Chapter 2: Proficiency in VHDL and Verilog Programming

2.1 Introduction to VHDL and Verilog

In the world of FPGA programming, **VHDL (VHSIC Hardware Description Language)** and **Verilog** are the two primary hardware description languages (HDLs) used to describe the behavior and structure of digital circuits. Both VHDL and Verilog allow designers to model complex systems at various abstraction levels, from high-level functionality to low-level hardware behavior.

This chapter will guide you through the key features of VHDL and Verilog, helping you gain proficiency in these languages for FPGA design. You will learn how to use these languages to describe and implement digital circuits, understand the differences between them, and explore how they interact with FPGA architectures.

2.2 VHDL Programming Language

2.2.1 Introduction to VHDL

VHDL is a strongly typed, verbose language used to model the behavior of digital circuits. It was originally developed for the U.S. Department of Defense's VHSIC (Very High-Speed Integrated Circuit) program and has since become the standard HDL for designing complex digital systems. VHDL is known for its high level of abstraction and strong support for simulation and testing.

2.2.2 VHDL Basic Structure

A typical VHDL program consists of three key parts:

1. **Entity**: The entity defines the interface to the circuit, specifying the input and output ports.

Example:

```
 ENTITY AND_GATE IS
  PORT (
    A : IN BIT;
    B : IN BIT;
    Y : OUT BIT
  );
END ENTITY;
```

    ○
2. **Architecture**: The architecture describes the internal workings of the circuit. It defines how the inputs are processed to produce the outputs.

Example:

```
 ARCHITECTURE behavior OF AND_GATE IS
BEGIN
  Y <= A AND B;
END ARCHITECTURE;
```

    ○
3. **Configuration**: This part binds an entity to a specific architecture and is optional in most cases.

2.2.3 VHDL Data Types and Operators

- **Data Types**: VHDL supports several data types, including `BIT`, `INTEGER`, `BOOLEAN`, `SIGNED`, and `UNSIGNED`.

- **Operators**: VHDL supports a range of operators such as logical (`AND`, `OR`), arithmetic (`+`, `-`, `*`), and relational (`=`, `<`, `>`) operators.

2.2.4 Example: Simple AND Gate

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

ENTITY AND_GATE IS
  PORT (
    A : IN STD_LOGIC;
    B : IN STD_LOGIC;
    Y : OUT STD_LOGIC
  );
END ENTITY AND_GATE;

ARCHITECTURE behavior OF AND_GATE IS
BEGIN
  Y <= A AND B;
END ARCHITECTURE behavior;
```

In this example, the `AND_GATE` entity defines two inputs `A` and `B`, and one output `Y`. The architecture implements the AND logic using the `<=` assignment operator.

2.3 Verilog Programming Language

2.3.1 Introduction to Verilog

Verilog is a hardware description language that is similar in syntax to the C programming language. It is widely used in the industry for FPGA design and digital circuit simulation. Verilog is more concise than VHDL, making it easier to write but also somewhat less verbose.

2.3.2 Verilog Basic Structure

A Verilog program typically consists of the following elements:

1. **Module**: The module is the basic unit of design in Verilog. It defines the inputs, outputs, and the internal behavior of the circuit.

Example:

```
 module AND_GATE (input A, input B, output Y);
  assign Y = A & B;
endmodule
```

2. **Always Block**: The `always` block is used to define behavior that executes continuously whenever the input signals change.

Example:

```
 always @ (A or B) begin
  Y = A & B;
end
```

2.3.3 Verilog Data Types and Operators

- **Data Types**: Verilog supports `wire`, `reg`, `integer`, `real`, `time`, and other types to represent signals and variables.

- **Operators**: Verilog supports similar operators to VHDL, such as bitwise (`&`, `|`), arithmetic (`+`, `-`, `*`), and comparison (`==`, `!=`) operators.

2.3.4 Example: Simple AND Gate

```
module AND_GATE (input A, input B, output Y);
  assign Y = A & B;
endmodule
```

In this Verilog example, the `AND_GATE` module defines two inputs `A` and `B`, and one output `Y`. The `assign` statement is used to implement the AND operation between `A` and `B`.

2.4 Comparing VHDL and Verilog

While both VHDL and Verilog are used for FPGA design, there are significant differences between the two languages:

- **Syntax**: VHDL is more verbose and strongly typed, while Verilog is shorter and more C-like in its syntax.
- **Data Types**: VHDL supports a broader range of complex data types, while Verilog uses simpler types such as `wire` and `reg`.
- **Simulation**: VHDL is typically favored for simulation-driven designs due to its strong type-checking, while Verilog is often preferred for hardware synthesis due to its concise syntax.

2.5 Writing and Simulating VHDL and Verilog Code

To design and verify your FPGA circuits, you will need to simulate your VHDL or Verilog code. Simulation helps detect errors and verify functionality before synthesizing the design onto an FPGA.

2.5.1 Simulation Tools

- **ModelSim**: A popular simulator for both VHDL and Verilog.

- **Vivado**: Xilinx's tool suite that supports simulation, synthesis, and implementation of FPGA designs.

- **Quartus**: Intel's design tool suite for FPGA designs, including simulation capabilities.

2.5.2 Writing Testbenches

A **testbench** is a piece of code written in VHDL or Verilog to simulate and verify the functionality of a design. It provides stimulus to the design and checks the output.

VHDL Testbench Example:

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

ENTITY TB_AND_GATE IS
END ENTITY TB_AND_GATE;

ARCHITECTURE behavior OF TB_AND_GATE IS
  SIGNAL A, B : STD_LOGIC := '0';
  SIGNAL Y : STD_LOGIC;
  COMPONENT AND_GATE
    PORT ( A : IN STD_LOGIC; B : IN STD_LOGIC; Y : OUT STD_LOGIC);
  END COMPONENT;
BEGIN
  uut: AND_GATE PORT MAP (A => A, B => B, Y => Y);

  stim_proc: PROCESS
  BEGIN
    A <= '0'; B <= '0'; WAIT FOR 10 ns;
    A <= '1'; B <= '0'; WAIT FOR 10 ns;
    A <= '0'; B <= '1'; WAIT FOR 10 ns;
    A <= '1'; B <= '1'; WAIT FOR 10 ns;
    WAIT;
  END PROCESS;
END ARCHITECTURE behavior;
```

-

Verilog Testbench Example:

```
 module TB_AND_GATE;
  reg A, B;
  wire Y;
```

```
  AND_GATE uut (A, B, Y);

  initial begin
    A = 0; B = 0;
    #10 A = 1; B = 0;
    #10 A = 0; B = 1;
    #10 A = 1; B = 1;
    #10 $finish;
  end
endmodule
```

-

2.6 Conclusion

In this chapter, we have explored the fundamentals of both **VHDL** and **Verilog**, including their syntax, basic structure, data types, and key components. By gaining proficiency in these languages, you will be able to describe and implement digital circuits on FPGA platforms effectively. Understanding how to write and simulate VHDL and Verilog code is crucial for FPGA design, and as you progress, you will gain the ability to handle more complex designs and optimizations.

With a solid foundation in both VHDL and Verilog, you will be well-prepared to tackle real-world FPGA design challenges and contribute to cutting-edge digital systems.