also of type `BIT`. The corresponding VHDL description is shown below:

```
entity FULL_ADDER is
  port (
    A, B, CIN : in BIT;
    SUM, CARRY : out BIT
  );
end FULL_ADDER;
```



This entity defines the inputs and outputs for a full adder circuit, providing a clear and structured representation of its interface. The `in` mode ensures that `A`, `B`, and `CIN` can only be read, while the `out` mode specifies that `SUM` and `CARRY` are outputs from the entity.

2. Architecture Declaration

In VHDL, the **architecture block** provides an "internal" view of an entity, defining how it operates. While the entity defines the external interface, the architecture specifies the internal behavior or structure. Notably, an entity can have more than one architecture, offering flexibility in design representation. An architecture defines the relationships between the inputs and outputs of an entity. These relationships can be described in different styles:

- **Behavioral Style**: Describes the logic behavior of the entity using processes or Boolean expressions.
- **Dataflow Style**: Emphasizes the flow of data between inputs and outputs.
- **Structural Style**: Specifies the components and their interconnections that form the entity.

The architecture consists of two main sections:

1. **Declaration Section**: Used to declare signals, types, constants, components, and subprograms.
2. **Concurrent Statements Section**: Defines the operation of the entity, written after the begin keyword.

General Syntax of Architecture

The architecture block follows this general syntax:

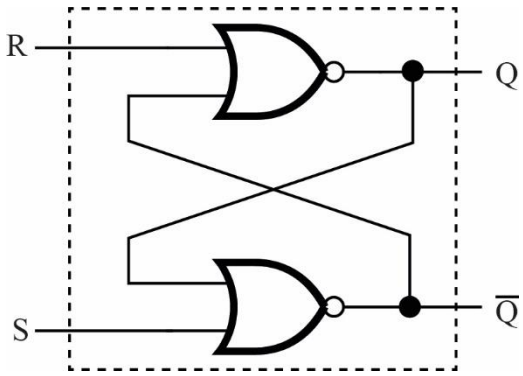
```
architecture architecture_name of entity_name is
```

```
   {architecture_declarative_part}
begin
   {concurrent_statement}
end [architecture_name];
```

- **architecture_name**: The name of the architecture.
- **entity_name**: The name of the associated entity.
- **architecture_declarative_part**: Optional section for declarations.
- **concurrent_statement**: Defines the logic or structure of the entity.

Example: Set/Reset NOR Latch

Figure-1 illustrates the interface of a set/reset NOR latch. Below is the VHDL description for the latch, showcasing both its entity and architecture blocks.




```
library ieee;  -- Importing the IEEE standard library
use ieee.std_logic_1164.all;  -- Using the std_logic package from IEEE for
logic types

-- Entity block: Defines the inputs and outputs of the latch
entity latch is
   port (
      s, r : in std_logic;  -- Inputs: s (set), r (reset)
      q, nq : out std_logic  -- Outputs: q (output), nq (negated output)
   );
end latch;

-- Architecture block: Describes the behavior of the latch
architecture flipflop of latch is
begin
   -- Assignment statements: Define how outputs are determined by inputs
   q <= r nor nq;  -- Output q is the result of the NOR operation between r
and nq
   nq <= s nor q;  -- Output nq is the result of the NOR operation between
s and q
end flipflop;
```

A. Library and Use Statements:

- `library ieee;` and `use ieee.std_logic_1164.all;` are used to include the IEEE standard logic library. This library gives us predefined logic types and functions such as `std_logic` and `std_logic_vector`, which are necessary to define digital circuits in VHDL.

B. Entity Block:

- The `entity latch` defines the interface for the NOR latch. It has two input signals (`s` and `r`) and two output signals (`q` and `nq`), all of type `std_logic`.

C. Architecture Block:

- The `architecture flipflop` defines how the latch behaves. It is linked to the entity `latch`.
- Inside the architecture, two signal assignment statements describe how the latch works using the NOR operation.
  - `q <= r nor nq;` means the output `q` is the result of applying the NOR operation to the signals `r` and `nq`.
  - `nq <= s nor q;` means the output `nq` is the result of applying the NOR operation to the signals `s` and `q`.
- The `<=` operator is used to assign values to signals. This is different from the `:=` operator, which is used for assigning values to variables or generics.

2 Basic Language Elements

VHDL is built upon fundamental language elements that provide the structure and rules for describing hardware. These basic elements, such as identifiers, data objects, and data types, form the backbone of any VHDL design. Understanding them is essential for writing efficient and accurate code, as they define how information is represented, stored, and manipulated within a digital circuit.

2.1 Identifiers

Identifiers are names used to represent various elements in VHDL, such as signals, variables, entities, architectures, or processes. They are essential for labeling and organizing the components of a VHDL design.

Rules for Identifiers:

1. **Starting Character:**
   - Identifiers must begin with a letter (A–Z or a–z).
2. **Allowed Characters:**
   - Identifiers can include letters, digits (0–9), and underscores (_).
3. **Restrictions on Underscores:**
   - An identifier cannot end with an underscore.
   - Consecutive underscores within an identifier are not allowed.
4. **Case Insensitivity:**
   - VHDL treats uppercase and lowercase letters as equivalent (e.g., `Count` and `count` are considered identical).
5. **Comments for Documentation:**
   - Comments in VHDL start with `--` and continue until the end of the line.
   - Comments can be placed anywhere in the code to document or explain functionality.

Examples:

Allowed	Not Allowed
Signal1	_start (starts with an underscore)
data_buffer	signal__1 (consecutive underscores)
clkOut	end_ (ends with an underscore)
temperature_sensor_reading	_dataBuffer (starts with an underscore)
reset_signal_active_low	clk__out (consecutive underscores)
state_machine_transition_signal	finish_ (ends with an underscore)
UART_Transmission_Flag	counter__value (consecutive underscores)
MemoryBlock32	_Address (starts with an underscore)
SystemClockDivider	value__reset (consecutive underscores)

2.2 Data Objects in VHDL

In VHDL, data objects hold values of a specified type and are created using object declarations. These data objects play a crucial role in storing and managing data throughout the simulation and hardware design. There are four primary classes of data objects in VHDL: Constant, Variable, Signal, and File. Each of these data object types serves a different purpose and behaves according to the rules of VHDL.

1. **Constant**

A **constant** is a data object that holds a fixed value that cannot change during the simulation or execution of a design. Constants are useful for representing values that are intended to remain unchanged throughout the entire design process, such as mathematical constants, configuration parameters, or hardware configuration settings.

Declaration Syntax:

constant <name>: <type> := <value>;

- `<name>`: The identifier for the constant.
- `<type>`: The data type of the constant (e.g., integer, real, boolean).
- `<value>`: The fixed value assigned to the constant at the time of declaration.

Example:

```
constant Pi: real:= 3.14159;          -- A constant representing the value of Pi
constant MaxValue: integer:= 255;     -- A constant representing the maximum value
```

2. Variable

A **variable** is a data object that holds a value, which can be changed during the execution of a process or procedure. Variables are typically used within processes or functions where temporary values are needed for intermediate calculations or logic.

Declaration Syntax:

variable <name>: <type> := <initial_value>;

- `<name>`: The identifier for the variable.
- `<type>`: The type of the variable (e.g., integer, boolean, std_logic).
- `<initial_value>`: The initial value assigned to the variable when it is declared.

Example:

```
variable Count: integer := 0;  -- A variable initialized to zero
variable TempData: real := 0.0;  -- A real-valued variable initialized to 0.0
```

3. Signal

A **signal** is a data object that represents a value which can change over time. Signals are used for communication between different processes or between different parts of a design. They are similar to wires or buses in hardware and can carry information across different parts of a VHDL design.

Declaration Syntax:

```
signal <name>: <type> := <initial_value>;
```

- `<name>`: The identifier for the signal.
- `<type>`: The type of the signal (e.g., std_logic, integer).
- `<initial_value>`: The initial value of the signal, which is used when simulation starts.

Example:

```
signal clk: std_logic := '0';          -- A signal for a clock with an initial value
signal Reset: std_logic := '1';        -- A reset signal initialized to '1'
```

4. File

A **file** is a data object that allows reading from or writing to an external file during simulation. Files are typically used when you need to log simulation results, read input data, or generate output data for further analysis.

Declaration Syntax:

```
file <name>: <type> is <external_file>;
```

- `<name>`: The identifier for the file.
- `<type>`: The type of data in the file (e.g., text, integer).
- `<external_file>`: The actual external file (e.g., "data.txt") that the simulation will read or write to.

Example:

```
file DataFile: text is "data.txt";      -- A file linked to "data.txt"
file LogFile: text is "log.txt";        -- A file used to store simulation logs
```

2.3 Data Types

In VHDL, data types are essential because they define the nature and behavior of the values that can be assigned to variables, signals, and other elements within a design. They specify the type of data that can be represented, manipulated, and operated on within a VHDL design. Whether you're designing a simple digital circuit or a complex system, understanding and choosing the correct data type is crucial for ensuring that the system works as expected and efficiently.

VHDL supports two broad categories of data types: **Predefined Data Types** and **User-Defined Data Types**. Predefined data types are those that come built into the VHDL language and are commonly used for general-purpose operations. These include types such as integers, Booleans, characters, and floating-point numbers. On the other hand, **User-Defined Data Types** offer more flexibility by allowing designers to define their own data types based on the requirements of the system. This includes the ability to create custom types like enumerations, arrays, records, and more.

2.3.1 Predefined Data Types

In VHDL, predefined data types are those that are built into the language and are commonly used across a wide range of digital designs. Predefined data types in VHDL are declared in the **IEEE standard libraries**. These libraries are commonly imported into a VHDL design to access predefined types and functions. The most commonly used library is ieee.std_logic_1164, which includes the `std_logic` type for representing individual logic signals and the `std_logic_vector` type for handling arrays of logic signals. For arithmetic operations, the **`ieee.numeric_std`** library is typically used, offering the `signed` and `unsigned` types for working with signed and unsigned integers. The **`ieee.std_logic_arith`** library, an older standard, also provides the `signed` and `unsigned` types for similar operations. Additionally, the **`ieee.math_real`** library provides the `real` type, which is used for representing floating-point numbers. These libraries and their corresponding data types are fundamental for creating digital circuits and performing various arithmetic computations in VHDL.

The table below provides an overview of the most commonly used predefined data types in VHDL.

S.No.	Data Type	Possible Range
1	BIT	'0', '1'
2	BOOLEAN	FALSE, TRUE
3	INTEGER	-2,147,483,648 to 2,147,483,647
4	REAL	-1.0E38 to 1.0E38
5	TIME	Depends on time unit: fs, ps, ns, us, ms, sec, min, hr
6	BIT_VECTOR	Array of BIT values
7	STD_LOGIC	'0', '1', 'Z', 'W', 'L', 'H', 'U', 'X', '—'
8	STD_LOGIC_VECTOR	Array of STD_LOGIC values
9	SIGNED	Depends on the size of the vector
10	UNSIGNED	Depends on the size of the vector
11	CHARACTER	Any valid character in the ASCII set

In VHDL, the predefined data type `std_logic` represents a single binary value, and it can take various values that indicate different logical states. These values are defined as part of the `std_logic` type, commonly used in digital circuit design and simulation. The values are:

- **'0'**: Represents a logic low, or binary 0.
- **'1'**: Represents a logic high, or binary 1.
- **'Z'**: High impedance state, often used for representing unconnected or floating signals.
- **'W'**: Weak unknown state, used when the logic value cannot be determined.
- **'L'**: Weak low, representing a value that is weakly driven to logic 0.
- **'H'**: Weak high, representing a value that is weakly driven to logic 1.
- **'U'**: Uninitialized, representing a signal that has not been assigned a value yet.
- **'X'**: Unknown state, used when the value of a signal cannot be determined due to conflicting drivers.
- **'—'**: Don't care state, used when the value of a signal does not matter or is irrelevant in a given context.

These values are essential for modeling and simulating realistic digital circuits in VHDL, especially when handling tri-state logic, initialization issues, or conflict resolution.

2.3.2 User-Defined Data Types

In VHDL, user-defined data types provide the flexibility to design custom data structures that fit specific requirements of the digital system being modeled. These data types are defined by the designer to represent more complex structures than the predefined types offered by VHDL. By creating user-defined types, a designer can model and manipulate data in ways that better suit the needs of a particular application, providing more control and enabling more efficient and readable code.

User-defined data types in VHDL are divided into four main categories, each offering unique ways to represent and manipulate data. These categories are **Scalar Type**, **Composite Type**, **Access Type and File Type**

1. Scalar Types (User-Defined)

Scalar types in VHDL represent single values or variables that store only one value at a time. They are fundamental in digital system design because they allow for simple, clear representation of data that can be used in a wide variety of applications. Scalar types are essential for modeling data that does not need to be broken down into more complex structures, such as arrays or records. They offer simplicity and efficiency when you need to define single-value variables in your design. Scalar types in VHDL can be further divided into four subcategories viz Enumeration type, Integer Type, Physical Type, and Floating-Point Type.

Enumeration Types

Enumeration types allow the designer to define a set of named values, which are treated as distinct elements in the design. These values are often used to represent states, modes, or other types of well-defined categories in a system. An enumeration type assigns a symbolic name to each value, which enhances readability and clarity in VHDL code.

Example:

```
type TrafficLight is (Red, Yellow, Green);
signal light: TrafficLight;
```

In this example, an enumeration type `TrafficLight` is created with three possible values: `Red`, `Yellow`, and `Green`. The signal `light` can then be assigned one of these values to represent the state of a traffic light.

Integer Types

Integer types are used to represent whole numbers. They can be defined within a specific range or with an unlimited range. Integer types are essential for counting, indexing, and performing arithmetic operations that require whole numbers.

Example:

```
signal Count: integer range 0 to 100;
```

Here, an integer signal `Count` is defined with a range of 0 to 100. This signal can hold any integer value between 0 and 100, inclusive.

Physical Types

Physical types are used to represent quantities that have units of measurement, such as time, voltage, or distance. These types allow for the creation of custom data types that include both the value and the units, making it easier to work with real-world quantities in simulations.

Example:

```
type Voltage is new real range 0.0 to 5.0;
signal v: Voltage;
```

In this example, a physical type `Voltage` is defined as a new real type, with a range of values from 0.0 to 5.0. This type could be used to represent the voltage level in a system.

Floating-Point Types

Floating-point types are used for representing real numbers, especially when precision is required for numbers with decimal points. These types are useful for modeling continuous values, such as temperature, pressure, or any other quantity that may require a fractional component.

Example:

```
signal Temperature: real := 37.5;
```

In this example, a signal `Temperature` is declared with a `real` type and initialized to 37.5. This allows for the storage and manipulation of floating-point numbers.

2. Composite Types (User-Defined)

Composite types in VHDL allow for the grouping of multiple values into a single data structure, enabling efficient representation and manipulation of complex data. These types are especially useful when working with collections of data or when logically related values need to be handled as a unit. By using composite types, designers can improve the clarity, modularity, and organization of their VHDL code, making it easier to model and simulate

real-world systems. Composite types are further subcategorized as Array Type and Record Type.

Array Types

Array types in VHDL are used to represent collections of values that are of the same type. Arrays are particularly useful for handling data in sequential or indexed forms, such as in digital signal processing, memory structures, or look-up tables. Arrays can be one-dimensional (vectors) or multi-dimensional, depending on the requirements of the design. Arrays are ideal for representing vectors of signals (e.g., data buses), tables, or matrices in VHDL.

Example:

```
type IntArray is array (0 to 7) of integer;
signal Data: IntArray := (0, 1, 2, 3, 4, 5, 6, 7);
```

In this example, `IntArray` is defined as an array with eight elements, where each element is an integer. The signal `Data` is an instance of this array, initialized with values from 0 to 7.

Record Types

Record types allow grouping of values of different types into a single structure, enabling the representation of complex entities. Each value in a record is referred to as a field, and fields can have different data types. Records are particularly useful for modeling real-world entities with multiple attributes, such as devices, configurations, or people. Records are ideal for organizing related but varied data, such as configuration settings, control structures, or hardware interfaces.

Example:

```
type Person is record
  Name: string(1 to 20);
  Age: integer;
end record;
signal John: Person := (Name => "John Doe", Age => 30);
```

In this example, the record type `Person` is defined with two fields: `Name` (a string of 20 characters) and `Age` (an integer). The signal `John` is declared as a `Person` type and initialized with specific values.

3. Access Types (User-Defined)

Access types in VHDL are analogous to pointers in traditional programming languages, enabling dynamic referencing of memory locations during simulation. These types allow the creation of variables that store references to objects of a specified type rather than the actual data. Access types are particularly useful for dynamically managing data, such as creating linked data structures, trees, or other advanced data models in VHDL simulations. However, they are typically used in simulation contexts rather than synthesis, as hardware does not naturally support dynamic memory allocation.

Example

```
type IntPointer is access integer;
variable Ptr: IntPointer;
begin
  Ptr := new integer'(42); -- Allocates memory and stores the value 42
  ...
  deallocate Ptr; -- Deallocates the memory
```

In this example:

- `IntPointer` is declared as an access type for integers.
- `Ptr` is a variable of type `IntPointer` that points to dynamically allocated memory holding an integer value (`42` in this case).
- The memory can be deallocated when no longer needed to prevent memory leaks.

4. File Types (User-Defined)

File types in VHDL enable interaction with external files during simulation. They are used to read data from or write data to files, facilitating tasks such as logging simulation results, initializing values, or analyzing data. File types are not synthesizable, as they are designed exclusively for simulation purposes. This feature is particularly useful for verifying designs and documenting the behavior of a VHDL model.

Example

```
file LogFile: text is "log.txt"; -- Declare a text file
```

```
variable LineBuffer: line;      -- Variable to hold a line of text
begin
  -- Write to the file
  write(LineBuffer, String'("Simulation started"));
  writeline(LogFile, LineBuffer);

  -- Read from the file
  readline(LogFile, LineBuffer);
  read(LineBuffer, VariableName); -- Assign read value to a variable
```

In this example:

- `LogFile` is declared as a file of type `text` and is associated with the external file `"log.txt"`.
- `LineBuffer` is a temporary buffer to hold lines of text for writing or reading.
- Data is written to the file using the `write` and `writeline` procedures. Similarly, the `read` and `readline` procedures are used to read data.

2.4 Operators

Operators in VHDL allow designers to perform various operations on data objects such as signals, variables, and constants. The predefined operators in VHDL are classified into the following categories.

- Logical Operators
- Relational Operators
- Shift Operators
- Arithmetic operators
- Miscellaneous operators

1. Logical Operators

Logical operators perform bit-wise operations on Boolean or bit-level data types like `std_logic` and `bit_vector`.

Operator	Operation	Example
`and`	Logical AND	`Result <= A and B;`
`or`	Logical OR	`Result <= A or B;`
`nand`	Logical NAND	`Result <= A nand B;`

nor	Logical NOR	`Result <= A nor B;`
xor	Logical XOR	`Result <= A xor B;`
xnor	Logical XNOR	`Result <= A xnor B;`
not	Logical NOT	`Result <= not A;`

Example:

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

entity LogicalOps is
    Port ( A, B : in std_logic;
           Result : out std_logic);
end LogicalOps;

architecture Behavioral of LogicalOps is
begin
    process(A, B)
    begin
        Result <= A and B;
    end process;
end Behavioral;
```

2. Relational Operators

Relational operators are used to compare two operands. They return a Boolean value (`TRUE` or `FALSE`).

Operator	Operation	Example
=	Equal to	`Flag <= A = B;`
/=	Not equal to	`Flag <= A /= B;`
<	Less than	`Flag <= A < B;`
>	Greater than	`Flag <= A > B;`
<=	Less than or equal to	`Flag <= A <= B;`
>=	Greater than or equal to	`Flag <= A >= B;`

Example:

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

```
entity RelationalOps is
    Port ( A, B : in integer;
           Flag : out boolean);
end RelationalOps;

architecture Behavioral of RelationalOps is
begin
    process(A, B)
    begin
        Flag <= A > B;
    end process;
end Behavioral;
```

3. Shift Operators

Shift operators are used to shift bits left or right in a vector.

Operator	Operation	Example
sll	**Shift left logical**	**Result <= A sll 1;**
srl	**Shift right logical**	**Result <= A srl 1;**
sla	**Shift left arithmetic**	**Result <= A sla 1;**
sra	**Shift right arithmetic**	**Result <= A sra 1;**
rol	**Rotate left**	**Result <= A rol 1;**
ror	**Rotate right**	**Result <= A ror 1;**

Example

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

entity ShiftExample is
    Port ( A : in std_logic_vector(7 downto 0);  -- 8-bit input A
           LShift : out std_logic_vector(7 downto 0);  -- Logical
left shift result
           RShift : out std_logic_vector(7 downto 0);  -- Logical
right shift result
           ASR : out std_logic_vector(7 downto 0));  -- Arithmetic
right shift result
end ShiftExample;

architecture Behavioral of ShiftExample is
begin
    -- Logical left shift
    LShift <= A sll 1;  -- Shift A to the left by 1 bit

    -- Logical right shift
    RShift <= A srl 1;  -- Shift A to the right by 1 bit

    -- Arithmetic right shift
    ASR <= A sra 1;  -- Shift A to the right by 1 bit, preserving the
sign bit
```

```
end Behavioral;
```

4. Arithmetic Operators

Arithmetic operators are used to perform basic mathematical operations. They work on numerical data types such as integers, real numbers, and bit vectors.

Operator	Operation	Example
+	Addition	Sum <= A + B;
-	Subtraction	Diff <= A - B;
*	Multiplication	Product <= A * B;
/	Division	Quotient <= A / B;
mod	Modulus	Rem <= A mod B;
rem	Remainder	Rem <= A rem B;

Example:

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

entity ArithmeticOps is
    Port ( A, B : in integer;
           Sum, Diff : out integer);
end ArithmeticOps;

architecture Behavioral of ArithmeticOps is
begin
    process(A, B)
    begin
        Sum <= A + B;
        Diff <= A - B;
    end process;
end Behavioral;
```

5. Concatenation Operators

Concatenation operators combine multiple signals or bit values into a single vector.

Operator	Operation	Example
&	Concatenation	Vector <= A & B;

Example:

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

entity ConcatenationOps is
```

```
    Port ( A : in std_logic;
           B : in std_logic_vector(3 downto 0);
           Result : out std_logic_vector(4 downto 0));
end ConcatenationOps;

architecture Behavioral of ConcatenationOps is
begin
    process(A, B)
    begin
        Result <= A & B; -- Concatenates A and B
    end process;
end Behavioral;
```

6. Miscellaneous Operators

These operators perform type conversions, conditional evaluations, or null operations.

Operator	Operation	Example
'	Attributes (e.g., width)	`Width <= A'length;`
? :	Conditional operator	`Result <= (A > B) ? A : B;`

3 Statements in VHDL

In VHDL, **statements** are the core elements used to describe the functionality and structure of digital circuits. They define how signals interact, how data is processed, and how hardware components operate. VHDL statements provide the necessary instructions for simulating and synthesizing hardware designs, enabling designers to translate circuit behavior into code. These statements are categorized into **Sequential Statements**, which execute in a defined order, and **Concurrent Statements**, which operate simultaneously, reflecting the parallel nature of digital hardware.

3.1 Sequential Statements

Sequential statements in VHDL are executed one after another, in the exact order they are written. These statements are used within specific constructs like **process statements**, **functions**, or **procedures**, and are essential for describing sequential logic. Sequential logic refers to circuits where the output depends not only on the current inputs but also on the order of operations and the history of inputs, often synchronized with a clock signal.

To function correctly, sequential statements must be placed within an appropriate construct that ensures they execute sequentially. Among these, the **process statement** is the primary

container for sequential operations in VHDL. This makes process statements indispensable for modeling timing-dependent circuits such as state machines, counters, and other clock-driven systems.

3.2 Process Statement

The `process` statement in VHDL is a key construct for describing sequential logic. It allows you to group a series of sequential statements and specify conditions under which they are executed. A process in VHDL can be used to model behavior that depends on the state of signals and variables, often in response to clock edges or changes in inputs. It is primarily used for describing circuits that involve timing, such as state machines, counters, and other clocked circuits.

Syntax

```
label: process (sensitivity-list)
  { declarations }
begin
  { sequential-statements }
end process label;
```

- **label:** The optional name given to the process. This label helps identify the process in a design, making it easier to reference, especially in complex systems.
- **sensitivity-list:** This defines the signals or variables that, when changed, will trigger the process to execute. It determines what the process is sensitive to.
- **declarations:** This section is optional and can contain declarations of variables, constants, or types used inside the process.
- **sequential-statements:** These are the statements that define the operations performed within the process. They will execute sequentially, one after another.

Sensitivity List:

The **sensitivity list** defines which signals or variables the process is "sensitive" to, meaning the process will be triggered and execute its contents whenever any of these signals change. If no sensitivity list is provided, the process runs continuously, which is typically not recommended for most designs.

A process without a sensitivity list will not be responsive to signal changes, so it is important to explicitly define which signals should cause the process to execute. If the process is only triggered on specific events, such as clock edges, the sensitivity list should contain only the signals that are relevant for that specific event.

Example

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;  -- Standard logic types
use IEEE.STD_LOGIC_ARITH.ALL; -- Arithmetic operations
use IEEE.STD_LOGIC_UNSIGNED.ALL; -- For unsigned operations

entity counter is
    Port ( clk    : in STD_LOGIC;    -- Clock input
           reset  : in STD_LOGIC;    -- Reset input
           count  : out STD_LOGIC_VECTOR (3 downto 0) ); -- 4-bit counter output
end counter;

architecture Behavioral of counter is
begin
    -- The process is triggered by changes in clk or reset
    counter_process: process (clk, reset)
    begin
        if reset = '1' then
            count <= "0000";  -- Reset counter to 0000 if reset is high
        elsif rising_edge(clk) then
            count <= count + 1;  -- Increment counter on rising edge of clk
        end if;
    end process counter_process;
end Behavioral;
```

Sequential Statements in a Process

After defining a process in VHDL, you can use various sequential statements to describe the logic that should execute within that process. These statements follow a specific order of execution and are essential for creating behaviors where the output depends on the history of inputs or the sequence of events. Examples of sequential statements include **if-else**, **case**, **loop**, and others. These statements allow you to specify conditions, iterate over values, or implement complex decision-making processes. Each statement serves a unique purpose, such as handling

conditions or repeating operations, and they are all executed in the order they appear within the process. The flexibility of these statements enables you to model a wide range of sequential behavior in hardware, from simple operations to intricate state machine logic.

1. If-Else Statement

The if-else statement is used to select one of several possible actions based on a condition. It is typically used to model conditional behavior in sequential logic. It executes sequentially until the first true condition is found; then the set of sequential statements associated with this condition is executed.

Syntax:

```
if boolean-expression then
   sequential_statement;
[elsif boolean-expression then   -- elsif clause; if statement can
   sequential_statement;]        -- have 0 or more elsif clauses.
Else                             -- else clause(optional)
   sequential_statement;
end if;
```

Example:

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

entity IfElseExample is
    Port ( Signal_in : in std_logic;  -- Input signal
           Signal_out : out std_logic); -- Output signal
end IfElseExample;

architecture Behavioral of IfElseExample is
begin
    -- The process is sensitive to changes in Signal_in
    process(Signal_in)  -- Sensitivity list includes Signal_in
    begin
        if Signal_in = '1' then
            Signal_out <= '0';  -- If Signal_in is '1', set Signal_out to '0'
```

```
        else
            Signal_out <= '1';  -- Otherwise, set Signal_out to '1'
        end if;
    end process;
end Behavioral;
```

2. Case Statement

The case statement is used to execute different actions based on the value of an expression. It provides a way to model decision-making where the input is compared to a set of possible choices, and the corresponding action is executed. It is particularly useful when there are multiple conditions based on a single variable or expression.

Syntax:

```
case expression is
    when choice1 =>
        sequential_statement;
    when choice2 =>
        sequential_statement;
    when others =>
        sequential_statement;  -- Optional, for all other cases
end case;
```

Example:

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

entity CaseExample is
    Port ( Signal_in : in std_logic_vector(1 downto 0);  -- 2-bit input signal
           Signal_out : out std_logic); -- Output signal
end CaseExample;

architecture Behavioral of CaseExample is
begin
    -- The process is sensitive to changes in Signal_in
    process(Signal_in)
    begin
        case Signal_in is
            when "00" =>
```

```
            Signal_out <= '0';  -- If Signal_in is 00, set Signal_out to '0'
        when "01" =>
            Signal_out <= '1';  -- If Signal_in is 01, set Signal_out to '1'
        when others =>
            Signal_out <= 'Z';  -- If Signal_in is anything else, set Signal_out to 'Z'
    end case;
  end process;
end Behavioral;
```

3. Loop Statement

The loop statement is used to execute a set of sequential statements repeatedly until a specific condition is met. Loops are helpful for modeling repetitive behavior or for performing tasks like counting, state transitions, or handling multiple conditions. In VHDL, there are different types of loops such as **for loops**, **while loops**, and **loop** (infinite loop). The loop runs until the specified exit condition is satisfied.

Syntax:

```
loop
   sequential_statement;
   exit when condition;  -- Optional exit condition to stop the loop
end loop;
```

Example:

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

entity LoopExample is
    Port ( clk : in std_logic;  -- Clock signal
           reset : in std_logic;  -- Reset signal
           counter : out integer range 0 to 10); -- Counter output
end LoopExample;

architecture Behavioral of LoopExample is
    signal count : integer range 0 to 10 := 0;  -- Internal counter
begin
    -- The process is sensitive to clock and reset
    process(clk, reset)
    begin
```

```
        if reset = '1' then
            count <= 0;  -- Reset the counter to 0 when reset signal is '1'
        elsif rising_edge(clk) then
            loop
                if count < 10 then
                    count <= count + 1;  -- Increment the counter on each clock cycle
                else
                    exit;  -- Exit the loop when count reaches 10
                end if;
            end loop;
        end if;
    end process;

    -- Output the value of the counter
    counter <= count;
end Behavioral;
```

4. Exit Statement

The **exit** statement is used to immediately terminate a loop or a process in VHDL. It is particularly useful when the loop has a condition that can end the iteration before reaching its natural conclusion. The **exit** statement can be used with any type of loop (for loop, while loop, or infinite loop) to break out of the loop when a specific condition is met.

Syntax:

```
loop
   sequential_statement;
   exit when condition;  -- Exit the loop when the condition is true
end loop;
```

Example:

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

entity ExitExample is
    Port ( clk : in std_logic;       -- Clock signal
           reset : in std_logic;       -- Reset signal
           done : out std_logic);    -- Signal to indicate completion
end ExitExample;

architecture Behavioral of ExitExample is
    signal count : integer range 0 to 5 := 0;  -- Counter variable
```

```
begin
    -- The process is sensitive to clock and reset
    process(clk, reset)
    begin
        if reset = '1' then
            count <= 0;  -- Reset the counter
            done <= '0';  -- Reset the done signal
        elsif rising_edge(clk) then
            loop
                if count < 5 then
                    count <= count + 1;  -- Increment counter
                else
                    done <= '1';  -- Set done to '1' when count reaches 5
                    exit;  -- Exit the loop once the count reaches 5
                end if;
            end loop;
        end if;
    end process;
end Behavioral;
```

5. Next Statement

The **next** statement is used to immediately skip the remaining iteration of a loop and proceed to the next iteration. It is useful when a condition is met during an iteration, and you want to bypass the rest of the loop body and start the next iteration. The **next** statement does not terminate the loop, but rather just skips the remaining sequential statements inside the loop for the current iteration.

Syntax:

```
loop
    sequential_statement;
    next when condition;  -- Skip the remaining statements in the loop if the condition is true
end loop;
```

Example:

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

entity NextExample is
    Port ( clk : in std_logic;  -- Clock signal
           reset : in std_logic;  -- Reset signal
           result : out std_logic);  -- Output signal
end NextExample;
```

```
architecture Behavioral of NextExample is
   signal count : integer range 0 to 4 := 0;  -- Counter variable
begin
   -- The process is sensitive to clock and reset
   process(clk, reset)
   begin
      if reset = '1' then
         count <= 0;  -- Reset the counter
         result <= '0';  -- Reset the result signal
      elsif rising_edge(clk) then
         loop
            if count = 2 then
               count <= count + 1;  -- Skip this iteration if count equals 2
               next;  -- Skip the rest of the loop and continue with the next iteration
            end if;

            if count = 4 then
               result <= '1';  -- Set result to '1' when count reaches 4
               exit;  -- Exit the loop once count reaches 4
            end if;

            count <= count + 1;  -- Increment counter
         end loop;
      end if;
   end process;
end Behavioral;
```

6. Assertion Statement

The **assertion** statement is used to check whether a given condition is true or false during simulation. If the condition is true, the simulation continues without any disruption. If the condition is false, an error message can be generated to indicate a problem, helping to detect issues in the design early. Assertions are useful for verifying that certain conditions hold true during simulation and can be used for debugging and validation.

Syntax:

```
assert condition
  report "message" severity level;
```

- **condition**: The Boolean expression that is evaluated.
- **message**: The message displayed when the assertion fails.
- **severity level**: The severity of the assertion failure (e.g., **warning**, **error**, **failure**).

Example:

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

entity AssertionExample is
    Port ( clk : in std_logic;   -- Clock signal
           reset : in std_logic; -- Reset signal
           signal_in : in std_logic; -- Input signal
           result : out std_logic); -- Output signal
end AssertionExample;

architecture Behavioral of AssertionExample is
begin
    -- The process is sensitive to clock and reset
    process(clk, reset)
    begin
        if reset = '1' then
            result <= '0';  -- Reset the output signal
        elsif rising_edge(clk) then
            -- Assertion to check if signal_in is not 'Z' (high impedance)
            assert signal_in /= 'Z'
                report "Error: signal_in should not be 'Z'"
                severity error;  -- Generate an error message if condition is false

            -- Normal behavior: assign result based on signal_in
            if signal_in = '1' then
                result <= '1';  -- If signal_in is '1', set result to '1'
            else
                result <= '0';  -- Otherwise, set result to '0'
            end if;
        end if;
    end process;
end Behavioral;
```

7. Report Statement

The **report** statement in VHDL is used to display messages during simulation. These messages can provide valuable information to the designer, such as errors, warnings, or other important details. The report statement is commonly used for debugging and logging purposes, as it allows the designer to display custom messages when specific conditions are met. It is typically used in combination with **assertion** statements but can also be used independently to print messages at various points in the simulation.

Syntax:

report "message" severity level;

- **message**: The string message that is displayed.
- **severity level**: The severity level of the message. Common levels include **note**, **warning**, **error**, and **failure**.

Example:

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

entity ReportExample is
    Port ( clk : in std_logic;   -- Clock signal
           reset : in std_logic; -- Reset signal
           signal_in : in std_logic; -- Input signal
           result : out std_logic); -- Output signal
end ReportExample;

architecture Behavioral of ReportExample is
begin
    -- The process is sensitive to clock and reset
    process(clk, reset)
    begin
        if reset = '1' then
            result <= '0';  -- Reset the output signal
            report "Reset is active, output is set to 0" severity note; -- Display a message on reset
        elsif rising_edge(clk) then
            if signal_in = '1' then
                result <= '1';  -- If signal_in is '1', set result to '1'
                report "Signal is high, result set to 1" severity note;  -- Display message when
signal_in is high
            else
                result <= '0';  -- Otherwise, set result to '0'
                report "Signal is low, result set to 0" severity note;  -- Display message when
signal_in is low
            end if;
        end if;
    end process;
end Behavioral;
```

8. Null Statement

The **null** statement in VHDL is a statement that does nothing. It is essentially a placeholder or a no-op (no operation). It is used when a statement is syntactically required, but no action is necessary in that particular case. The **null** statement is often used in conditional blocks or loops where no action needs to be taken under certain conditions, but the logic still requires a valid statement in place. This helps avoid compilation errors or logic mistakes when no other action is needed.

Syntax:

```
null;
```

Example:

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

entity NullExample is
    Port ( clk : in std_logic;   -- Clock signal
           reset : in std_logic; -- Reset signal
           signal_in : in std_logic; -- Input signal
           result : out std_logic); -- Output signal
end NullExample;

architecture Behavioral of NullExample is
begin
    -- The process is sensitive to clock and reset
    process(clk, reset)
    begin
        if reset = '1' then
            result <= '0';  -- Reset the output signal
        elsif rising_edge(clk) then
            if signal_in = '1' then
                result <= '1';  -- Set result to 1 when signal_in is '1'
            else
                null;  -- Do nothing when signal_in is '0'
            end if;
        end if;
    end process;
end Behavioral;
```

3.3 Concurrent Statements in VHDL

Concurrent statements in VHDL are used to describe hardware behavior that can operate simultaneously or independently of other parts of the design. Unlike sequential statements, which execute in a defined order, concurrent statements are executed concurrently, reflecting the parallel nature of hardware. These statements are typically used outside processes and represent components like signal assignments, component instantiations, and block declarations. Concurrent statements are fundamental for modeling real-world hardware circuits, where different operations can occur at the same time. Key examples of concurrent statements include **signal assignment**, **component instantiation**, and **generate statements**, all of which define hardware components that run in parallel.

Process

1. Simple Signal Assignment

The **simple signal assignment** statement in VHDL is the most basic form of signal assignment. It assigns a value or an expression directly to a signal. This assignment is concurrent, meaning it happens outside of any process or sequential block and is continuously monitored and executed.

Syntax:

```
signal_name <= expression;
```

- `signal_name`: The signal that will be assigned a value.
- `expression`: The value or logic operation to be assigned to the signal.

Example:

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

entity SimpleSignalAssignment is
   Port ( A : in std_logic;      -- Input signal
          B : in std_logic;      -- Input signal
          Y : out std_logic);    -- Output signal
```

```
end SimpleSignalAssignment;

architecture Behavioral of SimpleSignalAssignment is
begin
   -- Simple concurrent signal assignment
   Y <= A and B;  -- Output is the logical AND of A and B
end Behavioral;
```

2. When Statement (Conditional Signal Assignment)

The when statement is used for conditional signal assignment in a concurrent context. It provides a compact way to assign values to signals based on conditions. Unlike sequential statements, the when statement operates concurrently with other processes and assignments.

Syntax:

```
signal_name <= expression when condition else alternative_expression;
```

Example:

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

entity WhenExample is
    Port ( Signal_in : in std_logic;  -- Input signal
           Signal_out : out std_logic); -- Output signal
end WhenExample;

architecture Behavioral of WhenExample is
begin
    -- Concurrent conditional signal assignment
    Signal_out <= '1' when Signal_in = '0' else '0';  -- Output changes based on input
end Behavioral;
```

3. Selected Signal Assignment (Select-When Statement)

The select-when statement is used for assigning values to a signal based on multiple conditions. It works like a multi-way decision and is particularly useful for assigning signals based on specific patterns or conditions.

Syntax:

```
signal_name <= expression when condition1 else
               alternative_expression when condition2 else
               default_expression;
```

Example:

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

entity SelectExample is
   Port ( Signal_in : in std_logic_vector(1 downto 0);  -- 2-bit input signal
          Signal_out : out std_logic); -- Output signal
end SelectExample;

architecture Behavioral of SelectExample is
begin
   -- Concurrent selected signal assignment
   Signal_out <= '0' when Signal_in = "00" else
                 '1' when Signal_in = "01" else
                 'Z';  -- High impedance for other inputs
end Behavioral;
```

4. If-Else Concurrent Statement (Conditional Generate Statement)

The if-else construct can also be used in a concurrent context for conditional signal assignments, typically within a generate statement for structural design.

Syntax:

```
if condition then
   signal_name <= expression1;
else
   signal_name <= expression2;
end if;
```

Example:

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

entity IfElseConcurrentExample is
   Port ( Signal_in : in std_logic;  -- Input signal
          Signal_out : out std_logic); -- Output signal
```

```
end IfElseConcurrentExample;

architecture Behavioral of IfElseConcurrentExample is
begin
   -- Concurrent conditional assignment using if-else
   if Signal_in = '1' then
      Signal_out <= '0';
   else
      Signal_out <= '1';
   end if;
end Behavioral;
```

5. Block Statement

The **block statement** is used in VHDL to group a set of concurrent statements together, often to provide a local scope for signals, constants, or other declarations. It allows better organization and modularization of VHDL code, particularly in complex designs.

Syntax:

```
label: block [ (guard-expression) ]
begin
   concurrent_statements;
end block [ label ];
```

- **label**: An identifier for the block.
- **guard-expression**: An optional condition to enable or disable the block.
- **concurrent_statements**: The concurrent statements executed within the block.

Example:

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

entity BlockExample is
   Port ( A, B, C : in std_logic;
          Y1, Y2 : out std_logic);
end BlockExample;

architecture Behavioral of BlockExample is
begin
   -- Grouping logic using block
   Block1: block
```

```
    signal Temp: std_logic;  -- Local signal for the block
  begin
    Temp <= A and B;  -- Intermediate signal
    Y1 <= Temp or C;  -- Output based on Temp and input C
  end block Block1;
end Behavioral;
```

6. Concurrent Assert Statement

The **assert statement** in VHDL is primarily used for debugging and verification. A **concurrent assert statement** checks a condition continuously during simulation and reports a message if the condition fails.

Syntax:

```
assert condition
  report "message"
  severity level;
```

- **condition**: A boolean expression to check.
- **message**: A string to display if the condition is false.
- **severity level**: Indicates the seriousness (e.g., `note`, `warning`, `error`, `failure`).

Example:

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

entity ConcurrentAssertExample is
  Port ( A, B : in std_logic;
         C : out std_logic);
end ConcurrentAssertExample;

architecture Behavioral of ConcurrentAssertExample is
begin
  -- Concurrent signal assignment
  C <= A and B;

  -- Concurrent assert statement
  assert (A = '1' or B = '1')
    report "Both inputs are zero, unexpected behavior!"
    severity warning;
end Behavioral;
```

3.4 Differences Between Sequential and Concurrent Statements

Aspect	Sequential Statements	Concurrent Statements
Execution Order	Execute one after another, in the order written.	Execute independently and simultaneously.
Scope of Use	Used inside processes, functions, or procedures.	Defined outside processes in the architecture body.
Application	Suitable for modeling sequential logic like flip-flops, counters, and state machines.	Ideal for modeling combinational logic and parallel behavior.
Sensitivity	Triggered by changes in the sensitivity list or clock events.	Continuously evaluated whenever dependent signals change.
Nature of Execution	Sequential flow, mimicking software-like execution.	Parallel flow, representing hardware-like behavior.
Placement	Found in the body of a process, function, or procedure.	Found directly in the architecture or block statements.
Examples	if, case, loop, exit, etc.	when, select, block, concurrent assert, etc.

4 Modeling Styles of VHDL

Modeling styles in VHDL refer to the different approaches used to describe the behavior, structure, and functionality of a digital system. These styles allow designers to represent hardware systems at varying levels of abstraction, ranging from high-level behavior to detailed structural connections. Each modeling style serves a specific purpose and provides flexibility in describing different aspects of a design. The choice of a modeling style is critical in VHDL as it directly affects the readability, maintainability, and efficiency of the code. Some of the major modeling styles in VHDL are **Behavioral, Dataflow, Structural and Mixed Modeling Styles** as shows these modeling styles.

- **Behavioral Modeling** helps in understanding and verifying the system's functionality during the initial stages of design.
- **Dataflow Modeling** is useful for optimizing data operations and exploring the flow of data within the system.
- **Structural Modeling** is essential for implementing detailed hardware connections and verifying the final design.

- **Mixed Modeling** allows designers to combine styles, enabling a balanced approach to complex designs.

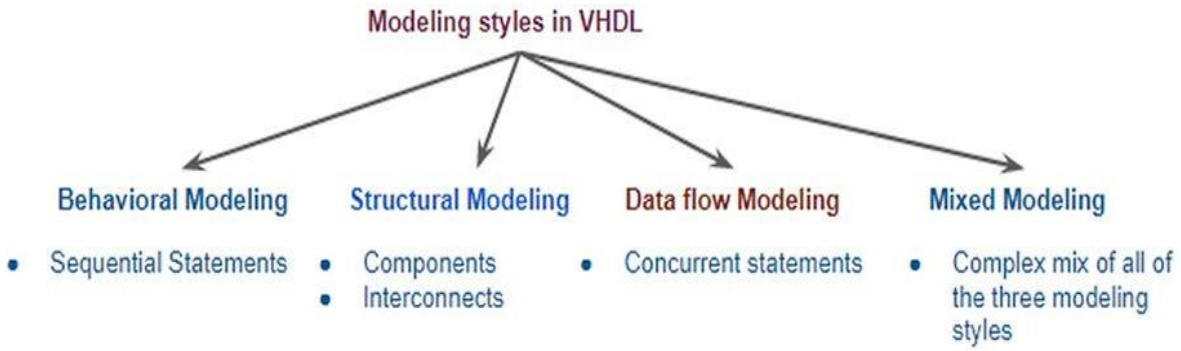


Figure 4-1: Modeling Styles in VHDL

1. Behavioral Modeling Style

Behavioral modeling is the highest-level abstraction in VHDL, focusing on **what a design should do**, rather than **how it is built**. It uses sequential statements inside **process blocks** to describe the system's functionality. This modeling style is simulation-oriented, allowing designers to validate functionality before moving to hardware implementation. The behavior of the system is event-driven, meaning that changes in signals (defined in the sensitivity list) trigger the execution of the process. Behavioral modeling is particularly suited for representing control logic, algorithmic operations, and finite state machines (FSMs), offering an abstract and flexible way to describe hardware behavior.

Example

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

entity BehavioralCounter is
    Port ( clk : in std_logic;       -- Clock signal
           reset : in std_logic;     -- Reset signal
           count : out integer range 0 to 15 ); -- 4-bit counter output
end BehavioralCounter;

architecture Behavioral of BehavioralCounter is
begin
    -- Process to describe the behavior of the counter
    process(clk, reset)
```

```
    begin
      if reset = '1' then
        count <= 0;  -- Reset the counter
      elsif rising_edge(clk) then
        count <= count + 1;  -- Increment the counter
      end if;
    end process;
end Behavioral;
```

2. Dataflow Modeling Style

Dataflow modeling describes a digital system's behavior using **concurrent signal assignments**, emphasizing the **flow of data** between signals. This modeling style expresses the design in terms of Boolean equations, logical expressions, and simple relationships between inputs and outputs. Unlike behavioral modeling, dataflow modeling relies heavily on **concurrent statements** and is closer to the hardware's functional implementation. It is particularly useful for **combinational logic design**, where the focus is on **what happens to the data as it propagates through the system**.

Dataflow modeling is a middle-level abstraction, providing a balance between high-level behavioral descriptions and low-level structural implementations. Designers can describe the system in a compact and intuitive way by specifying the logical or arithmetic relationships among signals.

Example

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

entity DataflowAdder is
    Port ( A : in std_logic_vector(3 downto 0);  -- 4-bit input A
           B : in std_logic_vector(3 downto 0);  -- 4-bit input B
           Cin : in std_logic;                   -- Carry-in
           Sum : out std_logic_vector(3 downto 0); -- 4-bit sum output
           Cout : out std_logic);                -- Carry-out
end DataflowAdder;

architecture Dataflow of DataflowAdder is
    signal temp_sum : std_logic_vector(4 downto 0); -- Intermediate 5-bit sum
begin
```

```
    -- Full adder logic using dataflow modeling
    temp_sum <= ('0' & A) + ('0' & B) + Cin;  -- Add A, B, and Cin
    Sum <= temp_sum(3 downto 0);            -- Assign lower 4 bits to Sum
    Cout <= temp_sum(4);                    -- Assign MSB to Cout
end Dataflow;
```

3. Structural Modeling Style

Structural modeling in VHDL emphasizes the physical arrangement and interconnection of hardware components, providing a hierarchical and modular approach to design. By connecting smaller, reusable components, designers can construct complex systems, similar to assembling circuits using gates and modules. This approach closely reflects the structure of actual hardware, making it well-suited for managing large and intricate designs. Key elements of structural modeling include **component declaration**, **component instantiation**, and **port mapping**, which are essential for defining and integrating these interconnected components effectively.

1. Component Declaration

This step defines the interface of a component by specifying its input and output ports. Component declarations provide a blueprint for the components to be used in the design and are typically placed in the architecture or package for reuse.

Syntax:

```
component ComponentName is
    Port (
        Port1 : in STD_LOGIC;
        Port2 : out STD_LOGIC;
        -- Add more ports as needed
    );
end component;
```

2. Component Instantiation and Port Mapping

Once a component is declared, it can be instantiated to create physical representations in the design. Each instance corresponds to an individual realization of the component, which can then be connected to other parts of the design. Port mapping connects the signals defined in the architecture to the ports of a component during instantiation. It ensures the proper flow of data between components, allowing them to communicate effectively.

Syntax:

```
InstanceLabel: ComponentName
    Port map (
        Port1 => Signal1,
        Port2 => Signal2
        -- Map additional ports as needed
    );
```

Example Full Adder Using Structural Modeling

A **Full Adder** is a combinational circuit that performs the addition of three binary inputs: two significant bits and a carry-in. Using structural modeling in VHDL, we can design a full adder by interconnecting two **Half Adders** and an **OR Gate**. The following example demonstrates how structural modeling principles, including **component declaration**, **component instantiation**, and **port mapping**, are used to assemble a larger design from smaller components.

VHDL Code for Half Adder

The Half Adder is a simple circuit that calculates the sum and carry for two input bits.

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

entity HalfAdder is
    Port (
        A    : in  STD_LOGIC;  -- Input A
        B    : in  STD_LOGIC;  -- Input B
        Sum  : out STD_LOGIC;  -- Sum output
        Carry: out STD_LOGIC   -- Carry output
    );
end HalfAdder;

architecture Behavioral of HalfAdder is
begin
    Sum  <= A XOR B;  -- XOR gate for Sum
    Carry <= A AND B; -- AND gate for Carry
end Behavioral;
```

VHDL Code for OR Gate

The OR Gate takes inputs and or each bit and give the result.

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

-- Entity for OR Gate
entity ORGate is
    Port ( Input1 : in STD_LOGIC;
           Input2 : in STD_LOGIC;
           Output : out STD_LOGIC);
```

```
end ORGate;

-- Architecture for OR Gate
architecture Behavioral of ORGate is
begin
    -- OR operation
    Output <= Input1 OR Input2;
end Behavioral;
```

Now, Full Adder Design Using Two Half Adders and OR Gate

The Full Adder combines two Half Adders and an OR gate to calculate the final `Sum` and `Carry-Out`.

Entity Declaration:

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

entity FullAdder is
    Port (
        A    : in  STD_LOGIC;  -- Input A
        B    : in  STD_LOGIC;  -- Input B
        Cin  : in  STD_LOGIC;  -- Carry-in
        Sum  : out STD_LOGIC; -- Sum output
        Cout : out STD_LOGIC  -- Carry-out
    );
end FullAdder;
```

Architecture (Structural):

```
-- Architecture for Full Adder using Structural Modeling
architecture Structural of FullAdder is
   -- Component Declarations
   component HalfAdder
      Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           Sum : out STD_LOGIC;
           Carry : out STD_LOGIC);
   end component;

   component ORGate
      Port ( Input1 : in STD_LOGIC;
           Input2 : in STD_LOGIC;
           Output : out STD_LOGIC);
   end component;

   -- Internal Signals
   signal Sum1, Carry1, Carry2 : STD_LOGIC;
```

```
begin
   -- Instantiate First Half Adder
   HA1: HalfAdder
      port map ( A => A, B => B, Sum => Sum1, Carry => Carry1 );

   -- Instantiate Second Half Adder
   HA2: HalfAdder
      port map ( A => Sum1, B => Cin, Sum => Sum, Carry => Carry2 );

   -- Instantiate OR Gate
   OR1: ORGate
      port map ( Input1 => Carry1, Input2 => Carry2, Output => Cout );
end Structural;
```

4. Mixed Modeling Style

Mixed modeling combines multiple modeling styles, such as **behavioral**, **dataflow**, and **structural**, within a single design. This approach allows designers to leverage the strengths of each style for different parts of the system. For instance, structural modeling can describe the overall organization of components, behavioral modeling can specify complex processes or finite state machines, and dataflow modeling can be used for arithmetic or logical expressions. Mixed modeling is particularly useful for complex systems, enabling a clear and modular design while optimizing specific sections for functionality or performance.

By combining styles, mixed modeling provides flexibility, readability, and scalability, allowing the design to be expressed in the most appropriate way for each subsystem. It is a highly practical method for hierarchical designs and facilitates simulation and testing of individual components.

Example Full Adder Using Mixed Modeling Style

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

-- Top-Level Entity for Full Adder
entity FullAdder_Mixed is
   Port ( A : in STD_LOGIC;
          B : in STD_LOGIC;
```

```
        Cin : in STD_LOGIC;
        Sum : out STD_LOGIC;
        Cout : out STD_LOGIC);
end FullAdder_Mixed;

-- Architecture with Mixed Modeling Style
architecture MixedModel of FullAdder_Mixed is
   -- Component Declaration for Half Adder
   component HalfAdder
      Port ( A : in STD_LOGIC;
             B : in STD_LOGIC;
             Sum : out STD_LOGIC;
             Carry : out STD_LOGIC);
   end component;

   -- Signal Declaration
   signal HA1_Sum, HA1_Carry, HA2_Carry : STD_LOGIC;
begin
   -- Component Instantiation for First Half Adder
   HA1: HalfAdder
      port map ( A => A, B => B, Sum => HA1_Sum, Carry => HA1_Carry );

   -- Dataflow Modeling for Second Half Adder
   HA2_Sum <= HA1_Sum XOR Cin;
   HA2_Carry <= HA1_Sum AND Cin;

   -- Behavioral Modeling for Carry-out and Sum
   process(HA1_Carry, HA2_Carry)
   begin
      Cout <= HA1_Carry OR HA2_Carry;  -- Carry-out logic
   end process;

   Sum <= HA2_Sum;  -- Final Sum output
end MixedModel;
```

5. Comparison of VHDL Modeling Styles

VHDL offers three primary modeling styles **Behavioral**, **Dataflow**, and **Structural**, each serving unique purposes in digital design. The table below highlights the key distinctions among these modeling styles, along with a concise comparison.

Feature	Behavioral Modeling	Dataflow Modeling	Structural Modeling
Description	Describes the system's behavior using sequential statements within processes.	Focuses on the flow of data using concurrent assignments and logical expressions.	Models the actual hardware structure by connecting components hierarchically.
Level of Abstraction	High-level abstraction.	Medium-level abstraction.	Low-level abstraction, closely mimicking physical hardware.
Primary Use Case	Suitable for complex algorithms, state machines, and control logic.	Ideal for combinational logic and arithmetic operations.	Best for representing hierarchical systems and interconnections among components.
Statements Used	Sequential statements like if-else, case, loop.	Concurrent statements like with-select, when, and signal assignments.	Component declarations, instantiations, and port mappings.
Simulation Focus	Focuses on the system's functionality without detailed hardware implementation.	Focuses on functional behavior and relationships between inputs and outputs.	Emphasizes hardware structure and connections for accurate synthesis.
Readability	Easier to write and understand for algorithms and logic.	Compact and intuitive for arithmetic or logical relations.	Can be complex for large designs due to detailed interconnections.
Example	A process implementing state transitions or a counter.	Boolean equations for a full adder.	Full adder implemented using two half adders and an OR gate.

5 VHDL Realization for Combinational Circuit

Combinational circuits are digital logic systems where the output depends only on the current inputs, without any memory or feedback loops. VHDL is widely used to describe and implement these circuits, leveraging its ability to define the functional relationships between inputs and outputs concisely. The most popular combinational circuits essential in digital design include **multiplexers**, **demultiplexers**, **encoders**, **decoders**, **adders**, **subtraction circuits**, **comparators**, and **Boolean function implementations**.

1. Full Adders

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

entity FullAdder is
   Port ( A    : in  std_logic;          -- Input A
          B    : in  std_logic;          -- Input B
          Cin  : in  std_logic;          -- Carry Input
```

```
        Sum   : out std_logic;          -- Sum Output
        Cout  : out std_logic);          -- Carry Output
end FullAdder;

architecture Behavioral of FullAdder is
begin
   process(A, B, Cin)
   begin
      -- Check the sum and carry based on inputs using if statement
      if (A = '0' and B = '0') then
         Sum <= Cin;
         Cout <= '0';
      elsif (A = '0' and B = '1') then
         Sum <= Cin;
         Cout <= '0';
      elsif (A = '1' and B = '0') then
         Sum <= Cin;
         Cout <= '0';
      else  -- when A = '1' and B = '1'
         Sum <= Cin;
         Cout <= '1';
      end if;
   end process;
end Behavioral;
```

2. Half Subtractor

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

entity HalfSubtractor is
   Port ( A     : in  std_logic;        -- Input A
          B     : in  std_logic;        -- Input B
          Diff  : out std_logic;        -- Difference Output
          Bout  : out std_logic);       -- Borrow Output
end HalfSubtractor;

architecture Dataflow of HalfSubtractor is
begin
   -- Difference is the XOR of A and B
   Diff <= A xor B;

   -- Borrow is the NOT of A AND B
   Bout <= not A and B;
```

```
end Dataflow;
```

3. Multiplexers (MUX)

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

entity Mux4to1 is
   Port ( A : in  std_logic;          -- Input A
          B : in  std_logic;          -- Input B
          C : in  std_logic;          -- Input C
          D : in  std_logic;          -- Input D
          S0 : in  std_logic;         -- Select line 0
          S1 : in  std_logic;         -- Select line 1
          Y  : out std_logic);        -- Output Y
end Mux4to1;

architecture Behavioral of Mux4to1 is
begin
   process (A, B, C, D, S0, S1)
   begin
      case (S1 & S0) is
         when "00" => Y <= A;  -- If S1S0 = 00, output A
         when "01" => Y <= B;  -- If S1S0 = 01, output B
         when "10" => Y <= C;  -- If S1S0 = 10, output C
         when "11" => Y <= D;  -- If S1S0 = 11, output D
         when others => Y <= '0'; -- Default case (safety net)
      end case;
   end process;
end Behavioral;
```

4. Demultiplexers (DEMUX)

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

entity Demux1to4 is
   Port ( D : in  std_logic;          -- Data input
          S : in  std_logic_vector(1 downto 0);  -- 2-bit select line (bus)
          Y : out std_logic_vector(3 downto 0)  -- 4-bit output (each bit routed based on
S)
```

```
        );
end Demux1to4;

architecture Behavioral of Demux1to4 is
begin
   process (D, S)
   begin
      -- Default case: no output active
      Y <= "0000";

      case S is
         when "00" => Y(0) <= D;  -- If S = "00", route D to Y(0)
         when "01" => Y(1) <= D;  -- If S = "01", route D to Y(1)
         when "10" => Y(2) <= D;  -- If S = "10", route D to Y(2)
         when "11" => Y(3) <= D;  -- If S = "11", route D to Y(3)
         when others => Y <= "0000"; -- Default case
      end case;
   end process;
end Behavioral;
```

5. 4-to-2 Binary Encoders

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

entity BinaryEncoder is
   Port ( D : in  std_logic_vector(3 downto 0);  -- 4-bit input
          Y : out std_logic_vector(1 downto 0)   -- 2-bit output
          );
end BinaryEncoder;

architecture Behavioral of BinaryEncoder is
begin
   -- Using conditional signal assignment (when-else) to encode
   Y <= "00" when (D(3) = '1') else  -- If D3 is high, output "11"
        "01" when (D(2) = '1') else  -- If D2 is high, output "10"
        "10" when (D(1) = '1') else  -- If D1 is high, output "01"
        "11";                        -- If D0 is high, output "00"

end Behavioral;
```

6. 2-to-4 Decoders

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

entity Decoder is
   Port ( A : in  std_logic_vector(1 downto 0);  -- 2-bit input
          Y : out std_logic_vector(3 downto 0)   -- 4-bit output (one-hot)
          );
end Decoder;

architecture Behavioral of Decoder is
begin
   process (A)  -- Process that reacts to changes in A
   begin
      case A is
         when "00" => Y <= "0001";  -- If A is 00, output is 0001
         when "01" => Y <= "0010";  -- If A is 01, output is 0010
         when "10" => Y <= "0100";  -- If A is 10, output is 0100
         when "11" => Y <= "1000";  -- If A is 11, output is 1000
         when others => Y <= "0000";  -- Default case for safety
      end case;
   end process;
end Behavioral;
```

7. 2-Bit Comparator

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

entity Comparator is
   Port ( A : in  std_logic_vector(1 downto 0);  -- 2-bit input A
          B : in  std_logic_vector(1 downto 0);  -- 2-bit input B
          G : out std_logic;                     -- Output: A > B (Greater)
          L : out std_logic;                     -- Output: A < B (Lesser)
          E : out std_logic                      -- Output: A = B (Equal)
          );
end Comparator;

architecture Behavioral of Comparator is
begin
   process (A, B)  -- Process triggered by changes in A or B
   begin
      if (A = B) then
```

```
            E <= '1';  -- A is equal to B
            G <= '0';  -- A is not greater than B
            L <= '0';  -- A is not less than B
        elsif (A > B) then
            E <= '0';  -- A is not equal to B
            G <= '1';  -- A is greater than B
            L <= '0';  -- A is not less than B
        else
            E <= '0';  -- A is not equal to B
            G <= '0';  -- A is not greater than B
            L <= '1';  -- A is less than B
        end if;
    end process;
end Behavioral;
```

8. Even Parity Generators

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

entity EvenParityGenerator is
    Port ( data : in  std_logic_vector(3 downto 0);  -- 4-bit input data
           parity_bit : out std_logic                -- Even parity bit
           );
end EvenParityGenerator;

architecture Behavioral of EvenParityGenerator is
begin
    process (data)  -- Process triggered by changes in data input
    variable count : integer := 0;  -- Variable to count the number of '1's
    begin
        -- Count the number of '1' bits in the data
        for i in 0 to 3 loop
            if (data(i) = '1') then
                count := count + 1;
            end if;
        end loop;

        -- If count is odd, set parity bit to '1' (to make total number of 1's even)
        if (count mod 2 = 1) then
            parity_bit <= '1';
        else
            parity_bit <= '0';
        end if;
```

```
    end process;
end Behavioral;
```

9. Boolean Logic Circuits

i. AND Gate

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

entity AND_Gate is
    Port ( A : in  std_logic;  -- Input A
           B : in  std_logic;  -- Input B
           Y : out std_logic   -- Output Y (A AND B)
           );
end AND_Gate;

architecture Behavioral of AND_Gate is
begin
    Y <= A AND B;  -- AND operation
end Behavioral;
```

ii. OR Gate

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

entity OR_Gate is
    Port ( A : in  std_logic;  -- Input A
           B : in  std_logic;  -- Input B
           Y : out std_logic   -- Output Y (A OR B)
           );
end OR_Gate;

architecture Behavioral of OR_Gate is
begin
    Y <= A OR B;  -- OR operation
end Behavioral;
```

iii. XOR Gate

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

entity XOR_Gate is
    Port ( A : in  std_logic;  -- Input A
           B : in  std_logic;  -- Input B
           Y : out std_logic   -- Output Y (A XOR B)
           );
end XOR_Gate;
```

```
architecture Behavioral of XOR_Gate is
begin
   Y <= A XOR B;  -- XOR operation
end Behavioral;
```

10. Arithmetic Logic Unit (ALU)

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.NUMERIC_STD.all;

-- Entity Declaration for ALU
entity ALU is
  generic (
    constant N: natural := 1  -- Number of shifted or rotated bits
    );

    Port (
    A       : in  STD_LOGIC_VECTOR(3 downto 0);   -- 4-bit input A
    B       : in  STD_LOGIC_VECTOR(3 downto 0);   -- 4-bit input B
    ALU_Sel : in  STD_LOGIC_VECTOR(2 downto 0);   -- 3-bit selector for ALU
operation
    ALU_Out : out STD_LOGIC_VECTOR(3 downto 0);   -- 4-bit ALU output result
    Carryout: out std_logic                       -- Carryout flag to indicate overflow/carry
    );
end ALU;

-- Architecture Body for ALU
architecture Behavioral of ALU is

  -- Internal Signals
  signal ALU_Result : std_logic_vector (3 downto 0); -- ALU operation result
  signal tmp        : std_logic_vector (4 downto 0); -- Temporary signal for carryout
calculation

begin

   -- Process for ALU operations based on ALU_Sel
   process(A, B, ALU_Sel)
   begin
     case(ALU_Sel) is
```

```
        -- Addition (000)
        when "000" =>
          ALU_Result <= A + B;

        -- Subtraction (001)
        when "001" =>
          ALU_Result <= A - B;

        -- AND (010)
        when "010" =>
          ALU_Result <= A and B;

        -- OR (011)
        when "011" =>
          ALU_Result <= A or B;

        -- XOR (100)
        when "100" =>
          ALU_Result <= A xor B;

        -- NOT A (101)
        when "101" =>
          ALU_Result <= not A;

        -- Logical shift left (110)
        when "110" =>
          ALU_Result <= std_logic_vector(unsigned(A) sll N);

        -- Logical shift right (111)
        when "111" =>
          ALU_Result <= std_logic_vector(unsigned(A) srl N);

        -- Default case (when no match)
        when others =>
          ALU_Result <= A + B;  -- Default to addition operation

    end case;
end process;

-- ALU output assignment
ALU_Out <= ALU_Result;

-- Carryout flag calculation (taking the MSB of the sum)
tmp <= ('0' & A) + ('0' & B);
```

```
    Carryout <= tmp(4); -- Carryout is set to the 5th bit (overflow flag)

end Behavioral;
```

Opcode (ALU_Sel)	Operation	Description
0	Addition	Perform A + B
1	Subtraction	Perform A - B
10	Logical AND	Perform A AND B
11	Logical OR	Perform A OR B
100	Logical XOR	Perform A XOR B
101	Logical NOT (A)	Perform NOT A (inverts A)
110	Logical Shift Left	Shift A left by N bits
111	Logical Shift Right	Shift A right by N bits

6 VHDL Realization for Sequential Circuit

Sequential circuits are digital circuits in which the output depends not only on the current inputs but also on the previous inputs, meaning they have memory. Unlike combinational circuits, which respond instantaneously to input changes, sequential circuits rely on clock signals to synchronize their behavior and store information. They are used to implement systems that require storage, timing, or state transitions, such as counters, registers, flip-flops, and finite state machines (FSMs).

These circuits are crucial for building systems like digital clocks, control units, and memory devices, where the sequence of events and previous states influences future actions. The core components of sequential circuits include storage elements (like flip-flops), combinational logic for processing, and a clock signal to trigger state transitions.

Here are 10 of the most important sequential circuits widely used in digital design:

1. **Flip-Flops**
   i. **SR Flip-Flop**

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

entity SR_FlipFlop is
    Port (
```

```
        S, R   : in  STD_LOGIC;  -- Set and Reset inputs
        CLK    : in  STD_LOGIC;  -- Clock input
        Q      : out STD_LOGIC;  -- Output Q
        Q_bar  : out STD_LOGIC   -- Complementary output Q_bar
    );
end SR_FlipFlop;

architecture Behavioral of SR_FlipFlop is
begin
    process(CLK)
    begin
        if rising_edge(CLK) then
            if (S = '1' and R = '0') then
                Q <= '1';
                Q_bar <= '0';
            elsif (S = '0' and R = '1') then
                Q <= '0';
                Q_bar <= '1';
            elsif (S = '0' and R = '0') then
                Q <= Q;  -- Hold state
                Q_bar <= Q_bar;
            else
                Q <= '0';  -- Invalid state
                Q_bar <= '0';
            end if;
        end if;
    end process;
end Behavioral;
```

ii. D Flip-Flop

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

entity D_FlipFlop is
    Port (
        D   : in  STD_LOGIC;  -- Data input
        CLK : in  STD_LOGIC;  -- Clock input
        Q   : out STD_LOGIC   -- Output Q
    );
end D_FlipFlop;

architecture Behavioral of D_FlipFlop is
begin
    process(CLK)
```

```
    begin
        if rising_edge(CLK) then
            Q <= D;  -- Assign the value of D to Q on the rising edge of CLK
        end if;
    end process;
end Behavioral;
```

iii. JK Flip-Flop

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

entity JK_FlipFlop is
    Port (
        J, K  : in  STD_LOGIC;  -- Inputs J and K
        CLK   : in  STD_LOGIC;  -- Clock input
        Q     : out STD_LOGIC;  -- Output Q
        Q_bar : out STD_LOGIC   -- Complementary output Q_bar
    );
end JK_FlipFlop;

architecture Behavioral of JK_FlipFlop is
begin
    process(CLK)
    begin
        if rising_edge(CLK) then
            if (J = '0' and K = '0') then
                Q <= Q;  -- Hold state
            elsif (J = '0' and K = '1') then
                Q <= '0';  -- Reset
            elsif (J = '1' and K = '0') then
                Q <= '1';  -- Set
            elsif (J = '1' and K = '1') then
                Q <= not Q;  -- Toggle
            end if;
        end if;
    end process;

    Q_bar <= not Q;  -- Complementary output
end Behavioral;
```

2. Serial-in-Serial-out Shift Registers

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

entity SISO_ShiftRegister is
    Port (
```

```
        CLK   : in  STD_LOGIC;        -- Clock input
        RESET : in  STD_LOGIC;        -- Reset input
        D_IN  : in  STD_LOGIC;        -- Serial input
        Q_OUT : out STD_LOGIC         -- Serial output
    );
end SISO_ShiftRegister;

architecture Behavioral of SISO_ShiftRegister is
    signal shift_reg : STD_LOGIC_VECTOR(3 downto 0);  -- 4-bit shift register
begin
    process(CLK, RESET)
    begin
        if RESET = '1' then
            shift_reg <= (others => '0');  -- Reset all bits
        elsif rising_edge(CLK) then
            shift_reg <= shift_reg(2 downto 0) & D_IN;  -- Shift data in
        end if;
    end process;

    Q_OUT <= shift_reg(3);  -- Serial output (MSB)
end Behavioral;
```

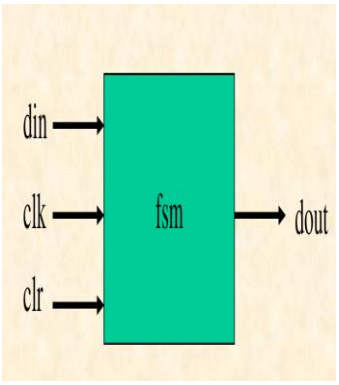
3. Binary Counter

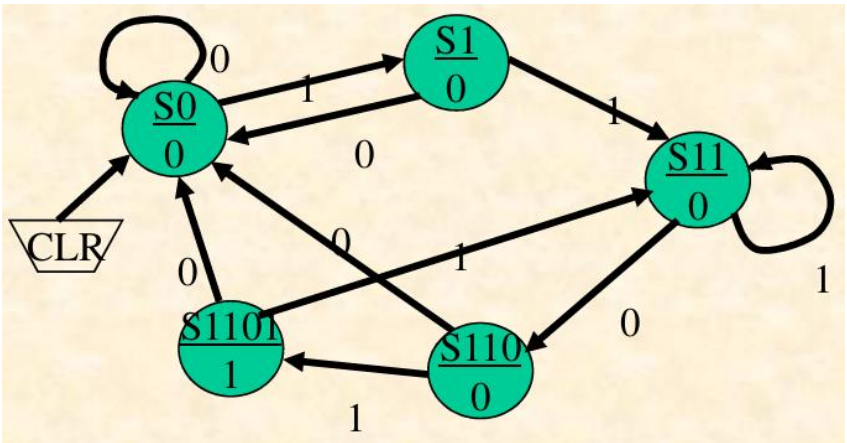
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity Binary_Counter is
    generic (
        N : natural := 4  -- Number of bits for the counter
    );
    Port (
        CLK   : in  STD_LOGIC;             -- Clock signal
        RESET : in  STD_LOGIC;             -- Reset signal
        COUNT : out STD_LOGIC_VECTOR(N-1 downto 0)  -- Counter output
    );
end Binary_Counter;

architecture Behavioral of Binary_Counter is
    signal counter : UNSIGNED(N-1 downto 0) := (others => '0');  -- Internal counter
begin
    process(CLK, RESET)
    begin
        if RESET = '1' then
            counter <= (others => '0');  -- Reset the counter to 0
        elsif rising_edge(CLK) then
```

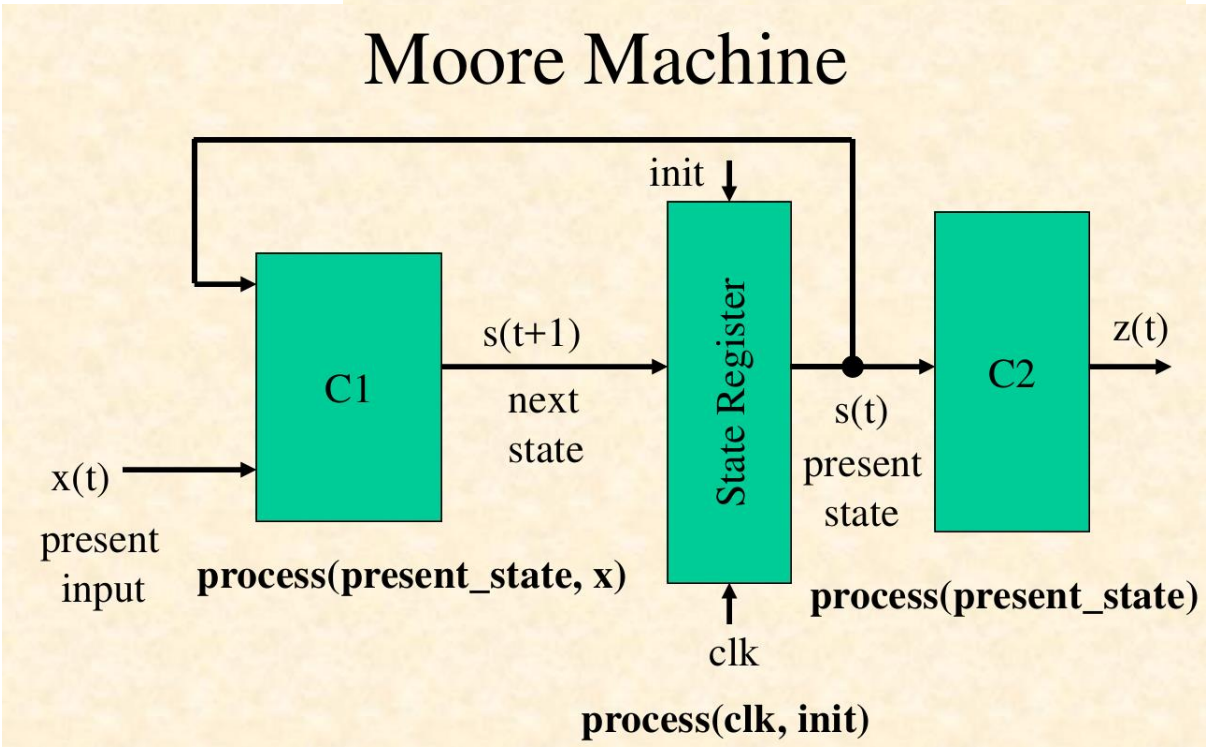
```
        counter <= counter + 1;      -- Increment the counter
    end if;
  end process;

    COUNT <= STD_LOGIC_VECTOR(counter);  -- Output the counter value
end Behavioral;
```

4. Finite State Machines (FSM)

i. 1101 Sequence Detector Using Moore Machine








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

-- Entity declaration for the FSM
entity fsm is
    Port (
        clk  : in  STD_LOGIC;   -- Clock input
```

```
        clr  : in  STD_LOGIC;   -- Asynchronous clear/reset

        din  : in  STD_LOGIC;   -- Input data
        dout : out STD_LOGIC    -- Output signal
    );
end fsm;

-- Architecture for the FSM
architecture fsm_arch of fsm is
    -- Define the state type with all possible states
    type state_type is (S0, S1, S11, S110, S1101);
    signal present_state, next_state: state_type; -- Signals for current and next states
begin
    -- State register process: handles state transitions
    synch: process(clk, clr)
    begin
        if clr = '1' then               -- Asynchronous clear
            present_state <= S0;        -- Reset to the initial state (S0)
        elsif clk'event and clk = '1' then -- On rising edge of the clock
            present_state <= next_state;   -- Update present state with the next state
        end if;
    end process;

    -- Combinational process to determine the next state
    comb1: process(present_state, din)
    begin
        case present_state is
            when S0 =>
                if din = '1' then
                    next_state <= S1;  -- Transition to S1 if input is 1
                else
                    next_state <= S0;  -- Remain in S0 if input is 0
                end if;
            when S1 =>
                if din = '1' then
                    next_state <= S11; -- Transition to S11 if input is 1
                else
                    next_state <= S0;  -- Return to S0 if input is 0
                end if;
            when S11 =>
                if din = '0' then
                    next_state <= S110; -- Transition to S110 if input is 0
                else
                    next_state <= S11;  -- Remain in S11 if input is 1
                end if;
            when S110 =>
                if din = '1' then
                    next_state <= S1101; -- Transition to S1101 if input is 1
                else
                    next_state <= S0;    -- Return to S0 if input is 0
                end if;
            when S1101 =>
```

```
            if din = '0' then
               next_state <= S0;   -- Return to S0 if input is 0
            else
               next_state <= S11;  -- Transition to S11 if input is 1
            end if;
         when others =>
            next_state <= S0;       -- Default to S0 for safety
      end case;
   end process;

   -- Combinational process to determine the output
   comb2: process(present_state)
   begin
      if present_state = S1101 then
         dout <= '1';  -- Set output to 1 when in state S1101
      else
         dout <= '0';  -- Set output to 0 for all other states
      end if;
   end process;

end fsm_arch;
```

5. Read-Only Memory

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

entity ROM is
   Port (
      ADDR  : in  STD_LOGIC_VECTOR(1 downto 0);  -- 2-bit address input (4 locations)
      DATA  : out STD_LOGIC_VECTOR(7 downto 0)   -- 8-bit data output
   );
end ROM;

architecture Behavioral of ROM is
   -- Declare a constant array for the ROM contents
   type ROM_ARRAY is array (0 to 3) of STD_LOGIC_VECTOR(7 downto 0);
   constant ROM_CONTENTS : ROM_ARRAY := (
      "00000001",  -- Data at address 00
      "00000010",  -- Data at address 01
      "00000100",  -- Data at address 10
      "00001000"   -- Data at address 11
   );
begin
   -- Process to read data based on address
   process(ADDR)
   begin
      DATA<=ROM_CONTENTS(to_integer(unsigned(ADDR)));   --Output corresponding data
   end process;
end Behavioral;
```

7 Subprogram and Packages in VHDL

Subprograms and packages in VHDL are powerful tools that promote code reusability, modularity, and organization. Subprograms, including functions and procedures, enable encapsulation of specific operations, while packages serve as a centralized repository for commonly used components, data types, and functions. Together, they simplify complex designs and ensure consistency across projects, making VHDL coding more efficient and maintainable.

1. Subprograms in VHDL

Subprograms in VHDL are reusable code blocks that enhance modularity and are classified into **functions** and **procedures**. A **function** computes and returns a single value and accepts only `in` parameters.

For Example

```
function Add_Bits(A, B: std_logic) return std_logic is
begin
    return A xor B;
end Add_Bits;
```

A **procedure** performs operations and modifies multiple signals through `in`, `out`, or `inout` parameters.

For Example

```
procedure Half_Adder(A, B: in std_logic; Sum, Carry: out std_logic)
is
begin
    Sum <= A xor B;
    Carry <= A and B;
end Half_Adder;
```

Subprograms can be used in architectures, processes, or packages to ensure code reusability and clarity.

2. Package

A package is a collection of reusable elements like data types, components, functions, and constants. These elements, once defined in a package, can be used across multiple VHDL design units. Packages are especially useful for **Global Information i.e.** Storing shared

parameters and data types in complex designs and **Team Projects** i.e. Sharing important details across different team members.

A package can be divided into two parts:

- **Package Declaration**: Contains declarations (e.g., constants, types, or functions).
- **Package Body**: Contains definitions (e.g., actual code for functions and values for constants).

Example

Package Declaration

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

package MySimplePackage is
    -- Declare a constant
    constant DELAY_TIME : time;

    -- Declare a function
    function Add_Bits(A, B: std_logic) return std_logic;

    -- Declare a type
    type State_Type is (Idle, Working, Done);

    -- Declare a component
    component Half_Adder is
        Port (
            A : in std_logic;
            B : in std_logic;
            Sum : out std_logic;
            Carry : out std_logic
        );
    end component;
end MySimplePackage;
```

Package Body

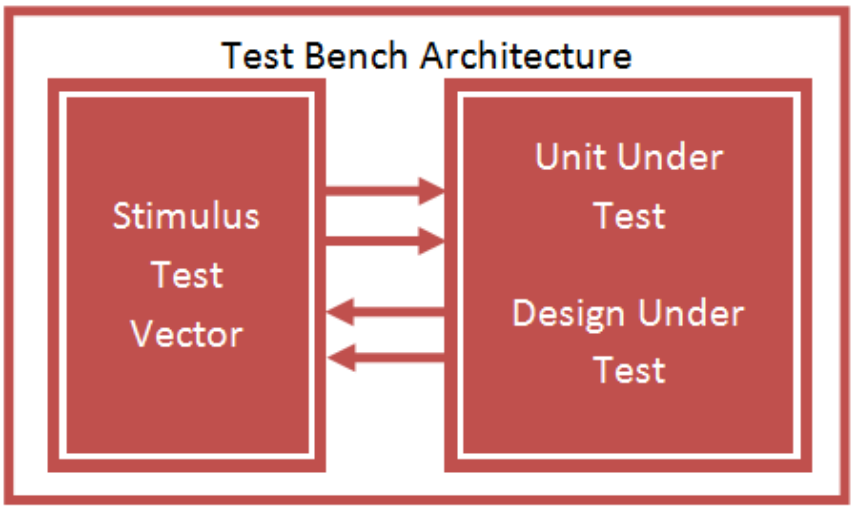
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

package body MySimplePackage is
    -- Define the constant
    constant DELAY_TIME : time := 5 ns;

    -- Define the function
    function Add_Bits(A, B: std_logic) return std_logic is
    begin
        return A xor B; -- Simple XOR operation
    end Add_Bits;
end MySimplePackage;
```

Example of Using the Package

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

entity Test_Design is
    Port (
        A : in std_logic;
        B : in std_logic;
        Sum : out std_logic;
        Carry : out std_logic
    );
end Test_Design;

architecture Behavioral of Test_Design is
    signal Temp_Sum : std_logic;
    signal Temp_Carry : std_logic;

    -- Instantiate the Half_Adder component
    component Half_Adder is
        Port (
            A : in std_logic;
            B : in std_logic;
            Sum : out std_logic;
            Carry : out std_logic
        );
    end component;
begin
    HA1: Half_Adder
    port map (
        A => A,
        B => B,
        Sum => Temp_Sum,
        Carry => Temp_Carry
    );

    -- Use the function from the package
    Sum <= Add_Bits(A, B);
    Carry <= Temp_Carry;
end Behavioral;
```

8 Test Bench

A VHDL Testbench is an essential part of the VHDL design process. It is used to verify the functionality of a design by simulating the output based on different input conditions. The Testbench provides a controlled environment to generate stimulus (inputs) for the design under test (DUT) or unit under test (UUT) and check its response. In simple terms, it lets you simulate the behavior of your design by applying test cases.



Key Components of a VHDL Testbench

- **Entity Declaration**: The testbench does not require input or output ports because it is used solely for simulation.
- **Architecture Declaration**: The main section of the testbench, where signals, components, and processes are defined.
- **Component Declaration**: The design under test (DUT) or UUT is declared as a component within the testbench.
- **Signal Declaration**: Signals are declared for connecting the testbench to the UUT. These signals will be driven by the testbench and used as inputs and outputs in the UUT.
- **Clock and Stimulus Generation**: A process to generate a clock signal (if applicable) and another to generate stimulus inputs for testing the UUT.
- **Component Instantiation**: The UUT is instantiated and connected to the testbench signals.

Example: VHDL Testbench for Half Adder:

Half Adder VHDL Code

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY HalfAdder IS
    PORT (
        A     : IN  std_logic;
        B     : IN  std_logic;
        Sum   : OUT std_logic;
        Carry : OUT std_logic
    );
END HalfAdder;

ARCHITECTURE behavior OF HalfAdder IS
BEGIN
    -- Half Adder logic
    Sum   <= A XOR B;
    Carry <= A AND B;
END behavior;
```

Testbench for Half Adder

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

ENTITY tb_HalfAdder IS
END tb_HalfAdder;

ARCHITECTURE tb_arch OF tb_HalfAdder IS
    -- Component declaration for the Half Adder (UUT)
    COMPONENT HalfAdder
        PORT (
            A     : IN  std_logic;
            B     : IN  std_logic;
            Sum   : OUT std_logic;
            Carry : OUT std_logic
        );
    END COMPONENT;

    -- Internal signals for the testbench
    SIGNAL A     : std_logic := '0';
    SIGNAL B     : std_logic := '0';
    SIGNAL Sum   : std_logic;
    SIGNAL Carry : std_logic;

    -- Clock period definition (not used in this case)
    CONSTANT clock_period : time := 20 ns;

BEGIN
    -- Instantiate the Unit Under Test (UUT)
    uut: HalfAdder PORT MAP (
        A     => A,
        B     => B,
        Sum   => Sum,
        Carry => Carry
    );

    -- Stimulus generation process
```

```
    stimulus_process : PROCESS
    BEGIN
        -- Test case 1
        WAIT FOR 20 ns;
        A <= '0'; B <= '0';  -- A=0, B=0
        WAIT FOR 20 ns;

        -- Test case 2
        A <= '0'; B <= '1';  -- A=0, B=1
        WAIT FOR 20 ns;

        -- Test case 3
        A <= '1'; B <= '0';  -- A=1, B=0
        WAIT FOR 20 ns;

        -- Test case 4
        A <= '1'; B <= '1';  -- A=1, B=1
        WAIT FOR 20 ns;

        -- End the testbench
        WAIT;
    END PROCESS;

END ARCHITECTURE;
```