A Tutorial on VHDL and FPGA

By

Shubhajit Roy Chowdhury

Contents

Chapter 1

Introduction to VHDL

Introduction

VHDL stands for **VHSIC Hardware Description Language**, and VHSIC in turn stands for **Very High Speed Integrated Circuits**. VHDL is an acronym for Very High Speed Integrated Circuit Hardware Description Language which is a programming language used to describe a logic circuit by function, data flow behavior, or structure.

VHDL is a programming language: although VHDL was not designed for writing general purpose programs, you can write any algorithm with the VHDL language. If you are able to write programs, you will find in VHDL features similar to those found in procedural languages such as C or Pascal or Ada. VHDL derives most of its syntax and semantics from Ada. Knowing Ada is an advantage for learning VHDL (it is an advantage in general as well).

However, VHDL was not designed as a general purpose language but as an HDL (hardware description language). As the name implies, VHDL aims at modeling or documenting electronics systems. Due to the nature of hardware components which are always running, VHDL is a highly concurrent language, built upon an event-based timing model.

Like a program written in any other language, a VHDL program can be executed. Since VHDL is used to model designs, the term simulation is often used instead of execution, with the same meaning.

Like a program written in another hardware description language, a VHDL program can be transformed with a synthesis tool into a netlist, that is, a detailed gate-level implementation.

Let us start our course in VHDL with a simple program:

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

entity ha is
port        (a, b   : in std_logic;
             s, c : out std_logic);
end entity ha;

architecture df of ha is
begin
  s<=a xor b;
  c<=a and b;
end architecture behave;
```

The program cited above is used to model a half adder using basic AND and XOR gates. VHDL is one of the HDL languages available in the industry for designing the Hardware. VHDL allows us to design a Digital design at Behavior Level, Register Transfer Level (RTL), Gate level and at switch level. VHDL allows hardware designers to express their designs with behavioral constructs, deferring the details of implementation to a later stage of design in the final design.

History of VHDL

VHDL was born out of the United States government's Very High Speed Integrated Circuits (VHSIC/ program. In the course of this program, it became clear that there was a need for a standard language for describing the structure and function of integrated circuits (ICs). Hence the VHSIC Hardware Description Language (VHDL) was developed. It was subsequently developed

further under the auspices of the Institute of Electrical and Electronic Engineers (IEEE) and adopted in the form of the IEEE Standard 1076, Standard VHDL Language Reference Manual, in 1987. This first standard version of the language is often referred to as VHDL-87.

Like all IEEE standards, the VHDL standard is subject to review at least every five years. Comments and suggestions from users of the 1987 standard were analyzed by the IEEE working group responsible for VHDL, and in 1992 a revised version of the standard was proposed. This was eventually adopted in 1993, giving us VHDL-93. A further round of revision of the standard was started in 1998. That process was completed in 2001. In 2002, the current version of VHDL (IEEE-1076-2002) was adopted.

Design Styles

VHDL like any other hardware description language, permits the designers to design a design in either Bottom-up or Top-down methodology.

Bottom-Up Design

The traditional method of electronic design is bottom-up. Each design is performed at the gate-level using the standard gates (Refer to the Digital Section for more details) With increasing complexity of new designs this approach is nearly impossible to maintain. New systems consist of ASIC or microprocessors with a complexity of thousands of transistors. These traditional bottom-up designs have to give way to new structural, hierarchical design methods. Without these new design practices it would be impossible to handle the new complexity.

Top-Down Design

The desired design-style of all designers is the top-down design. A real top-down design allows early testing, easy change of different technologies, a structured system design and offers many other advantages. But it is very difficult to follow a pure top-down design. Due to this fact most designs are mix of both the methods, implementing some key elements of both design styles. Figure 1 shows a Top-Down design approach.

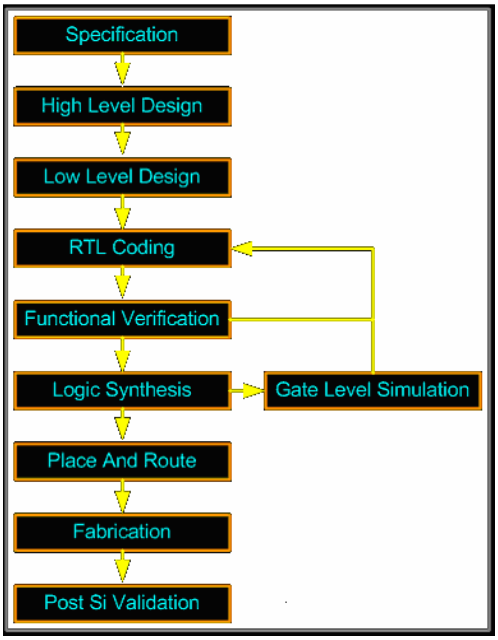

Figure 1. Top down design approach

Abstraction Levels of VHDL

VHDL supports a design at many different levels of abstraction. Three of them are very important:

- Behavioral level
- Register-Transfer Level
- Gate Level

Behavioral level

This level describes a system by concurrent algorithms (Behavioral). Each algorithm itself is sequential, that means it consists of a set of instructions that are executed one after the other. Functions, Tasks and Always blocks are the main elements. There is no regard to the structural realization of the design.

Register-Transfer Level

Designs using the Register-Transfer Level specify the characteristics of a circuit by operations and the transfer of data between the registers. An explicit clock is used. RTL design contains exact timing possibility, operations are scheduled to occur at certain times. Modern definition of a RTL code is "Any code that is synthesizable is called RTL code".

Gate Level

Within the logic level the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (AND, OR, NOT etc gates). *Using gate level modeling might not be a good idea for any level of logic design. Gate level code is generated by tools like synthesis tools and this netlist is used for gate level simulation and for backend.*

Chapter 2

VHDL as a Modeling Language

Introduction

VHDL is popularly used to describe a model for a digital hardware device. This model specifies the external view of the device and one or more internal views. The internal view of the device specifies the functionality or the structure, while external view specifies the interface of the device through which it communicates with other models in its environment. Figure 2.1 shows the hardware device and the corresponding software model.

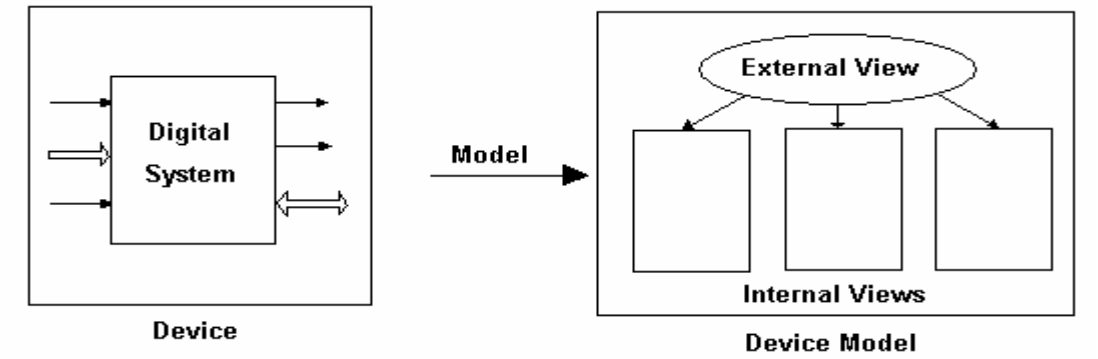

Figure 2.1 Device versus device model

The device to device model mapping is strictly one to many. That is, a hardware device may have many device models. In VHDL, device is treated as a distinct representation of a unique device called entity. The internal details of an entity are specified by an architecture body using any of the following modeling styles:

1. As a set of interconnected components (to represent structure)
2. As a set of concurrent assignment statements (to represent data flow)
3. As a set of sequential assignment statements (to represent behavior)
4. As any combination of the above.

Figure 2.2 shows the VHDL view of a hardware device that has multiple device models, with each device model representing one entity.

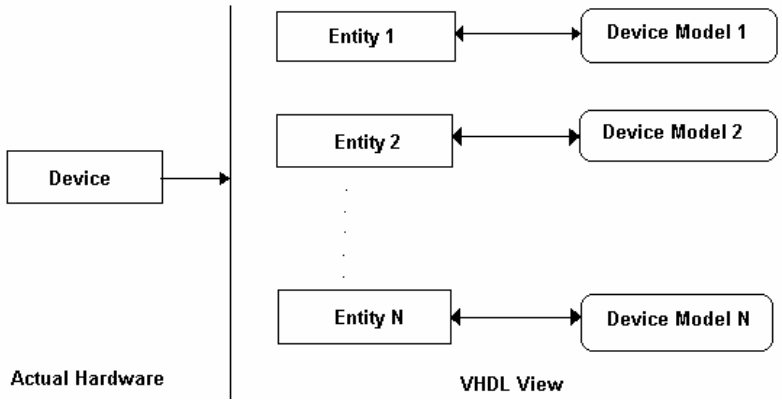

Figure 2.2 VHDL view of a device

Even though entities 1 through N represent N different entities from the VHDL point of view, in reality they represent the same hardware device. The entity is thus a hardware abstraction of the actual hardware device. Each entity is described using one model, which contains one external view and one or more internal views. At the same time, the hardware device may be represented by one or more entities.

VHDL, indeed, is a strongly-typed language for describing a digital hardware design. The structure of a VHDL design resembles the structure of a modern, object-oriented software design in the sense that every VHDL design describes both an external interface and an internal implementation.

A VHDL design consists of the following specifications:

- Entity
- Architecture
- Configuration

Entity

The entity is a specification of the external interface to the design. The specification of the external interface to the design is unique. The entity gives a "black box" description of the design.

Architecture

The architecture is a specification of the internal implementation of the design. There can be several specifications of the internal implementation of the design.

Configuration

The configuration is a specification of the mapping between an architecture and a particular instance of an entity. There must be a configuration for each instance of an entity. The configuration defaults to the last compiled architecture if one has not been explicitly specified.

Chapter 3

The Language of VHDL

Introduction

In this chapter we shall study the language constructs of VHDL.

Datatypes

VHDL supports a set of built-in data types as well as user-defined datatypes**.**

BIT

It represents a Boolean value of 0 or 1.

BIT_VECTOR

It refers to an array of bits.

INTEGER

It refers to an integer value (within a range).

STD_ULOGIC

The STD_ULOGIC values are:

Value	Meaning
'U'	Uninitialized
'X'	Forcing Unknown
'-'	Don't care
'Z'	High Impedance
'1'	Forcing 1
'0'	Forcing 0
'W'	Weak Unknown
'L'	Weak 0
'H'	Weak 1

STD_ULOGIC_VECTOR

It represents an array of STD_ULOGIC.

STD_LOGIC

It is a resolved version of STD_ ULOGIC. If two or more distinct STD LOGIC are driven onto a bus in a design at one instant in time, the bus will have a defined value (usually X).

STD_LOGIC_VECTOR

It refers to an array of STD_LOGIC.

NOTE: IEEE recommends the use of STD_LOGIC and STD_LOGIC_VECTOR.

Entity declaration

The entity declaration specifies the following:

- Name of the entity
- Set of port declarations defining the inputs and outputs to the digital hardware design.
- Port name consists of letters, digits, and/or underscores
- Port name must begin with a letter
- Port name is case insensitive
- Port directions:
  - IN Input port
  - OUT Output port
  - INOUT Bidirectional port
  - BUFFER Buffered output port
  - LINKAGE Deprecated in IEEE 1076-2000
- Port signal type :
  - STD LOGIC
  - STD LOGIC VECTOR(max DOWNTO min)

Signals

In VHDL, signals are used to convey information between (and within) entities. Signals represent connection points in a VHDL design.

Sample Signal Specifications

```
SIGNAL x1 : BIT;
SIGNAL c : BIT_VECTOR(1 TO 4);
SIGNAL byte : BIT_VECTOR(7 DOWNTO 0);
SIGNAL din0 : STD_LOGIC;
SIGNAL dinA : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL clk : STD_LOGIC;
```

Signals are associated with a value and a set of attributes. Attributes allow VHDL designers to check for signal transitions, model delays, and create ”generic” descriptions of entities. Attributes are often used during simulation.

Signal attributes

Signal Attribute	Meaning
clk’event	The attribute is true if an event just happened on signal clk
clk’delayed(T)	Creates a signal same as clk but delayed by a time T

Signal operators

Logical Operators :

• AND
• OR
• XOR
• XNOR
• NOT
• NAND
• NOR

Relational Operators :

• Equal =
• Not Equal /=

• Less Than <
• Greater Than >

Accessing Vector Elements :
some_signal(0) <= other_signal(1);

Signal Assignments

```
SIGNAL x, y, z : STD_LOGIC;
SIGNAL a, b, c : STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL sel : STD_LOGIC_VECTOR(2 DOWNTO 0);

-- Concurrent Signal Assignment Statements
-- NOTE: Both x and a are produced concurrently
x <= y AND z;
a <= b OR c;

-- Alternatively, signals may be assigned constants
x <= '0';
y <= '1';
z <= 'Z';
a <= "00111010"; -- Assigns 0x3A to a
b <= X"3A"; -- Assigns 0x3A to b
c <= X"3" & X"A"; -- Assigns 0x3A to c


SIGNAL x, y, z : STD_LOGIC;
SIGNAL a, b, c : STD_LOGIC_VECTOR( 7 downto 0);
SIGNAL sel : STD_LOGIC_VECTOR( 2 downto 0);

-- Conditional Assignment Statement
-- NOTE: This implements a tree structure of logic gates!
x <= '0' WHEN sel = "000" ELSE
y WHEN sel = "011" ELSE
z WHEN x = '1' ELSE
'1';

-- Selected Signal Assignment Statement
-- NOTE: The selection values must be constants.
WITH sel SELECT
x <= '0' WHEN "000",
y WHEN "011",
z WHEN "100",
'1' WHEN OTHERS;

-- Selected signal assignments also work with vectors.
WITH x SELECT
a <= "01010101" WHEN '1',
b WHEN OTHERS;
-- NOTE: Conditional assignment statements are evaluated
-- in (priority) order while selected signal assignment
-- statements are evaluated in parallel.
```

Process

A process is a fundamental concept of VHDL. The process statement allows a VHDL designer to describe the behavior of a portion of an architecture. It is easy to fall into the trap of believing that statements within a process are sequential. This is not (necessarily) the case! The following is a quote from the Brown and Vranesic text...

"The tendency for the novice is to write code that resembles a computer program, containing many variables and loops. It is difficult to determine what logic circuit the CAD tools will produce when synthesizing such code. ... A good general guideline is to assume that if the designer cannot readily determine what logic circuit is described by the VHDL code, then the CAD tools are not likely to synthesize the circuit that the designer is trying to describe."

Process Specific Statements

Variables :
Variables can be declared within a process. Variables can be used like signals.
However, variables and signals have a subtle difference with respect
to the assignment of a value.
VARIABLE variable_name : variable_type;

IF Statements :
IF condition1 THEN
statements1;
ELSIF condition2 THEN
statements2;
ELSE
statements3;
END IF;

Case Statements :
CASE variable_name IS
WHEN value1 =>
statement1;
WHEN value2 =>
statement2;
WHEN OTHERS =>
statement4;
END CASE;

Loop Statements :
WHILE boolean_expression LOOP
statement;
-- More statements could be added here.
END LOOP;

Wait Statements :
Wait statements halt the execution of a process until a desired event or delay has happened. Wait statements can be used to build sequential designs.

Process usage in combinational logic system

```
SIGNAL a, b, c : STD_LOGIC;
SIGNAL Sel, x, y, z : STD_LOGIC;
SIGNAL DV : STD_LOGIC_VECTOR(17 DOWNTO 0);
-- Assume that a, b, c and Sel are set on elsewhere

PROCESS (a, b, c)
```

```
-- a, b, and c are in the sensitivity list.
-- In this case (combinational) they are
-- inputs.
BEGIN
-- Signal assignment statements can go anywhere
-- inside a process.
x <= a;

-- If statements
IF a = '0' THEN
x <= c AND b;
y <= c OR b;
ELSIF b = '0' THEN
z <= c OR a;
ELSE
w <= '0';
END IF; -- Note x, y, and z are
-- not always assigned
-- in this example

-- Case statement
CASE Sel IS
WHEN '0' =>
z <= a AND b;
WHEN OTHERS
z <= a OR b;

-- For Loops
FOR i IN 1 to 4 LOOP
DV(i) <= a XOR b;
END LOOP;

-- While Loops
WHILE boolean_expression LOOP
statement;
-- More statements could be added here.
END LOOP;
END PROCESS;
```

Process usage in sequential logic system

```
-- Assume we have defined ports and signals such that:
-- d is the flip-flop input pin
-- reset is the flip-flop reset pin
-- clock is the flip-flop clock pin
-- q is the flip-flop output pin
ff1: PROCESS(clock,reset)
BEGIN
IF (reset = '1') THEN
q <= '0';
ELSIF (clock = '1') AND (clock'EVENT) THEN
q <= d;
END IF;
END PROCESS ff1;
```

Chapter 4

Different styles of modeling using VHDL

Introduction

As discussed earlier, VHDL is used for modeling digital hardware. Traditionally, there are three styles of modeling using VHDL. They are:

- Dataflow Modeling
- Behavioral Modeling
- Structural Modeling

To understand the three styles of modeling we take up the simple example of a half adder. The entity of the half adder is declared as:

```
entity ha is
port (a,b:in std_logic;
s,c:out std_logic);
end ha;
```

The entity declaration is same in the three styles of modeling. But the architecture block varies from one style of modeling to another.

Dataflow Modeling

To understand the dataflow behavior of the half adder we consider the following dataflow graph:

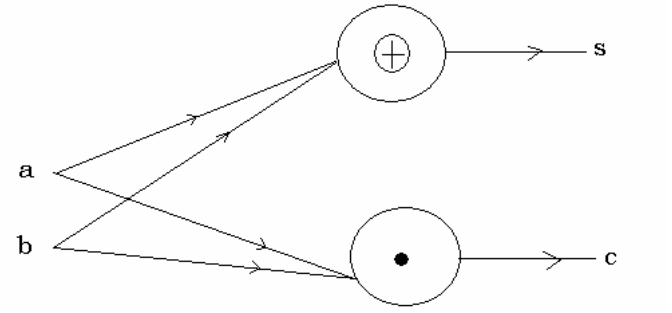

Figure 4.1 Dataflow graph of a half adder

Considering the dataflow graph, the architecture of the half adder using dataflow style of modeling is as follows:

```
architecture df of ha is
begin
s<=a xor b;
c<=a and b;
end df;
```

In dataflow modeling, the architecture is expressed by concurrent signal assignment statements.

Behavioral Modeling

The behavior of the half adder is expressed by the following flowchart:

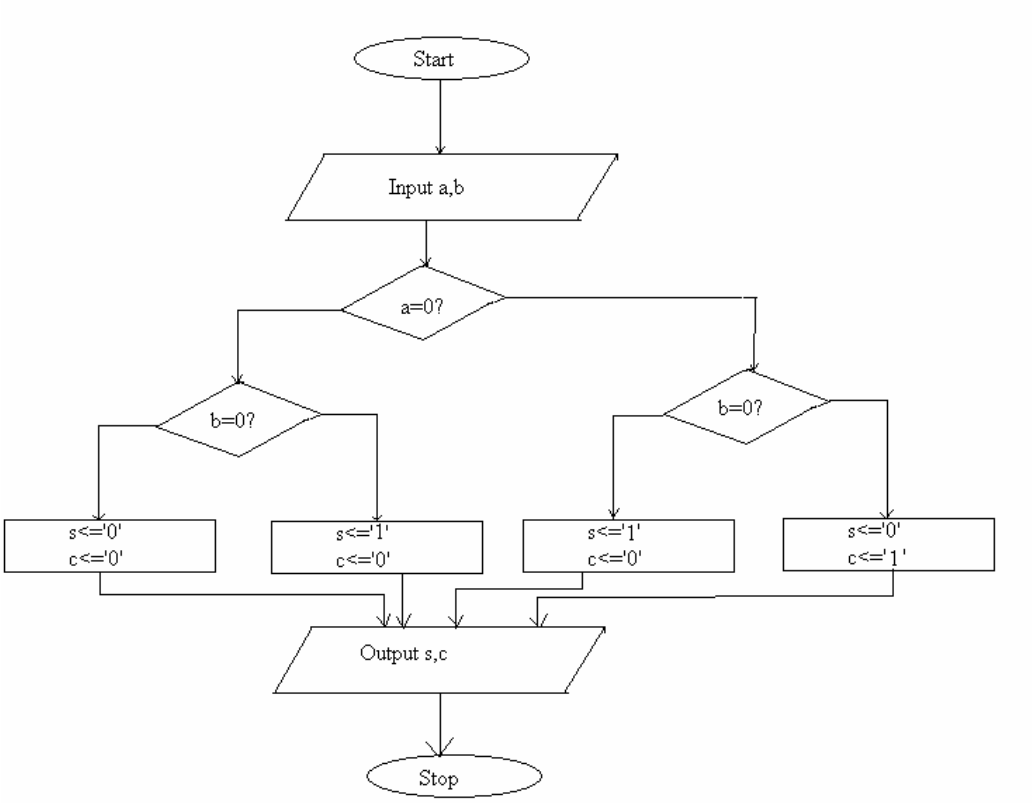

Figure 4.2 Flowchart of the half_adder

From the flowchart we can visualize the half adder as a sequential execution of instructions. The half adder is behaviorally modeled as:

```
architecture behave of ha is
   begin
   process(a,b)
   if (a='0')then
      c<='0';
   if(b='0')then
     s<='0';
     elsif(b='1')then
     s<='1';
    end if;
elsif(a='1')then
    if(b='0')then
       s<='1';
       c<='0';
    else
       s<='1';
       c<='1';
  end behave;
```

In behavioral modeling, the architecture is visualized as a group of concurrent processes, where the statements within each process are executed sequentially.

Structural Modeling

For structural modeling, we consider the following logic network of the half adder:

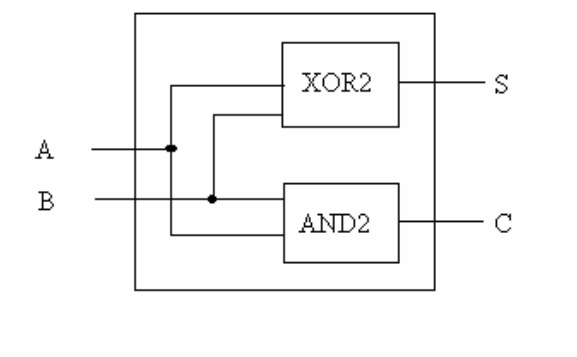

Figure 4.3 Logic Network of half adder

The half adder has two components – a 2 input XOR gate and a 2 input AND gate. The XOR gate and AND gate has previously been described in some other architecture and stored in the library. However, we can browse those components in our present work and use them in our current modeling.

The structural model of the half adder is described by the following architecture:

```
architecture struct of ha is
component AND2 is
port(P,Q:in std_logic;
R:out std_logic);
end component;
component XOR2 is
port(X,Y:in std_logic;
Z:out std_logic);
end component;
begin
X1:XOR2 port map (a,b,s);
A1:AND2 port map (a,b,c);
end struct;
```

In structural modeling, the architecture of the hardware is visualized as an interconnection of its lower level components without any reference to its behavior. Such type of modeling provides an opportunity of modeling the hardware using a bottom up approach.

Mixed Style of Modeling

Sometimes, we may use one or more styles of modeling together to build up the entire hardware. For example we may model the half adder in this way:

```
architecture mixed of ha is
component AND2 is
port(P,Q:in std_logic;
R:out std_logic);
end component;
begin
s<=a xor b;
A1:AND2 port map (a,b,c);
end mixed;
```

Here, we are using a combination of structural and dataflow style of modeling to develop the architecture of the half adder. Since multiple styles of modeling are used together, it is called a mixed style of modeling.

Chapter 5

Modeling Finite State Machines using VHDL

Introduction

State Machines constitute a special modeling technique for sequential circuits. Such a modeling technique can prove to be useful for certain systems especially in those whose tasks form a well defined sequence. A finite state machine is a circuit that has finite number of states and can switch between them if certain conditions exist. In this case, I will be designing a circuit that depends on a single input to determine the state.

A finite state machine has many applications. Lets say you want to design a circuit that unlocks a door when a certain password is entered on a keypad. Such a design would require a finite state machine. Typically, finite state machines are used for modeling the behavior of controllers.

Finite State Machines fall into two fundamental categories:

- Mealy Machine
- Moore Machines

Mealy Machine

Mealy machines are generalized finite state machines where the output at any instant of time depends on the present state and present input.

Moore Machine

Moore machines are a restricted type of finite state machines, where the output at any instant of time depends only on the present state and not on the present input.

A VHDL Modeling example of a Finite State Machine

In this case, we are going to design a circuit that sets an output, z, to high when it detects a sequence of 0-1-1-0 on the input, x. The circuit will be synchronized.

Here is the actual VHDL code. You can easily modify this code to include more states, have more outputs, and even have parallel input values. To modify the number of states you would have to modify the number of statetypes declared and also modify the number of if statement blocks in the process.

```
library ieee;
use ieee.std_logic_1164.all;
entity finitestatemachine is
port
( reset, clk, x : in std_logic;
z : out std_logic );
end finitestatemachine;

architecture behavioral of finitestatemachine is
-- The following is an example of a user defined "enumerated" data type
type statetype is (s0, s1, s2,s3,s4);
signal state, next_state : statetype := s0; -- -- Declaration of two signals of this type
begin
comb_proc : process (state, x)
begin
case state is
when s0 =>
if x = '0' then
```

```
next_state <= s1;
z <= '0';
else
next_state <= s0;
z <= '0';
end if;
when s1 =>
if x = '1' then
next_state <= s2;
z <= '0';
else
next_state <= s1;
z <= '0';
end if;
when s2 =>
if x = '1' then
next_state <= s3;
z <= '0';
else
next_state <= s1;
z <= '0';
end if;
when s3 =>
if x = '1' then
next_state <= s0;
z <= '0';
else
next_state <= s4;
z <= '0';
end if;
when s4 =>
if x = '0' then
next_state <= s1;
z <= '1';
else
next_state <= s2;
z <= '1';
end if;
end case;
end process;
clk_proc : process
begin
wait until (clk'event and clk = '1'); -- wait for rising edge of clk
if reset = '1' then
state <= s0;
else
state <= next_state;
end if;
end process;
end behavioral;
```

Chapter 6

Subprograms: Procedures and Functions

Introduction

Writing subprograms is an essential part of learning a language as it saves the time and effort of the programmer to write repetitive sequence of the same block of code. There are two kinds of subprograms: procedures and functions. Both procedures and functions written in VHDL must have a body and may have declarations.

Procedures

Procedures perform sequential computations and return values in global objects or by storing values into formal parameters.

Functions

Functions perform sequential computations and return a value as the value of the function. Functions do not change their formal parameters.

Subprograms may exist as just a procedure body or a function body. Subprograms may also have a procedure declarations or a function declaration.

When subprograms are provided in a package, the subprogram declaration is placed in the package declaration and the subprogram body is placed in the package body.

Procedures

Procedure Declaration

Procedure declaration is used to declare the calling interface to a procedure.

```
procedure identifier [ ( formal parameter list ) ] ;
```

```
procedure print_header ;
procedure build ( A : in constant integer;
                  B : inout signal bit_vector;
                  C : out variable real;
                  D : file ) ;
```

Formal parameters are separated by semicolons in the formal parameter list. Each formal parameter is essentially a declaration of an object that is local to the procedure. The type definitions used in formal parameters must be visible at the place where the procedure is being declared. No semicolon follows the last formal parameter inside the parenthesis. Formal parameters may be constants, variables, signals or files. The default is variable. Formal parameters may have modes in, inout and out. Files do not have a mode. The default is in. If no type is given and a mode of in is used, constant is the default.

The equivalent default declaration of "build" is

```
procedure build ( A : in integer;
                  B : inout signal bit_vector;
                  C : out real;
                  D : file ) ;
```

Procedure Body

Procedure body is used to define the implementation of the procedure.

```
procedure identifier [ ( formal parameter list ) ] is
   [ declarations, see allowed list below ]
begin
   sequential statement(s)
end procedure identifier ;
```

```
procedure print_header is
   use STD.textio.all;
   variable my_line : line;
begin
   write ( my_line, string'("A    B    C"));
   writeline ( output, my_line );
end procedure print_header ;
```

The procedure body formal parameter list is defined above in Procedure Declaration. When a procedure declaration is used then the corresponding procedure body should have exactly the same formal parameter list.

Functions

Function Declaration

Function declaration is used to declare the calling and return interface to a function.

```
function identifier [ ( formal parameter list ) ] return a_type ;
```

```
function random return float ;
function is_even ( A : integer) return boolean ;
```

Formal parameters are separated by semicolons in the formal parameter list. Each formal parameter is essentially a declaration of an object that is local to the function. The type definitions used in formal parameters must be visible at the place where the function is being declared. No semicolon follows the last formal parameter inside the parenthesis. Formal parameters may be constants, signals or files. The default is constant.

Formal parameters have the mode **in**. Files do not have a mode. Note that **inout** and **out** are not allowed for functions. The default is **in**.

The reserved word **function** may be preceded by nothing, implying **pure**, **pure** or **impure**. A **pure function** must not contain a reference to a file object, slice, subelement, shared variable or signal with attributes such as 'delayed, 'stable, 'quiet, 'transaction and must not be a parent of an impure function.

Function Body

Function body is used to define the implementation of the function.

```
function identifier [ ( formal parameter list ) ]
            return a_type is
   [ declarations, see allowed list below ]
begin
```

```
    sequential statement(s)
    return some_value; -- of type  a_type
end function identifier ;

function random return float is
    variable X : float;
begin
    -- compute X
    return X;
end function random ;
```

The function body formal parameter list is defined above in Function Declaration. When a function declaration is used then the corresponding function body should have exactly the same formal parameter list.

Chapter 7

Introduction to FPGA

Introduction

FPGA stands for **F**ield **P**rogrammable **G**ate **A**rray. There are many forms of devices which are field programmable. These are PAL, PLD, CPLD, and FPGA. These devices differ on their granularity, how the programming is accomplished etc. PAL, PLA and CPLD devices are usually smaller in capacity but more predictable in timing and they can be implemented with Sum-of-Products, Product-of-Sums or both. FPGA devices can be based on Flash, SRAM, EEPROM or Anti-Fuse connectivity. The most successful FPGA devices are based on SRAM. This is because all other memory types are much less dense in terms of area than SRAM. Also some types of connectivity are One-Time Programmable (i.e. Anti-Fuse) so they are not very flexible. SRAM based FPGAs have no maximum erase cycle limitations either.

Synthesis

FPGA development is in some sense similar to ASIC development. One can talk about Front-End Tools which can be Schematic Entry or an HDL (Hardware Description Language). Most common HDLs used for FPGA Design are Verilog and VHDL. After describing an FPGA design in an HDL, a tool called a Synthesizer which effectively converts Verilog, VHDL into the specific primitives which exist an an FPGA family. SRAM based FPGAs have Lookup-Tables which can be programmed to implement any function of N variables (usually 4~5) and Flip-Flops which can be programmed to implement different types of storage (JK, T, Latch, DFF with set and/or reset etc).

After a device level netlist is generated with synthesis, one uses a back-end tool called Place & Route which is most of the time supplied by the device vendor (i.e. Xilinx or Altera). In contrast synthesis tools are usually supplied by third party vendors and even the ones packaged with vendors' toolset usually are restricted versions of third party tools. Most popular synthesis tools come from Synplicity, Exemplar, Cadence and Synopsys. DSPIA Inc. suggests Synplicity for FPGA development because of their high quality tools.

Most synthesis tools understand only a subset of HDLs which are synthesizable subsets. Up until recently these subsets weren't standardized and varied from vendor to vendor. These days there are efforts under way to standardize synthesizable Verilog and VHDL code.

From the P&R tool, one obtains a file which can be used to download onto an FPGA to program it with the hardware design described by the original HDL code. This download can be either done through a serial connection, a JTAG cable or programmed into a ROM and loaded into the FPGA every time power is applied. This is very roughly the flow of FPGA development..

Field Programmable Gate Array (FPGA) is a semiconductor device containing programmable logic components and programmable interconnects. The programmable logic components can be programmed to duplicate the functionality of basic logic gates (such as AND, OR, XOR, NOT) or more complex combinatorial functions such as decoders or simple math functions. In most FPGAs, these programmable logic components (or logic blocks) also include memory elements, which may be simple flip-flops or more complete blocks of memories.

Architecture of an FPGA

A typical SRAM based FPGA consists of three major types of elements:

- Combinational logic

- Interconnect
- I/O pins

Figure 6.1 shows the basic architecture of FPGA that incorporates these three elements.

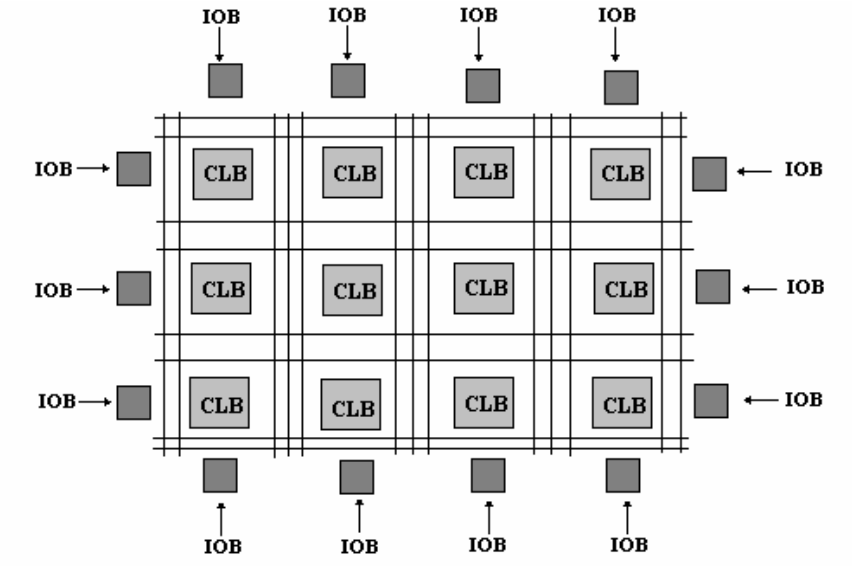

Figure 5.1 Basic Architecture of FPGA

The combinational logic is divided into realtively small units which may be know as logic elements (LEs) or combinational logic blocks(CLBs). The LE or CLB can usually form the function of typical logic gates but it is still small compared to the typical combinational logic block found in a large design. The interconnects are made between the logic elements using programmable interconnect. The interconnects may be logically organized into channels or other units. FPGA typically offer several types of interconnect depending on the distance between the combinational logic blocks that are to be connected; clock signals are also provided with their own interconnection networks. I/O pins may be referred to as I/O blocks (IOBs). They are generally programmable to be inputs or outputs and often provide other features such as low power or hich speed connections. Multiple I/O pads may fit into the height of one row or the width of one column. An application circuit must be mapped into an FPGA with adequate resources.

A typical FPGA combinational logic block consists of a 4-input lookup table (LUT), and a flip-flop, as shown in figure 6.2 below:

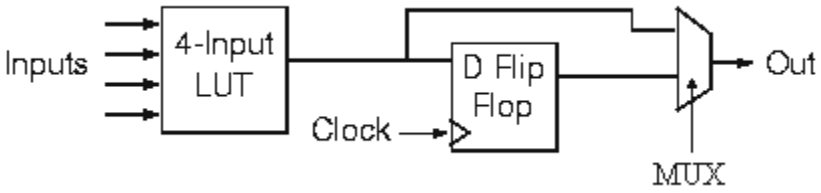

Figure 6.2 Structure of a Combinational Logic block

There is only one output, which can be either the registered or the unregistered LUT output. The logic block has four inputs for the LUT and a clock input. Since clock signals (and often other high-fanout signals) are normally routed via special-purpose dedicated routing networks in commercial FPGAs, they are accounted for separately from other signals.

In a typical CLB, the locations of the FPGA logic block pins are shown in figure 6.3 below:

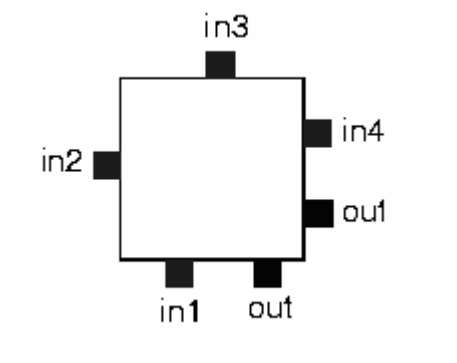

Figure 6.3 Logic Block Pin Locations

Each input is accessible from one side of the logic block, while the output pin can connect to routing wires in both the channel to the right and the channel below the logic block. Each logic block output pin can connect to any of the wiring segments in the channels adjacent to it.

Similarly, an I/O block can connect to any one of the wiring segments in the channel adjacent to it. For example, an I/O block at the top of the chip can connect to any of the W wires (where W is the channel width) in the horizontal channel immediately below it.

Generally, the FPGA routing is unsegmented. That is, each wiring segment spans only one logic block before it terminates in a switch box. By turning on some of the programmable switches within a switch box, longer paths can be constructed. For higher speed interconnect, some FPGA architectures use longer routing lines that span multiple logic blocks.

Whenever a vertical and a horizontal channel intersect there is a switch box. A typical switch box topology is depicted in figure 5.4. In this architecture, when a wire enters a switch box, there are three programmable switches that allow it to connect to three other wires in adjacent channel segments. The pattern, or topology, of switches used in this architecture is the planar or domain-based switch box topology. In this switch box topology, a wire in track number one connects only to wires in track number one in adjacent channel segments, wires in track number 2 connect only to other wires in track number 2 and so on. The figure below illustrates the connections in a switch box.

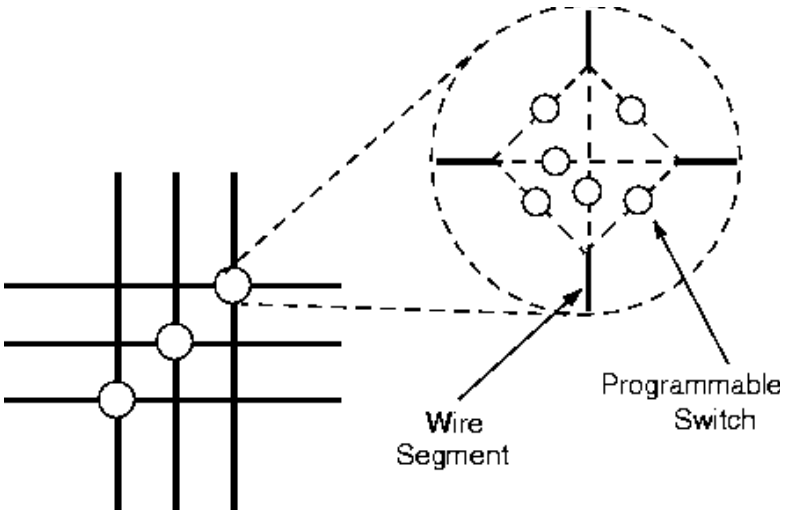

Figure 6.4 Switch box topology at the intersection of horizontal and vertical channels

Modern FPGA families expand upon the above capabilities to include higher level functionality fixed into the silicon. Having these common functions embedded into the silicon reduces the area required and gives those functions increased speed compared to building them from primitives. Examples of these include multipliers, generic DSP blocks, embedded processors, high speed IO logic and embedded memories.

To define the behavior of the FPGA the user provides a hardware description language (HDL) or a schematic design. Common HDLs are VHDL and Verilog. Then, using an electronic design automation tool, a technology-mapped netlist is generated. The netlist can then be fitted to the actual FPGA architecture using a process called place-and-route, usually performed by the FPGA company's proprietary place-and-route software. The user will validate the map, place and route results via timing analysis, simulation, and other verification methodologies. Once the design and validation process is complete, the binary file generated (also using the FPGA company's proprietary software) is used to (re)configure the FPGA device.