ADDIS ABABA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

FACULTY OF TECHNOLOGY

# DESIGN OF PULSE WIDTH MODULATION (PWM) AND IT'S IMPLEMENTATION IN XILINX FIELD PROGRAMMABLE GATE ARRAY (FPGA)

A thesis submitted to the School of Graduate Studies of Addis Ababa University in partial fulfillment of the Degree of Masters of Science in Electrical Engineering

By

**Teshome Ababu**

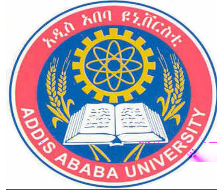
Advisor

Prof. Dr.Gerald Higelin

Addis Ababa

Ethiopia

February, 2007



**ADDIS ABABA UNIVERSITY**

**SCHOOL OF GRADUATE STUDIES**

**FACULTY OF TECHNOLOGY**

# DESIGN OF PULSE WIDTH MODULATION (PWM) AND IT'S IMPLEMENTATION IN XILINX FIELD PROGRAMMABLE GATE ARRAY (FPGA)

BY

**Teshome Ababu**

Approval by Board of Examiners

_Dr. Mengesha Mamo________________ ________________

Chairman (Department of graduate committee) Signature

Prof.Dr.Gerald Higelin ________________

Advisor Signature

Dr. Eneyew Adugna ________________

Interna Examiner Signature

Dr. Goshal ________________

External Examiner Signature

## DECLARATION

I, the undersigned, hereby declare that this thesis is my original work performed under the supervision of Prof. Dr.Gerald Higelin has not been presented as a thesis for a degree program in any other university and all sources of materials used for the thesis are duly acknowledged.

Name: Teshome Ababu

Signature: -------------

Place: Addis Ababa

Date of submission: ---------------------

This thesis has been submitted for examination with our approval as university advisor.

Prof. Dr.Gerald Higelin ----------------------

Signature

Addis Ababa

Ethiopia

February 2007

# Acknowledgments

First and foremost, I would like to thank my advisor, Prof. Dr.Gerald Higelin for his inspiration and continuous support of my research, a great combination who is always willing to listen, encourage, and give insightful comments and valuable criticism. He read all the drafts of my thesis and taught me to be thorough in analyzing problems and rigorous in presenting ideas. This thesis would not have been possible without his support and guidance. I also thank my previous supervisor the late Prof. Dr.Santhanam A. who got me started in research.

My gratitude is also conveyed to all my friends, for their cooperation and support during the time I studies.

I am deeply grateful to my parents for their everlasting patience and love.

I also would like to acknowledge the Xlinx FPGA Company for their valuable software and on line help.

Finally, I would like to thank the almighty of God for helping me to complete this thesis. I all of thank you!

# Table of Contents

Page

Acknowledgements ..........................................i

Abstract ..........................................ii

List of Tables ..........................................vi

List of Figures ..........................................vii

Abbreviations ..........................................ix

Chapter 1: Introduction

1.1 Motivation..........................................1

1.2 Objective of the thesis..........................................2

1.3 Thesis organization..........................................3

Chapter 2: Literature Review

2.1 Pulse Width Modulation..........................................5

2.2 Digital technology..........................................5

2.3 PWM Technique ..........................................7

Chapter 3: Programmable Logic Devices and VHDL

3.1 The History of Programmable Logic..........................................11

3.2 Simple Programmable Logic Devices (SPLDs)..........................................13

3.3 Complex Programmable Logic Devices (CPLDs)..........................................13

3.4 Field Programmable Gate Arrays (FPGAs)..........................................15

3.4.1 FPGAs Architectures..........................................16

3.4.2 Xilinx SRAM-based FPGAs……………………………………………………..18
3.4.3 Xilinx devices……………………………………………………………….21
3.4.4 Slices and CLBs……………………………………………….22
3.4.5 Advantage of FPGA……………………………………………………….25
3.5 HDL…………………………………………………………………………..26
3.6 VHDL………………………………………………………………………...27
3.6.1 The History of VHDL…………………………………………………...27
3.6.2 Design units……………………………………………………………..31
3.6.3 Levels of abstraction……………………………………………………35
3.6.4 Signals and Variables…………………………………………………...36
3.6.5 Test benches……………………………………………………………..37
3.6.6 Advantages of VHDL……………………………………………………38

## Chapter 4: Design of PWM in Xilinx FPGA

4.1 Xilinx ISE Overview……………………………………………………….39
4.2 FPGA Design Flow ………………………………………………………...41
**4**.3 Functional description of the PWM design ……………………………...43
4.4 VHDL Modeling of the Functional description of PWM design………………...46

## Chapter 5: Simulation results and discussion

5.1 XST synthesis ……………………………………………………………52
5.2 Viewing a Synthesis Report………………………………………………53
5.3 Simulation Overview……………………………………………………..58

5.4 Test benches …………………………………………………………………...58
5.5 Xilinx Synthesis and simulation results……………………………………..59

5.6 Simulation waveforms…………………………………………………………65

5.7 View technology schematic results……………………………………………67

## Chapter 6: Conclusions & future work

6.1 Conclusions……………………………………………………………..73
6.2 Future work……………………………………………………………..73

Appendix A: VHDL Listing.......................................................................................75
References..................................................................................................................80

# List of Tables

Table Page

5.1Device utilization summary……………………………………………………65

5.2 Some of the Data Values for different Duty Cycles (N=8)………………………65

5.3 PWM user interface signal description…………………………………………...67

# List of Figures

Figure Page

2.1 PWM signal with different duty cycles. . . . . . ……………………………………..7

2.2 PWM Technique……………………………………………………………..8

2.3 Speed control of DC Motor using PWM…………………………………………8

3.1 Digital logic technology…………………………………………………………12

3.2 SPLD Architectures……………………………………………………………13

3.3 CPLD Architecture…………………………………………………………….4

3.4 FPGA Architecture……………………………………………………………16

3.6 Xilinx XC4000 Configurable Logic Block (CLB)…………………………………..19

3.7Xilinx XC4000 Wire Segments…………………………………………………20

3.8. Slices…………………………………………………………………………...22

3.9. Simplified Slice structure………………………………………………………23

3.10. Connecting Look-Up Tables………………………………………………….24

3.11 VHDL block structure…………………………………………………………28

3.12 .VHDL-file structures………………………………………………………….28

3.13 (a) Overview of entity and (b) Architecture syntax…………………………………29

3.14 Overview of architecture declarations……………………………………………29

4.1 FPGA design flow……………………………………………………………….39

4.2 FPGA Design methodology…………………………………………………….41

4.3 functional block diagram of PWM…………………………………………………42

4.4 Pulse as a Function of the Count Value (ds = pwm_data)……………………………44

5.1 XST Design Flow Overview……………………………………………………….52

5.2 XST design flow…………………………………………………………………53

5.3 Waveform 1 pulse with 50% Duty cycle……………………………………………..66

5.4 Waveform 2: pulse with 25% Duty cycle…………………………………………….66
5.5 PWM user interface signal ……………………………………………………...67
5.6. RTL schematic view of PWM…………………………………………………………….68
5.7. (a) and (b) Equivalent Generic schematic of PWM of –MUX 0024-imp…………………69
5.8. (a) and (b) Technology schematic view of PWM…………………………………………71
5.9.The equivalent Look-up Table function generators of LUT 4 INT _OC09 in fig 5.8……..72

# Abbreviations

| | |
|---|---|
| ASIC | Application Specific Integrated Circuit |
| ASSP | Application Specific Standard Product |
| CPLD | Complex Programmable Logic Device |
| CLB | Configurable Logic Block |
| CMOS | Complimentary Metal Oxide Semiconductor |
| FPGA | Field Programmable Gate Array |
| HDL | Hardware description language |
| I/O | Inputs and Outputs |
| ISE | Integrated software environment |
| ISP | In-System Programming |
| IEEE | Institute of Electrical and Electronic Engineers |
| JTAG | Joint Test Action Group |
| LUT | Look Up Table |
| MSI | Midium Scale Integration |
| NGC | Xilinx specific netlist files |
| OTP | One time programmable |
| PAL | Programmable Array Logic device |
| PLA | Programmable Logic Array |
| PLD | Programmable Logic Device |
| PWM | Pulse Width Modulation |
| RAM | Random Access Memory |
| RTL | register transfer level |
| SPLD | Simple Programmable Logic Device |
| SRAM | Static Random Access Memory |
| Tpd | Time of Propagation Delay through the device |
| TTL | Transistor Transistor Logic |

| | |
|---|---|
| UUT | Unit under test |
| VHDL | VHISC High Level Description Language |
| VHSIC | Very High Speed Integrated Circuit |
| XST | Xilinx Synthesis Technology |
| XCF | Xilinx constrains file |

# Abstract

The following thesis describes the design, the synthesis, and the implementation of pulse width modulation (PWM) in Xilinx Field Programmable Gate Array (FPGA). The contribution of this thesis is the development of PWM in Xilinx Integrated System Environment (ISE) CAD tools and The VHDL modeling is used in the design process of PWM.

Pulse width modulation has been widely used in many applications especially in communication and control systems. The paper develops high frequency PWM generator architecture for using FPGA. The resulting FPGA frequency depends on the target FPGA speed grade and the duty cycle resolution requirements.

In most industrial application due to the need of design integration in control systems FPGA based PWM controller is advantageous over the other controller systems like microprocessor, microcontroller and so on.

As geometries shrink and device counts multiply, opportunities abound to do incredible things with in the confines of a single chip (FPGA). Greater focus on design reuse, where earlier design is utilized and reused in later design. The power, compactness and flexibility of the FPGA based controller could be useful in motor control, particularly in robotics where those qualities are important. The FPGA provides advantages over traditional methods such as microcontroller based designs and PLD/ASIC designs by combining the strengths of both. The FPGA allows for implementation of parallel processing for generating the required waveforms. In addition to this the paper describes the architectural features of Xilinx FPGA to the other programmable logic device and explores the design using Very high speed integrated circuit hardware description language (VHDL). The VHDL model was implemented on the Spartan 3e FPGA and optimized for space. The optimized implementation was found to consume 34 numbers of slices, 18 numbers of slice flip flops, 65 numbers of 4 input LUTs and 11 numbers of bonded IOBs.

# Chapter 1

# Introduction

## 1.1 Motivation

As semiconductors increase in density, growing trend toward moving complete systems from the board level to the chip level such as FPGA [1].

In the very early days, modulating currents was made mainly by inserting variable resistor in the path. Obviously, this led to power losses, heavy and extensive ’Controllers’ and poor performance. Heat dissipated implies large more expensive components (power semiconductor and heat sinks). How can we reduce this wasted power? One clever method is a technique called pulse width modulation

Today, thanks to electronic development, there are different means for controlling current with out much power loss and bulky components. One of these methods is the usage of pulse width modulation (PWM) in power electronics to control high energy with maximal efficiency and power saving.

Using pulse width modulation (PWM) in power electronics control system is not new, there are different approaches for developing pulse width modulation. Many digital circuits can generate PWM signals, but what is interesting is, to generate pulse width modulation using Hardware Description Language (VHDL) and implementing it in FPGA. FPGA implementation of PWM is selected because FPGA can process information faster, controller architecture can be optimized for space or speed, available in radiation tolerant package, implementation in VHDL allows the targeting of a variety of commercially available device, FPGA allows for implementation of parallel processing.

A field programmable Gate Array (FPGA) is a large Programmable logic device (PLD) that can be used to implement large logic equations as well as arbitrary combinational and sequential logic circuits. Thus, an FPGA can also be used to implement the control- logic design.

FPGA are programmable semiconductor devices that are based around a matrix of configurable logic block (CLBs) connected via programmable interconnects as opposed to ASICs where the device is custom built for a particular design. FPGA can be programmed to the desired application or functionality requirements.

VHDL is a language that is used to describe the behavior of digital circuit designs. It is VHSIC (Very High Speed Integrated Circuit) Hardware Description Language , and now used extensively by industry and academia for the purpose of simulating and synthesizing digital circuit design. Its designs can be simulated and translated into a form suitable for hardware implementation. VHDL is rapidly being embraced as the universal communication medium of design FPGA vendors, and ASIC vendors thorough the industry is standardizing on VHDL as input and output from their tools.

In this thesis, VHDL modeling is used to generate the PWM signal and XILINX WebPack ISE Design Software is used for the design of PWM. And also XILINX FPGA Spartan 3E family is used for the process, the VHDL language is exhaustively used for PWM generation and the language's capability is explored.

The PWM developed in this way can be integrated into complete system where PWM is required, as in applications requiring motor and motion control, robotics etc. Therefore, most industries and others involve PWM in their control system benefited from this thesis.

## 1.2 Objectives of the thesis

Getting Field Programmable Gate Array based PWM controller is the choice of every controller designer, because of the design flexibility, design complexity, the gate capacity, reconfigure ability, design time, speed, etc and this paper describes the development of PWM in Xilinx FPGA. Many digital and Transistor Transistor Logic circuit (such as microprocessor, microcontroller, etc) can develop PWM, but what is interesting is to design the PWM using the latest Programmable device so as to use the features of FPGA.

### 1.2.1 General objective

- The general objective of this thesis is to Design PWM using XILINX WebPack ISE design software.

### 1.2.2 Specific objective

Specifically this thesis is very important in the area of PWM generation systems

- To Design PWM using VHDL modeling
- To study the architectural features of XILINX FPGAs
- To explore these features using HDL (VHDL)
- Synthesizing and simulating the design using Xilinx ISE simulator & XST
- Implementing the design in FPGA and realizing the design

## 1.3 Thesis Organization

The Design of Pulse Width Modulation in Xilinx Field Programmable Gate Array has a great role in the control system where the pulse width modulation is used. This thesis contains the total of six chapters,

Chapter one is general introduction of a thesis, which highlights the basic idea and gives an insight or a direction to the body of the thesis.

Chapter two focuses on the introduction of pulse width modulation systems, the advantages and the application of it.

Chapter three tells about Programmable Logic Devices and VHDL. This gives an overview of how and where programmable logic devices are used. It gives a brief history of the programmable logic devices and goes on to describe the different ways of designing with PLDs. And also discuses the architecture of the Xilinx FPGA. In addition to this, it explores the VHDL Modeling.

Chapter four: - deals about the design of PWM in Xilinx FPGA, and describes the functional description of the design PWM and the FPGA design flow is explored. This chapter also describes the VHDL modeling of the design PWM in Xilinx Integrated System Environment.

Chapter five develops the Synthesis and simulation results of the VHDL design of PWM in Xilinx Field Programmable Gate Array and discuses the result of simulation and synthesis.

Chapter six describes conclusion on what is done in this project and suggest the recommendations for the future work.

# Chapter 2

# Literature Review

## 2.1 Pulse Width Modulation

Pulse width modulation (PWM) is a technique to provide a logic “1” and logic “0” for a controlled period of time. It is a signal source involves the modulation of its duty cycle to control the amount of power sent to a load [6] .The following sections describe the design of Pulse Width Modulation (PWM) on a Xilinx FPGA using very high speed integrated circuit hardware description language (VHDL). The PWM generates pulses on its output. The pulses are made in such a way that the average value of highs and lows is proportional to the PWM input. By filtering the pulses, we obtain an analog value proportional to the PWM input. A PWM input can be of any width. Most common values are 8-bits and 16-bits. The PWM developed can be used in many diverse and complex applications like robotics, motor and motion control.

## 2.2 Digital Technology

There are two approaches for implementing control systems using digital technology [1]. The first approach is based on software which implies a memory-processor interaction. The memory holds the application program while the processor fetches, decodes, and executes the program instructions. Programmable Logic Controllers (PLCs), microcontrollers, microprocessors, Digital Signal Processors (DSPs), and general purpose computers are tools for software implementation.

On the other hand, the second approach is based on hardware. Early hardware implementation is achieved by magnetic relays extensively used in old industry automation systems. It then became achievable by means of digital logic gates and Medium Scale Integration (MSI) components[1]. When the system size and complexity increases, Application Specific Integrated Circuits (ASICs)

are utilized [2]. The ASIC must be fabricated on a manufacturing line, a process that takes several

months, before it can be used or even tested . FPGAs are configurable ICs and used to implement logic functions [10]. Early generations of FPGAs were most often used as glue logic which is the logic needed to connect the major components of a system. They were often used in prototypes because they could be programmed and inserted into a board in a few minutes, but they did not always make it into the final product. Today's high-end FPGAs can hold several millions gates and have some significant advantages over ASICs [10]. They ensure ease of design, lower development costs, more product revenue, and the opportunity to speed products to market. At the same time they are superior to software-based controllers as they are more compact, power-efficient, while adding high speed capabilities. The target FPGA device used in this paper is Spartan-3 manufactured recently by Xilinx [23]. Digital controllers usually encompass input/output (I/O) modules to communicate with users. Embedded systems typically consist of both application-specific hardware and a general purpose microprocessor or microcontroller. Many of the functions performed by the system can be implemented either on dedicated hardware (for example, in an FPGA, in an ASIC, or in a special-purpose function block added to the microcontroller itself) or can be implemented by the microprocessor in software. The decision to implement a function in hardware or software depends on trade-offs between the hardware/software implementations like cost, speed, power consumption, design time, size (silicon area or program size), risk and others. Partitioning functions efficiently between hardware and software can be key to timely design of high performance, low cost digital systems. The speed of a DC motor is approximately proportional to the supply voltage, so reducing the supply voltage by half will reduce the speed by approximately one-half. The speed of the motor can therefore be controlled by varying the average supply voltage[7]. The supply voltage could be changed using a variable supply voltage source, but this technique is inefficient since voltage is controlled in these cases through a voltage drop across a transistor. Since all current must go through the transistor and P=VI (the drop across the transistor times the current), a significant amount of power is lost at the transistor. A better way to control the motor is to switch the motor's supply on and off very quickly. If the on time is equal to the off time, the average voltage

seen by the motor will equal half the supply voltage and the motor will run at half the maximum speed. As the on time increases compared with the off time, the average speed of the motor will increase. The user should not notice the motor turning on and off, because it is done very quickly. A pulse-width modulated (PWM) signal is a constant period square wave with a varying duty cycle (on-time compared to off-time)[25]. In other words, the frequency of a PWM signal is constant but the time the signal remains high varies as shown in Figure 2.1. The duty cycle (percent ontime) is given by τ/T.

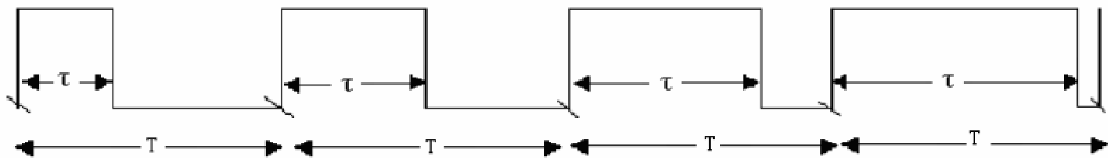



Figure 2.1 PWM signal with different duty cycles

## 2.3 PWM Technique

PWM can be used to reduce the total amount of power delivered to a load without losses normally incurred when a power source is limited by resistive means. This is because the average power delivered is proportional to the modulation duty cycle. High frequency PWM Power control systems are realizable with semiconductor switches. The discrete on/off states of the modulation are used to control the state of the switch(es) which correspondingly control the voltage across or current through the load .The major advantage of this system is the switches are either off and not conducting any current, or on and have (ideally) no voltage drop across them. The product of the current and the voltage at any given time defines the power dissipated by the switch, thus (ideally) no power is dissipated by the switch .Realistically, semiconductor switches such as MOSFETs or BJTs are non ideal switches, but high efficiency controllers can still built[6].

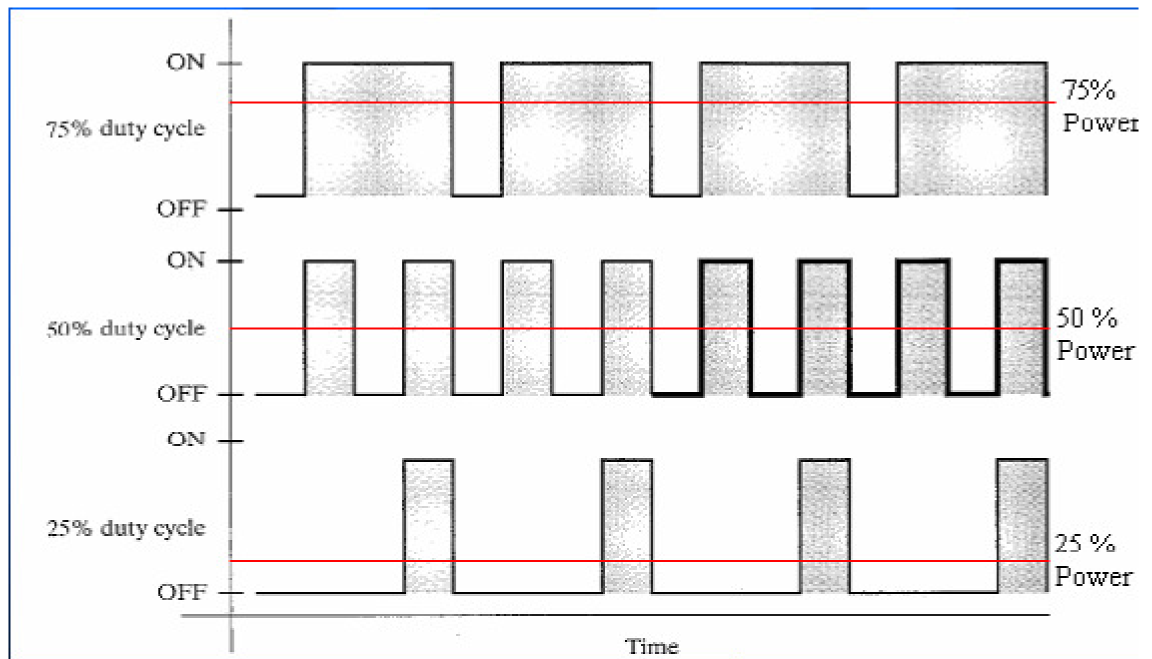


Figure 2.2. PWM Technique

A PWM signal can be used to switch the motor on and off as shown in Figure 2.3.

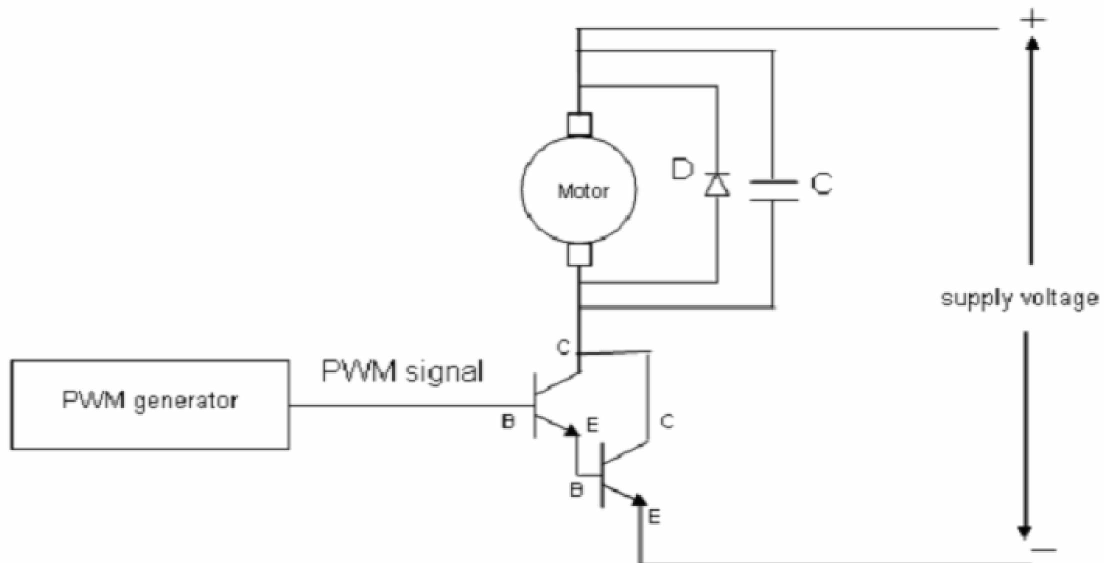


Figure 2.3: Speed control of DC Motor using PWM

The PWM signal is applied to the transistor pair, which acts as a switch. Whenever the PWM signal is high, the switch is closed and the entire supply voltage is applied across the motor terminals. When the PWM signal is low, the switch is open and the supply voltage across the motor is 0 volts. If we apply a PWM signal with a 50 % duty cycle then the average voltage across the motor is 50%. It does not take much work to show that the average voltage across the motor is given by:

Vmotor, average = Vsupply*duty cycle

and, therefore, motor speed = (motor speed when driven by Vsupply) * duty cycle. A PWM signal can be generated in a number of ways. One using software implemented by the microcontroller and the other using hardware implemented in an FPGA. A software implementation may also be lower risk than a hardware implementation because it may be changed at the last minute by changing the program in memory (changing the program, though, also increases the risk of undetected software bugs). However, the capability of a microcontroller to perform these tasks may be limited. The microprocessor may have to perform several tasks in addition to implementing the PWM and thus may not be able to implement the PWM fast enough or with enough accuracy. Implementing the PWM in hardware frees up the microprocessor for these other tasks and ensures that the task is performed quickly and accurately. The best choice - hardware or software - depends significantly on the application [2].

Advantages of PWM

- low power, noise-free ,low cost features

- High efficiency
- Flexibility in control
- Light weight
- Quick Response

# Chapter 3

# Programmable Logic Devices and VHDL

Programmable logic devices were invented in the late seventies and since then have proved to be very popular and are now one of the largest growing sectors in the semiconductor industry[4]. Why are programmable logic devices so widely used? Programmable logic devices provide designers ultimate flexibility, time to market advantage, design integration, are easy to design with and can be reprogrammed time and time again even in the field to upgrade system functionality. The following chapter gives an overview of how and where programmable logic devices are used. It gives a brief history of the programmable logic devices and goes on to describe the different ways of designing with PLDs. And also explore the VHDL.

## 3.1 The History of Programmable Logic

By the late 70's, standard logic devices were the rage and printed circuit boards were loaded with them[10]. Then someone asked the question: "What if we gave the designer the ability to implement different interconnections in a bigger device?" This would allow the designer to integrate many standard logic devices into one part. In order to give the ultimate in design flexibility Ron Cline from Signetics (which was later purchased by Philips and then eventually Xilinx) came up with the idea of two programmable planes[5]. The two programmable planes provided any combination of 'AND' and 'OR' gates and sharing of AND terms across multiple OR's. This architecture was very flexible, but at the time due to wafer geometry's of 10um the input to output delay or propagation delay (Tpd) was high which made the devices relatively slow[5]. MMI (later purchased by AMD) was enlisted as a second source for the PLA array but after fabrication issues was modified to become the Programmable Array Logic (PAL)

architecture by fixing one of the programmable planes. This new architecture differs from that of the PLA by having one of the programmable planes fixed - the OR array. This PAL architecture had the added benefit of faster Tpd and less complex software but without the flexibility of the PLA structure. Other architectures followed, such as the PLD (Programmable Logic Device). This category of devices is often called Simple PLD (SPLD). The architecture has a mesh of horizontal and vertical interconnect tracks. At each junction, there is a fuse. With the aid of software tools, the user can select which junctions will not be connected by "blowing" all unwanted fuses. (This is done by a device programmer or more commonly nowadays using In-System Programming or ISP). Input pins are connected to the vertical interconnect and the horizontal tracks are connected to AND-OR gates, also called "product terms". These in turn connect to dedicated flip-flops whose outputs are connected to output pins. PLDs provided as much as 50 times more gates in a single package than discrete logic devices![4] A huge improvement, not to mention fewer devices needed in inventory and higher reliability over standard logic. Programmable Logic Device (PLD) technology has moved on from the early days with such companies as Xilinx producing ultra low power CMOS devices based on Flash technology. Flash PLDs provide the ability to program the devices time and time again electrically programming and Erasing the device! Gone are the days of erasing taking in excess of twenty minutes under an UV eraser[5].

**Digital logic technologies**

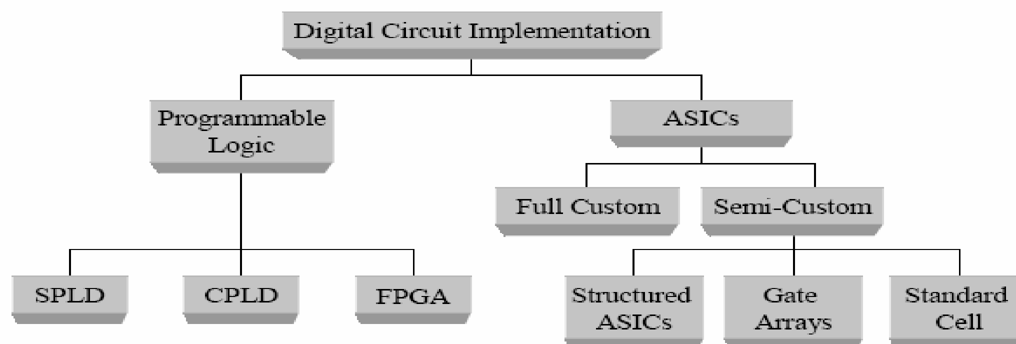



Fig 3.1. Digital logic technology

## 3.2 simple Programmable Logic Devices (SPLDs)

The Simple Programmable Logic Device architectures has the following features

- One programmable plane AND/Fixed OR
- Finite combination of ANDs/Ors
- medium logic density available to user
- programmable logic array

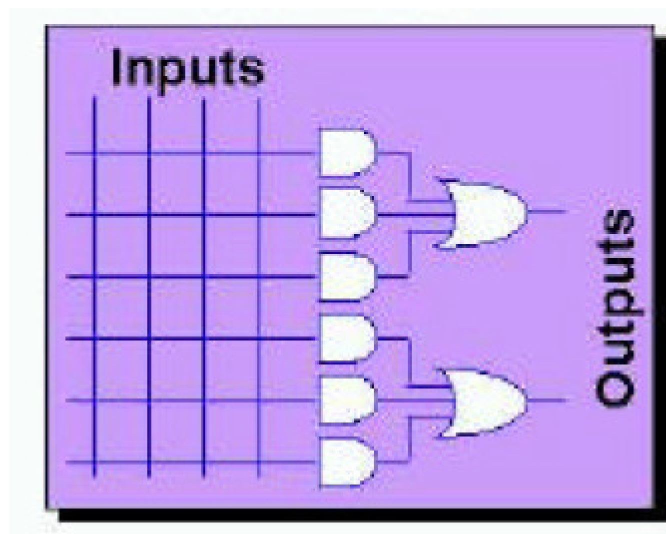



Figure 3.2 SPLD Architectures

## 3.3 Complex Programmable Logic Devices (CPLDs)

Complex Programmable Logic Devices (CPLD) are another way to extend the density of the simple PLDs. The concept is to have a few PLD blocks or macrocells on a single device with general purpose interconnect in between. Simple logic paths can be implemented within a single block. More sophisticated logic will require multiple blocks and use the general purpose interconnect in between to make these connections. CPLDs are great at handling wide and complex gating.

CPLD architectures has the following features

- Central, global interconnect
- Simple, deterministic timing
- Easily routed
- PLD tools add only interconnect
- Wide, fast complex gating

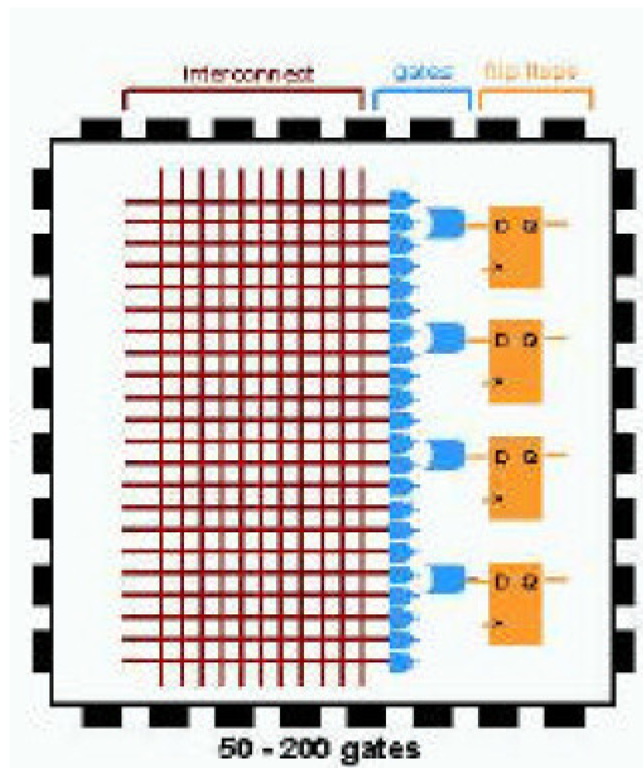



Figure 3.3 CPLD Architecture

## 3.4 Field Programmable Gate Arrays (FPGAs)

A field Programmable Gate array is a digital integrated circuit that can be programmed to do any type of digital function. There are two main advantages of an FPGA over a microprocessor chip for controller:

**1.** An FPGA has the ability to operate faster than a microprocessor chip.

**2.** The new FPGAs that are on the market will support hardware that is upwards of one million gates.

FPGAs are programmed using support software and a download cable connected to a computer. Once they are programmed, they can be disconnected from the computer and will retain their functionality until the power is removed from the chip[10].

The FPGA consists of three major configurable elements:

**1.** Configurable Logic Blocks (CLBs) arranged in an array that provides the functional elements and implements most of the logic in an FPGA.

**2.** Input-output blocks (IOBs) that provide the interface between the package pins and internal signals lines.

**3.** Programmable Interconnect resources that provide routing path to connect inputs and outputs of CLBs and IOBs onto the appropriate network.

Many manufacturers deliver FPGAs such as Quicklogic, Altera, Atmel, xilinx, etc. In this paper the architectural design of xilinx FPGAs is studied. In 1985, a company called Xilinx introduced a completely new idea. The concept was to combine the user control and time to market of PLDs with the densities and cost benefits of gate arrays. A lot of customers liked it - and the FPGA was born. Today Xilinx is still the number one FPGA vendor in the world![10] An FPGA is a regular structure of logic cells or modules and interconnect which is under the designer's complete control. This means the user can design, program and make changes to his circuit whenever he wants. And with FPGAs now exceeding the 10 million gate limit (Xilinx VirtexII is the current record holder), the designer can dream big![5].

### 3.4.1 FPGAs Architectures

Generally the FPGA architecture contains configurable logic block, input output block and Programmable interconnect resources. The Architectures provides the following features.

- Channel Based Routing
- Tools more complex than CPLDs
- Fine Grained
- Fast register pipelining
- Post layout timing

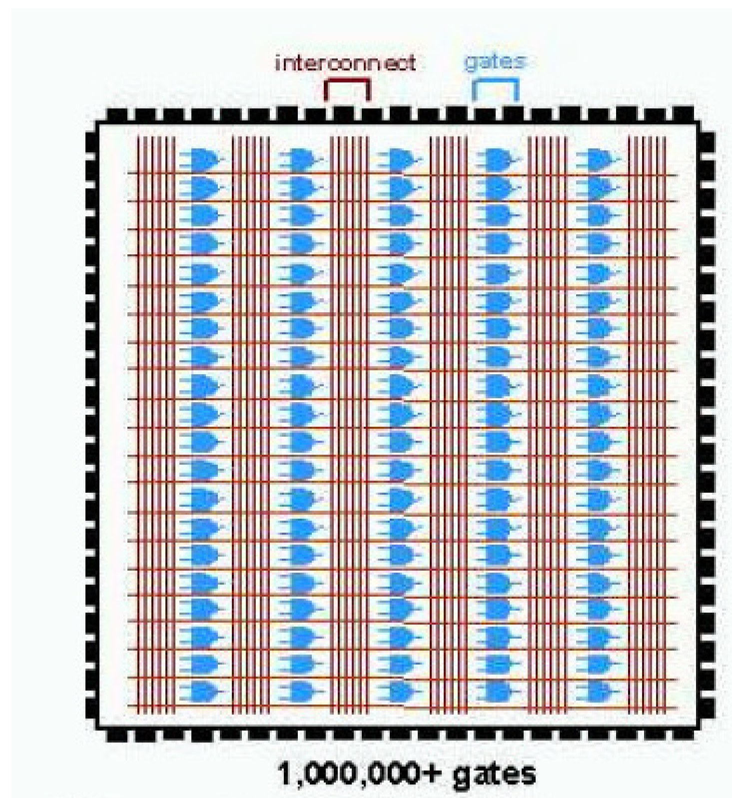



Figure 3.4 FPGA Architecture

With the introduction of the Spartan range of FPGAs we can now compete with Gate Arrays on all aspects - price, gate and I/O count, performance and cost! The new Spartan IIE will provide

up to 300k gates at a price point that enables Application Specific Standard Product (ASSP) replacement [23].

There are 2 basic types of FPGAs:

1. SRAM-based reprogrammable
2. One-time programmable (OTP).

These two types of FPGAs differ in the implementation of the logic cell and the mechanism used to make connections in the device. The dominant type of FPGA is SRAM-based and can be reprogrammed by the user as often as the user chooses. One-time programmable (OTP) FPGAs use anti-fuses (contrary to fuses, connections are made not "blown" during programming) to make permanent connections in the chip.[5]

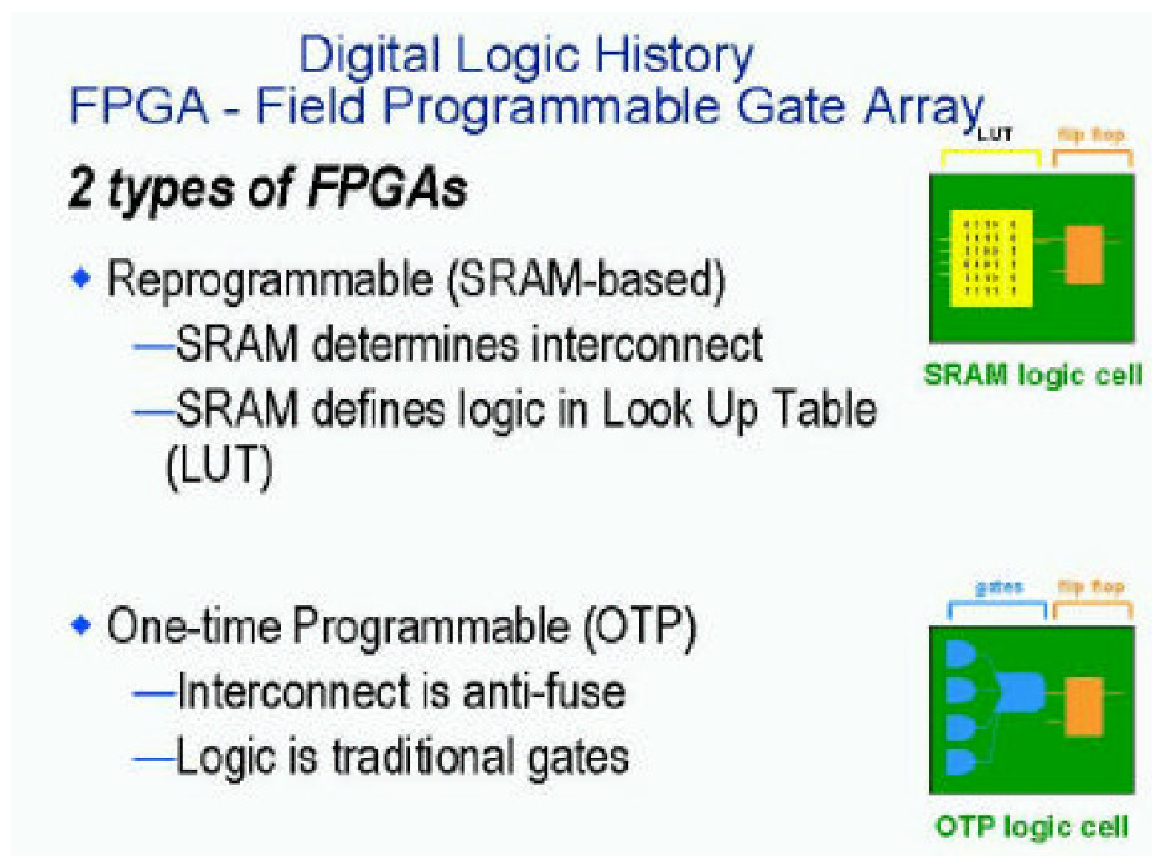



Figure 3.5 Digital Logic History

In the SRAM logic cell, instead of conventional gates there is instead Look Up Table (LUT) which determines the output based on the values of the inputs. (In the “SRAM logic cell” diagram above you can see 6 different combinations of the 4 inputs that will determine the values of the output). SRAM bits are also used to make connections. One-time programmable (OTP) FPGAs use anti-fuses (contrary to fuses, connections are made not “blown” during programming) to make Permanent connections in the chip and so do not require a SPROM or other means to download the program to the FPGA. However, every time you make a design change, you must throw away the chip! The OTP logic cell is very similar to PLDs with dedicated gates and flipflops. [21]

### 3.4.2 XILINX SRAM-BASED FPGAs

The basic structure of Xilinx FPGAs is array_based, meaning that each chip comprises a two dimensional array of logic blocks that can be interconnected via horizontal and vertical routing channels. An illustration of this type of architecture was shown in Figure 3.4. Xilinx introduced the first FPGA family, called the XC2000 series, in about 1985 and now offers three more generations: XC3000, XC4000, and XC5000. Although the XC3000 devices are still widely used, we will focus on the more recent and more popular XC4000 family. We note that XC5000 is similar to XC4000, but has been engineered to offer similar features at a more attractive price. We should also note that Xilinx has recently introduced an FPGA family based on anti-fuses, called the XC8100. The XC8100 has many interesting features, but since it is not yet in widespread use, we will not discuss it here. The Xilinx 4000 family devices range in capacity from about 2000 to more than 15,000 equivalent gates. The XC4000 features a logic block (called a Configurable Logic Block (CLB) by Xilinx) that is based on look-up tables (LUTs). A LUT is a small one bit wide memory array, where the address lines for the memory are inputs of the logic block and the one bit output from the memory is the LUT output. A LUT with K inputs would then correspond to a $2_K$ x 1 bit memory, and can realize any logic function of its K inputs by programming the logic function’s truth table directly into the memory. The XC4000 CLB contains three separate LUTs, in the configuration shown in Figure 3.6. There are two 4-input

LUTS that are fed by CLB inputs, and the third LUT can be used in combination with the other two. This arrangement allows the CLB to implement a wide range of logic functions of up to nine inputs, two separate functions of four inputs or other possibilities. Each CLB also contains two flip-flops.

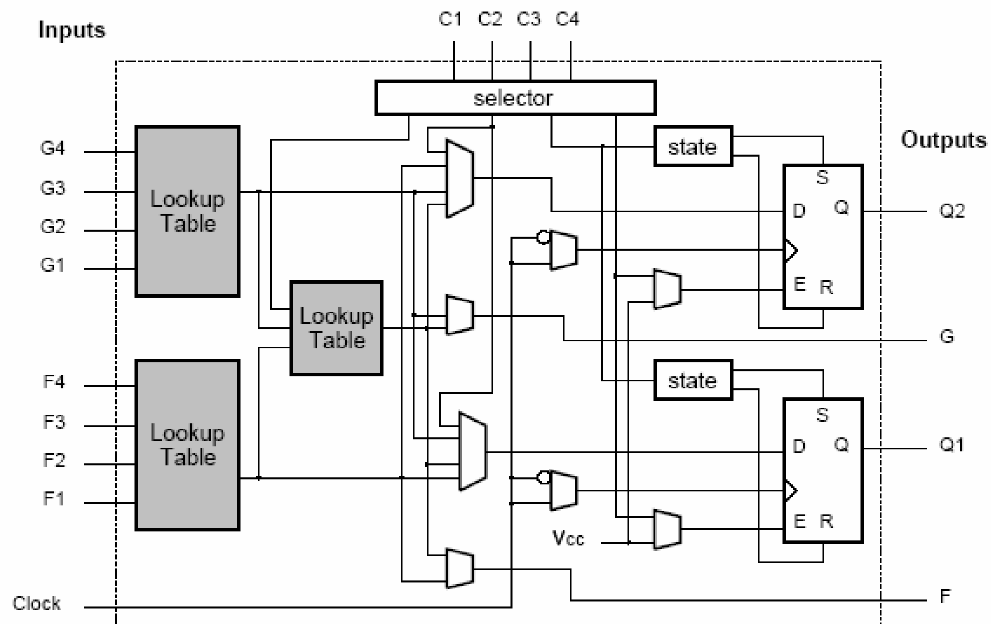



Figure 3.6 Xilinx XC4000 Configurable Logic Block (CLB).

Toward the goal of providing high density devices that support the integration of entire systems, the XC4000 chips have “system oriented” features[10]. For instance, each CLB contains circuitry that allows it to efficiently perform arithmetic (i.e., a circuit that can implement a fast carry operation for adder-like circuits) and also the LUTs in a CLB can be configured as read/write RAM cells. A new version of this family, the 4000E, has the additional feature that the RAM can be configured as a dual port RAM with a single write and two read ports. In the 4000E, RAM blocks can be synchronous RAM. Also, each XC4000 chip includes very wide AND-planes around the periphery of the logic block array to facilitate implementing circuit blocks such as wide decoders. Besides logic, the other key feature that characterizes an FPGA is its interconnect

structure. The XC4000 interconnect is arranged in horizontal and vertical channels. Each channel contains some number of short wire segments that span a single CLB (the number of segments in each channel depends on the specific part number), longer segments that span two CLBs, and very long segments that span the entire length or width of the chip. Programmable switches are available to connect the inputs and outputs of the CLBs to the wire segments, or to connect one wire segment to another. A small section of a routing channel representative of an XC4000 device appears in Figure 3.6. The figure shows only the wire segments in a horizontal channel, and does not show the vertical routing channels, the CLB inputs and outputs, or the routing switches. An important point worth noting about the Xilinx interconnect is that signals must pass through switches to reach one CLB from another, and the total number of switches traversed depends on the particular set of wire segments used. Thus, speed-performance of an implemented circuit depends in part on how the wire segments are allocated to individual signals by CAD tools. segments that span the entire length or width of the chip. Programmable switches are available to connect the inputs and outputs of the CLBs to the wire segments, or to connect one wire segment to another.

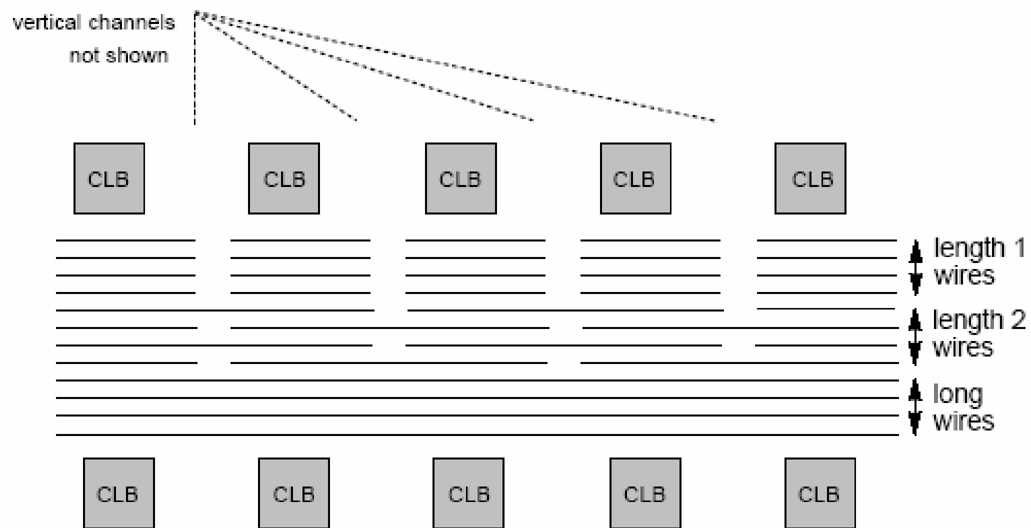



Figure 3.7 Xilinx XC4000 Wire Segments.

### 3.4.3 Xilinx Devices

#### 3.4.3.1 Platform FPGAs

The Virtex-II solution is the first embodiment of the Platform FPGA, once again setting a new benchmark in performance, and offering a feature set that is unparalleled in the industry.

With densities ranging from 40,000 up to 10 million system gates. Virtex-II solutions are empowered by advanced design tools that drive time to market advantages through fast design, powerful synthesis, smart implementation algorithms, and efficient verification capabilities. [10]

#### 3.4.3.2 Virtex FPGAs

The Xilinx Virtex™ series was the first line of FPGAs to offer one million system gates. Introduced in 1998, the Virtex product line fundamentally redefined programmable logic by expanding the traditional capabilities of field programmable gate arrays (FPGAs) to include a powerful set of features that address board level problems for high performance system designs. The latest devices in the Virtex-E series, unveiled in 1999, offer more than three million system gates. The Virtex-EM devices, introduced in 2000 and the first FPGAs to be manufactured using an advanced copper process, offer additional on chip memory for network switch applications.[10]

#### 3.4.3.3 Spartan FPGAs

Xilinx Spartan™ FPGAs are ideal for low-cost, high volume applications and are targeted as replacements for fixed-logic gate arrays and for application specific standard products (ASSP) products such as bus interface chip sets. The are four members of the family Spartan IIE (1.8V), Spartan II (2.5V), Spartan XL (3.3V) and Spartan (5V) devices. The Spartan-IIE (1.8V core) family offers some of the most advanced FPGA technologies available today, including programmable support for multiple I/O standards, on-chip block RAM.[10]

All xilinx FPGA contain the same basic resources

1. Slices (grouped into CLBs)
   - .contain combinational logic and register resources

- IOBs
  - .Interface between the FPGA and the outside world

2. Programmable interconnect
3. Other resources
   - Memory
   - Multipliers

### 3.4.4 Slices and CLBs

Each Virtex-II CLB contains four slices. Local routing provides feedback between slices in the same CLB, and it provides routing to neighboring CLBs. A switch matrix provides access to general routing resources.

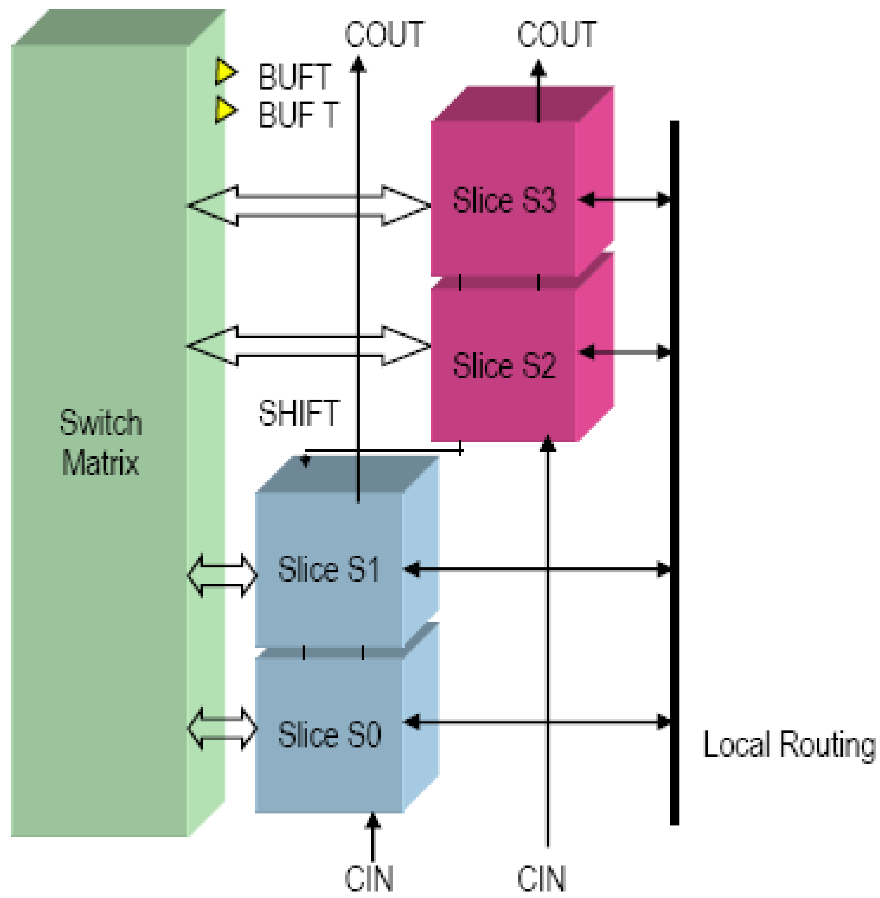



Figure 3.8. slices

Each slice has four outputs. Two registered outputs and two non registered outputs. Two BUFTs associated with each CLB, accessible by all 16 CLB outputs.

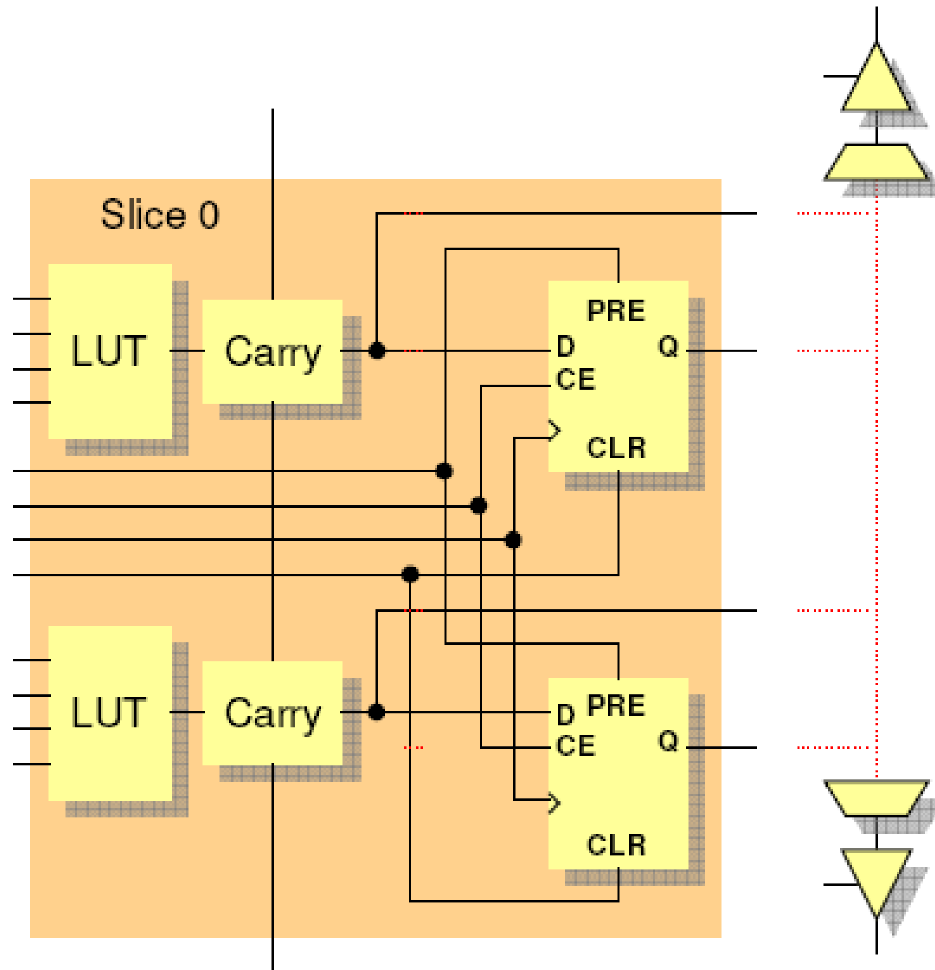



Figure 3.9. Simplified Slice structure

**Look-Up Tables**

Combinational logic is stored in Look-Up Tables (LUTs)

- Also called Function Generator
- capacity is limited by the number of inputs, not by complexity
- delay through Look-Up table is constant

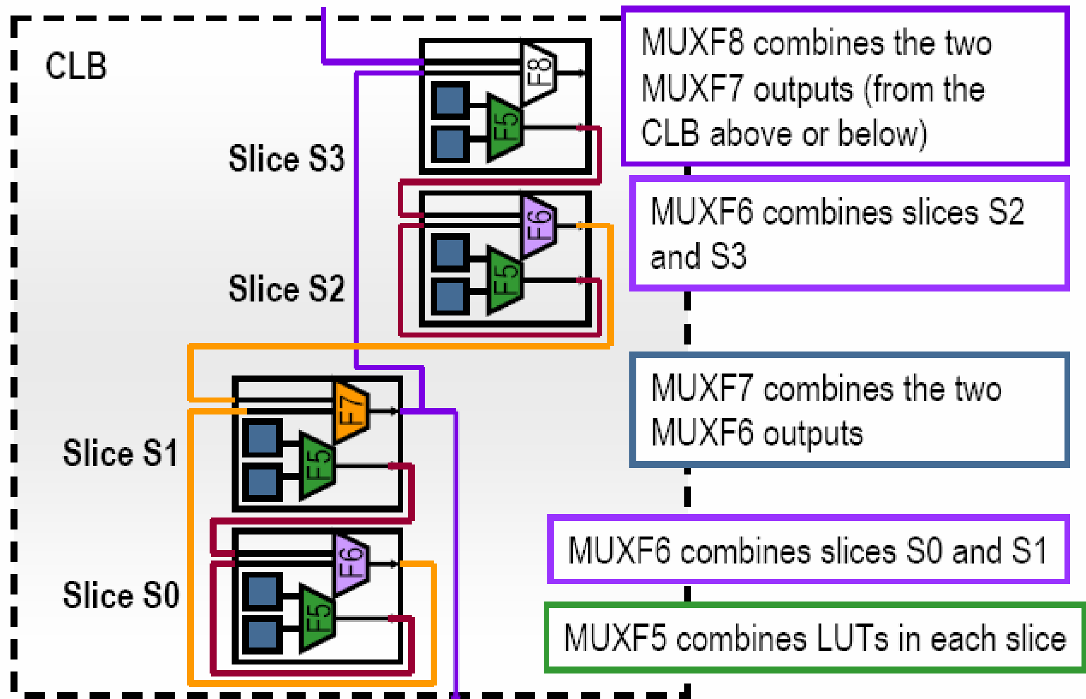



Figure 3.10. Connecting Look-Up Tables

### 3.4.5 Advantage of FPGA

The advantages of using these techniques over a classical TTL-based system are speed and adaptability [21]. When an error is made or a system upgrade is ready, reprogramming the device is sufficient to alter the system. Thus, cutting down design time. The automatic syntheses, easy testability and in-system logic analyzers shorten this even more. On the other hand, the same device can perform different tasks according to the configuration downloaded into it (this can even be done on board without removing the component). Due to the increased complexity of FPGA's, it is possible to implement more complex and demanding structures and even integrate a complete design on one chip. The difference with e.g. a microprocessor is that the logic can execute in parallel. In a microprocessor all instructions are performed one after the other. The software gives the sequence of instructions that will execute. When comparing to a normal processor, FPGA's can perform better. The parallelism allows for more efficient computation. If

the latter can carry out 100 instructions at the same time, the clock can be 100 times slower without losing any execution speed. An FPGA running at 50MHz can therefore be faster than a processor running at 500MHz, depending on the application [10]. All this is possible by the large amount of transistors available in FPGA's. Programming these devices isn't easy. Each Look-Up Table (LUT), I/O blocks and interconnection needs to be programmed. This is the price you pay for flexibility. Luckily a programming language and tools were developed to make this a less complex task. The hardware isn't programmed bit by bit, but described in a high-level description language. By describing the behaviour, design can be completed in a shorter period and the code will be more readable and portable than a schematic entry. There fore FPGA based controller is preferable because of the above and the following advantages.

- ✓ FPGA can process information faster
- ✓ controller architecture can be optimized for space or speed
- ✓ available in radiation tolerant package
- ✓ implementation in VHDL allows the targeting of a variety of commercially available device
- ✓ FPGA allows for implementation of parallel processing

## 3.5 Hardware Description Languages (HDL)

HDL describes how hardware behaves. There are two main differences between traditional programming languages and HDL:

1) Traditional languages are a sequential process whereas HDL is a parallel process.

2) HDL runs forever whereas traditional programming languages will only run if directed.

A digital design can be created using schematic digital design editor that uses graphic symbols of the circuit or by using hardware description languages such as Very High Speed Integrated Circuit hardware description language (VHDL). One of the key features of using VHDL is that it can be used to achieve all the goals for documentation, simulation, verification and synthesis of digital designs, thus saving a lot of effort and time. VHDL can be used to model a digital system at many levels of abstraction, ranging from the algorithmic level to the gate level. The VHDL language can be regarded

as an integrated amalgamation of the following factors, such as, Sequential language + Concurrent language + Net-list language + timing-specifications + waveform generation language => VHDL. Therefore, the language has constructs that enable one to express the concurrent or sequential behaviour of a digital system with or without timing. In this paper, the implementation of the controller on a FPGA using VHDL is presented.

## 3.6 VHDL

VHDL is a language for describing digital electronic systems. It arose out of the United States Government's Very High Speed Integrated Circuits (VHSIC) program, initiated in 1980[3]. 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, and subsequently adopted as a standard by the Institute of Electrical and Electronic Engineers (IEEE) in the US. VHDL is designed to fill a number of needs in the design process. Firstly, it allows description of the structure of a design that is how it is decomposed into sub-designs, and how those sub-designs are interconnected. Secondly, it allows the specification of the function of designs using familiar programming language forms. Thirdly, as a result, it allows a design to be simulated before being manufactured, so that designers can quickly compare alternatives and test for correctness without the delay and expense of hardware prototyping.

### 3.6.1 The History of VHDL

**IEEE Standard 1076**

In the early 1980s, a team of engineers from three companies -- IBM, Texas Instruments and Intermetrics -- were contracted by the Department of Defense to complete the specification and implementation of a new, language-based design description method. The first publicly available version of VHDL, version 7.2, was released in 1985. In 1986, the Institute of Electrical and Electronics Engineers, Inc. (IEEE) was presented with a proposal to standardize the language,

which it did in 1987 after substantial enhancements and modifications were made by a team of commercial, government and academic representatives. The resulting standard, IEEE 1076-1987, is the basis for virtually every VHDL simulation and synthesis product sold today. An enhanced and updated version of the language, IEEE 1076-1993, was released in 1994, and VHDL tool vendors have been responding by adding these new language features to their products.[11]

**IEEE Standard 1164**

Although IEEE Standard 1076 defines the complete VHDL language, there are aspects of the language that make it difficult to write completely portable design descriptions (descriptions that can be simulated identically using different vendors' tools). The problem stems from the fact that VHDL supports many abstract data types, but it does not address the simple problem of characterizing different signal strengths or commonly used simulation conditions such as unknowns and high-impedance Soon after IEEE 1076-1987 was adopted, simulator companies began enhancing VHDL with new signal types (typically through the use of syntactically legal, but nonstandard, enumerated types) to allow their customers to accurately simulate complex electronic circuits. This caused problems because design descriptions entered using one simulator were often incompatible with other simulation environments. VHDL was quickly becoming nonstandard. To get around the problem of nonstandard data types, another standard, numbered 1164, was created by an IEEE committee. It defines a standard package (a VHDL feature that allows commonly used declarations to be collected into an external library) containing definitions for a standard nine-valued data type. This standard data type is called std_logic, and the IEEE 1164 package is often referred to as the standard logic package or MVL9 (for multi-valued logic, nine values). The IEEE 1076-1987 and IEEE 1164 standards together form the VHDL standard in widest use today.

Building blocks are the basic structures of the VHDL language. The top-level block is composed of different building blocks. Each sub-block can also be composed of other blocks. (See Figure 3.11)

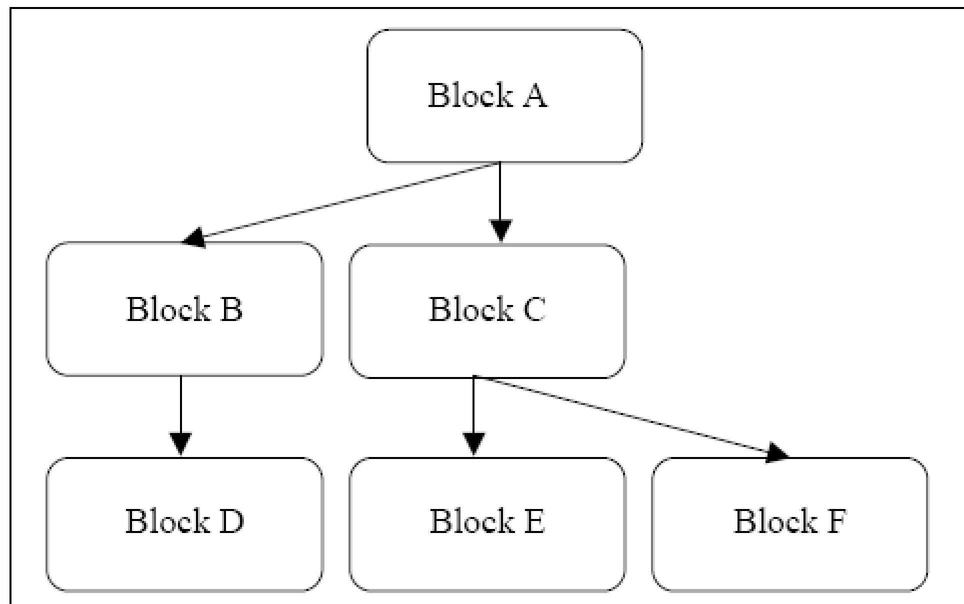



Figure 3.11. VHDL block structure

Each block consists of two parts: an entity and an architecture element (see Figure 3.12). The entity describes the interface to that block and a separate part, the architecture, associated with the entity describes how that block operates. The entity is like a pin description in a data book, specifying the inputs and outputs to the block. The architecture can be compared to a schematic for the block, describing in words (not symbols) what the block does.

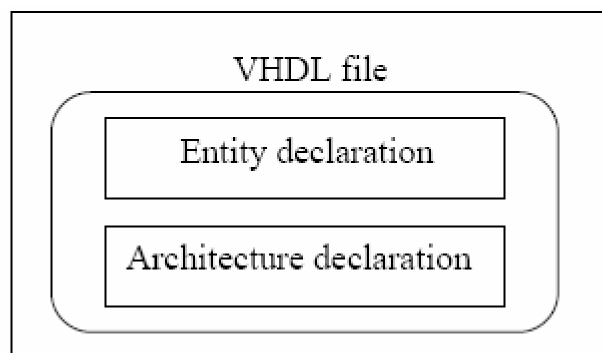



Figure 3.12 .VHDL-file structures

When analyzing the code, a syntax for writing VHDL code can be derived.

| Entity | Architecture |
|---|---|
| ***entity*** *entity-name* ***is***<br>  ***port (***<br>        *signal-names* ***:*** *mode signal-type****;***<br>        *signal-names* ***:*** *mode signal-type****;***<br>        ...<br>        *signal-names* ***:*** *mode signal-type****);***<br>***end*** *entity-name****;***<br><br>(a) | ***architecture*** *architecture-name* ***of*** *entity-name* ***is***<br>  *variable declarations*<br>  *signal declarations*<br>  *constant declarations*<br>  *function definitions*<br>  *component declarations*<br>***begin***<br>  *concurrent-statement****;***<br>  ...<br>  *concurrent-statement****;***<br>***end*** *architecture-name****;***<br>(b) |

Figure 3.13 (a) Overview of entity

(b) Architecture syntax

| | |
|---|---|
| **Signal declarations** | Signals are the 'wires' of VHDL. They are used to connect different blocks.<br>Syntax: **signal** *signal-names* **:** *signal-type* |
| **Variable declarations** | Same as signals, but no physical significance.<br>Syntax: **variable** *variable-names* **:** *variable-type* |
| **Constant declarations** | Fixed numbers can de declared constant to improve readability<br>Syntax: **constant** *constant-names* **:** *constant-type* **:=** *constant-value* |
| **Function definitions** | Functions have the same meaning in VHDL as in most programming languages. They can be used to define subroutines. |
| **Component declarations** | Definitions of the components (blocks) you want to use in your design. |

Figure3.14 Overview of architecture declarations

**Every VHDL design description consists of the following:**

1. At least one entity/architecture pair, which in VHDL jargon is sometimes referred to as a "design entity". In a large design, you will typically write many entity/architecture pairs and connect them together to form a complete circuit.

An entity declaration describes the circuit as it appears from the "outside", that is, from the perspective of its input and output interfaces. If you are familiar with schematics, you might think of the entity declaration as being analogous to a block symbol on a schematic.

2. The architecture declaration, which refers to the fact that every entity in a VHDL design description must be bound with a corresponding architecture. The architecture describes the actual function -- or contents -- of the entity to which it is bound.

An entity declaration provides the complete interface for a circuit. Using the information provided in an entity declaration (the names, data types and direction of each port), you have all the information you need to connect that portion of a circuit into other, higher-level circuits, or to develop input stimulus (in the form of a test bench) for testing purposes.

**Entity Declarations**

The entity declaration includes a name, compare, and port declaration statement defining all the inputs and outputs of the entity. Each of these three ports is given a direction (in, out or inout), and a type (in this case, either bit vector(0 to 7), which specifies an 8-bit array, or bit, which represents a single-bit value). There are many different data types available in VHDL. The simplest data types in VHDL, bit and bit vector, will be used.

**Architecture Declarations**

Every entity declaration you write must be accompanied by at least one corresponding architecture.

## 3.6.2 Design Units

One concept unique to VHDL (when compared to software programming languages and to Verilog HDL) is the concept of a "design unit". Design units (which may also be referred to as "library units") are segments of VHDL code that can be compiled separately and stored in a library. You have been introduced to two design units already: the entity and the architecture. There are actually five types of design units in VHDL: entities, architectures, packages, package bodies, and configurations. [11]

**Entities**

A VHDL entity is a statement (identified by the entity keyword) that defines the external specification of a circuit or sub-circuit. The minimum VHDL design description must include at least one entity and one corresponding architecture. When you write an entity declaration, you must provide a unique name for that entity and a port list defining the input and output ports of the circuit. Each port in the port list must be given a name, direction (or "mode", in VHDL jargon) and a type. Optionally, you may also include a special type of parameter list (called a generic list) that allows you to pass additional information into an entity.

**Architectures**

A VHDL architecture declaration is a statement (beginning with the architecture keyword) that describes the underlying function and/or structure of a circuit. Each architecture in your design must be associated (or bound) by name with one entity in the design. VHDL allows you to create more than one alternate architecture for each entity. This feature is particularly useful for simulation and for project team environments in which the design of the system interfaces (expressed as entities) is done by a different engineer than the lower-level architectural description of each component circuit. An architecture declaration consists of zero or more declarations (of items such as intermediate signals, components that will be referenced in the architecture, local functions and procedures, and constants) followed by a begin statement, a series of concurrent statements, and an end statement.

**Packages and Package Bodies**

A VHDL package declaration is identified by the package keyword, and is used to collect commonly-used declarations for use globally among different design units. You can think of a package as a common storage area, one used to store such things as type declarations, constants, and global subprograms. Items defined within a package can be made visible to any other design unit in the complete VHDL design, and they can be compiled into libraries for later re-use. A

package can consist of two basic parts: a package declaration and an optional package body. Package declarations can contain the following types of statements:

- type and subtype declarations
- constant declarations
- global signal declarations
- function and procedure declarations
- attribute specifications
- file declarations
- component declarations
- alias declarations
- disconnect specifications
- use clauses.

Items appearing within a package declaration can be made visible to other design units through the use of a use statement. If the package contains declarations of subprograms (functions or procedures) or defines one or more deferred constants (constants whose value is not immediately given), then a package body is required in addition to the package declaration. A package body (which is specified using the package body keyword combination) must have the same name as its corresponding package declaration, but it can be located anywhere in the design (it does not have to be located immediately after the package declaration). The relationship between a package and package body is somewhat akin to the relationship between an entity and its corresponding architecture. (There may be only one package body written for each package declaration, however.) While the package declaration provides the information needed to use the items defined within it (the parameter list for a global procedure, or the name of a defined type or subtype), the actual behavior of such elements as procedures and functions must be specified within package bodies.

**Package and Package Body Declarations**

A package is a collection of types, constants, subprograms and possibly other things, usually intended to implement some particular service or to isolate a group of related items. In particular,

the details of constant values and subprogram bodies can be hidden from users of a package, with only their interfaces made visible. A package may be split into two parts: a package declaration, which defines its interface, and a package body, which defines the deferred details. The body part may be omitted if there are no deferred details. The syntax of a package declaration is:
package_declaration ::=
**package** identifier **is**
package_declarative_part
**end** [ *package*_simple_name ] ;
package_declarative_part ::= { package_declarative_item }
package_declarative_item ::=
subprogram_declaration
type_declaration
subtype_declaration
constant_declaration
alias_declaration
use_clause
The syntax for a package body is:
package_body ::=
**package body** *package*_simple_name **is**
package_body_declarative_part
**end** [ *package*_simple_name ] ;
package_body_declarative_part ::= { package_body_declarative_item }

**Configurations**

The final type of design unit available in VHDL is called a configuration declaration. A configuration declaration (identified with the configuration keyword) specifies which architectures are to be bound to which entities, and allows you to change how components are connected in your design description at the time of simulation or synthesis.
Configuration declarations are always optional, no matter how complex a design description you create. In the absence of a configuration declaration, the VHDL standard specifies a set of rules that provide you with a default configuration. For example, in the case where you have provided more than one architecture for an entity, the last architecture compiled will take precedence and will be bound to the entity.

### 3.6.3 Levels of Abstraction

VHDL supports many possible styles of design description. These styles differ primarily in how closely they relate to the underlying hardware. The different styles of VHDL refer to the differing levels of abstraction possible using the language -- behavior, dataflow, and structure.

**The three levels of abstraction are as follows:**

1. **Behavior**

The highest level of abstraction supported in VHDL is called the behavior level of abstraction. When creating a behavioral description of a circuit, you will describe your circuit in terms of its operation over time. The concept of time is the critical distinction between behavioral descriptions of circuits and lower-level descriptions (specifically descriptions created at the dataflow level of abstraction).

In a behavioral description, the concept of time may be expressed precisely, with actual delays between related events (such as the propagation delays within gates and on wires), or it may simply be an ordering of operations that are expressed sequentially (such as in a functional description of a flip-flop). When you are writing VHDL for input to synthesis tools, you may use behavioral statements to imply that there are registers in your circuit. It is unlikely, however, that your synthesis tool will be capable of creating precisely the same behavior in actual circuitry as you have defined in the language.

2. **Dataflow**

In the dataflow level of abstraction, you describe your circuit in terms of how data moves through the system. At the heart of most digital systems today are registers, so in the dataflow level of abstraction you describe how information is passed between registers in the circuit. You will probably describe the combinational logic portion of your circuit at a relatively high level (and let

a synthesis tool figure out the detailed implementation in logic gates), but you will likely be quite specific about the placement and operation of registers in the complete circuit.

3. **Structure**

The third level of abstraction, structure, is used to describe a circuit in terms of its components. Structure can be used to create a very low-level description of a circuit (such as a transistor-level description) or a very high-level description (such as a block diagram).

In a gate-level description of a circuit, for example, components such as basic logic gates and flip-flops might be connected in some logical structure to create the circuit. This is what is often called a netlist. For a higher-level circuit (one in which the components being connected are larger functional blocks), structure might simply be used to segment the design description into manageable parts. Structure-level VHDL features such as components and configurations are very useful for managing complexity. The use of components can dramatically improve your ability to reuse elements of your designs, and they can make it possible to work using a top-down design approach.

## 3.6.4 Signals and Variables

There are two fundamental types of objects used to carry data from place to place in a VHDL design description: signals and variables. In virtually all cases, you will want to use variables to carry data between sequential operations (within processes, procedures and functions) and use signals to carry information between concurrent elements of your design (such as between two independent processes).

**Signal Declarations**

Signals are used to connect submodules in a design. They are declared using the syntax:
signal_declaration ::=

**signal** identifier_list : subtype_indication [ signal_kind ] [ := expression ] ;

signal_kind ::= **register | bus**

**Architecture Blocks**

The submodules in an architecture body can be described as blocks. A block is a unit of module structure, with its own interface, connected to other blocks or ports by signals. A block is specified using the syntax:

block_statement ::=

*block*_label :

**block** [ ( *guard*_expression ) ]

block_header

block_declarative_part

**begin**

block_statement_part

**end block** [ *block*_label ] ;

### 3.6.5 Test Benches

At this point, the sample circuit is complete and ready to be processed by synthesis tools. Before processing the design, however, you should take the time to verify that it actually does what it is intended to do. You should run a simulation. Simulating a circuit such as this one requires that you provide more than just the design description itself. To verify the proper operation of the circuit over time in response to input stimulus, you will need to write a test bench. The easiest way to understand the concept of a test bench is to think of it as a virtual tester circuit. This tester circuit, which you will describe in VHDL, applies stimulus to your design description and (optionally) verifies that the simulated circuit does what it is intended to do. To apply stimulus to your design, your test bench will probably be written using one or more sequential processes, and it will use a series of signal assignments and wait statements to describe the actual stimulus. You will probably use VHDL's looping features to simplify the description of repetitive stimulus

(such as the system clock), and you may also use VHDL's file and record features to apply stimulus in the form of test vectors.

In addition VHDL has got the following properties

- VHDL is not case sensitive.
- The semicolon is used to indicate termination of a statement.
- Two dashes ('—') are used to indicate the start of a comment.
- Identifiers must begin with a letter, subsequent characters must be alphanumeric or '_' (underscore).

## 3.6.6 Advantages of VHDL

Because of the need of digital design flexibility, easy to debug, easy to maintain, standard interface , digital designers (specially engineers, work of large companies) use VHDL now as a Design Tool.

Some of the advantages of VHDL is given below

- Standard format for design exchange
- Technology independent
- Multiple vendor support
- Support for large as well as small designs
- Support for wide range of abstraction in modeling
- Simulation oriented (including for writing testbench)
- User defined value abstractions
- Timing constructs

# Chapter 4

# Design of PWM in XILINX FPGA

To design the PWM in Field programmable gate array first the functional description of the design modeled in very high speed integrated circuit HDL using the behavioral abstraction level and this VHDL code is synthesized and simulated using xilinx Synthesis and simulation tool. After successfully synthesized and simulated the design it can be downloaded to the targeting device (FPGA). These chapter discuses about how the functional description generates and this functional description modeled in VHDL to generate the pulse width modulation. In the process Xilinx ISE and FPGA design flow is used. This chapter discuses about the design of pulse width modulation in xilinx FPGA .as can be discussed in chapter 3, the hardware description language (VHDL) is used to design the PWM in the Field Programmable Gate Array (FPGA).

## 4.1 Xilinx ISE Overview

The Integrated Software Environment (ISE™) is the Xilinx® design software suite that allows taking the design from design entry through Xilinx device programming. The ISE Project Navigator manages and processes the design through the FPGA design flow. The ISE™ design flow comprises the following steps: design entry, design synthesis, design implementation, and Xilinx® device programming. Design verification, which includes both functional verification and timing verification, takes places at different points during the design flow. This section describes what to do during each step.

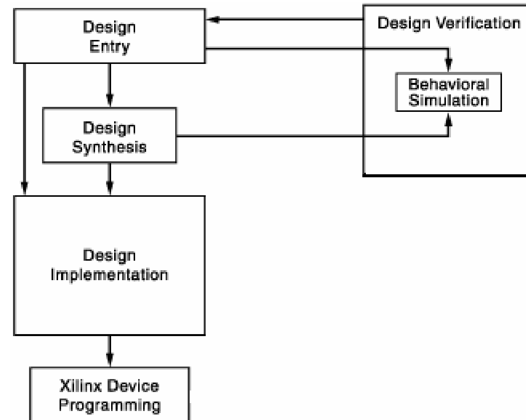



Figure4.1. FPGA design flow

**Design Entry**

Design entry is the first step in the ISE design flow. During design entry, source files are created based on design objectives. We can create the top-level design file using a Hardware Description Language (HDL), such as VHDL. We can use multiple formats for the lower-level source files in the design. After design entry and optional simulation, we run synthesis. During this step, VHDL, designs become netlist files that are accepted as input to the implementation step.

**Implementation**

After synthesis, we run design implementation, which converts the logical design into a physical file format that can be downloaded to the selected target device. From Project Navigator, we can run the implementation process in one step, or we can run each of the implementation processes separately. Implementation processes vary depending on targeting a Field Programmable Gate Array (FPGA).

**Verification**

We can verify the functionality of the design at several points in the design flow. we can use simulator software to verify the functionality and timing of the design or a portion of the design. The simulator interprets VHDL code into circuit functionality and displays logical results of the described HDL to determine correct circuit operation. Simulation allows creating and verifying complex functions in a relatively small amount of time. We can also run in-circuit verification after programming the device.

**Device Configuration**

After generating a programming file, we can configure the device. During configuration, we generate configuration files and download the programming files from a host computer to a Xilinx device.

HDL Overview

A hardware description language (HDL) is used in the design, such as VHDL, for the top-level or lower-level design files. HDL files describe the behavior and structure of system and circuit designs. The design functionality can be verified early in the flow by simulating the HDL description. Testing the design decisions at the register transfer level (RTL) or gate level before the design is implemented allows us to make changes early in the design process. a synthesis engine is used to translate the design into gates.
Synthesis decreases design time by eliminating the need to define every gate. Synthesis to gates reduces the number of errors that may occur during a manual translation of the hardware description to a schematic design.

## 4.2 FPGA Design Flow

With designs of low to moderate complexity, we can process the design using the ISE™ Basic Flow as follows:

1. Write the functional description of the design PWM.

2. Write the VHDL modeling of the design PWM.

3. Synthesize the design using xilinx XST.

4. Run behavioral simulation (also known as RTL simulation).

5. Implement the design: Run the Implement Design process on the top module design, which automatically runs the following processes:

-Translate

- Map

- Place and Route

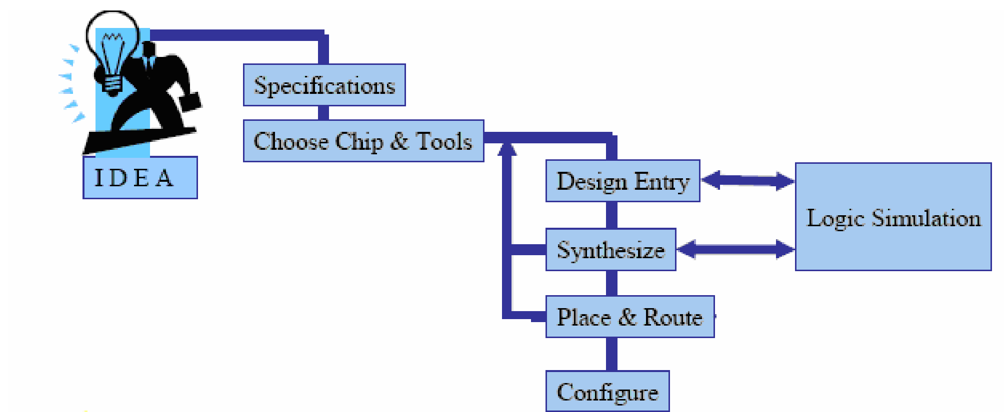



Fig 4.2 FPGA Design methodology

## 4.3 FUNCTIONAL DESCRIPTION OF THE PWM DESIGN

A PWM circuit works by making a pulsating DC square wave with a variable on-to-off ratio. The average on time may be varied from 0 to 100 percent. The widths of the pulses are proportional to the input signal. When the signal is small, a series of narrow pulses is generated. When the signal is large, a series of wide pulses is generated. In many of the applications, the single bit digital output is subject to a low-pass filter that results in an analog output level. The output level is the analog equivalent of the digital PWM's duty-cycle.

**Functional Block Diagram of PWM**

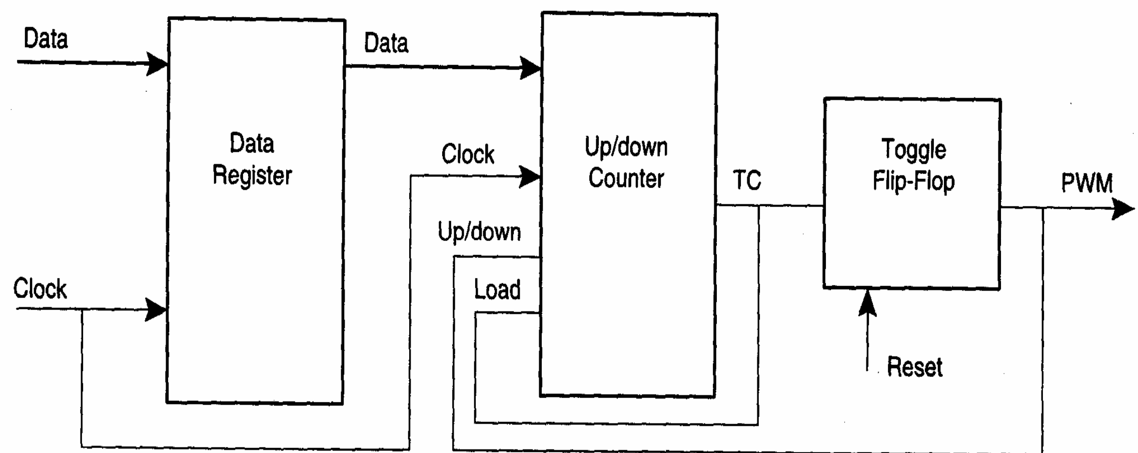
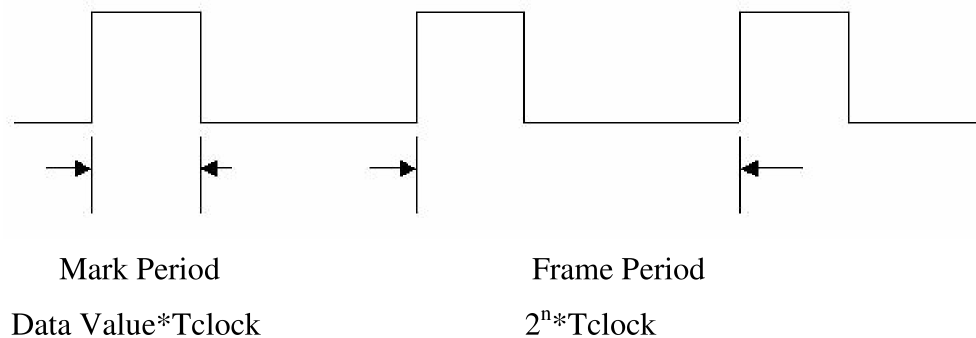



Figure 4.3 functional block diagram of PWM

Basic principle: a register to store the value which is loaded on to the Up/Down Counter whenever the counter reaches its terminal count. The terminal counter is used to generate the pulse width modulation. A data register is used to store the value for the counter. Value determines the pulse width. The Up/Down Counter loaded with a new value from the data

register when the counter reaches its terminal count. Toggle Flip-flop generates the PWM output when data value is first loaded, counter counts down from data value to 0. Terminal count and PWM signals are Low. When counter goes through 0 transitions, terminal count is generated. Triggers Toggle Flip-flop to drive PWM signal High. Data value is re-loaded and counting proceeds up to maximum value. Terminal count generated again when counter reaches its maximum value Drives PWM signal to toggle from High to Low. Data value is re-loaded and cycle repeats. Direction of counter controlled by PWM signal and counter is set to count down when PWM is Low, and count up when PWM is high. Terminal count controls data value loaded to counter from data register. Data is loaded when terminal count is high. Duty cycle of the PWM signal is controlled by data value loaded to the up/down counter. Higher the data value, higher the duty cycle

Sample PWM Output Waveform



Mark Period — Data Value*Tclock

Frame Period — $2^n$*Tclock

Duty Cycle = Mark Period/Frame Period

= Data Value/$2^n$

Frame Period =Tclock*$2^n$

= tperiod

n = log2 (tperiod/Tclock)

For an 8-bit pulse width modulation resolution there is 256 states and the design uses an 8-bit Pulse width modulation, n=8 resulting $2^8$ different duty states and an on-board 4 MHz clock gives the value of the Tclock=250nsec.

And

Mark Period = (Data Value *250nsec)

Frame Period = 250nsec*$2^8$

Duty cycle = (Data value/$2^8$ )

The following shows the count value to pulse output. The counter-comparator circuit functions in the following manner:

**1.** All the registers are reset to 0, including the pulse output.

**2.** At the next active clock edge, the pulse width modulations n-bit value is registered to be compared with the value from the counter.

**3.** The n-bit counter starts counting from [11….111].

**4.** Since the starting value of the counter is the terminal count, the S-R flip-flop is set to 1 and pul**se** is 1 on the next clock edge when reset is 0.

**5.** When the count value equals pulse width modulation _data**,** the S-R flip-flop is reset to 0 and pulse is 0 on the next clock edge.

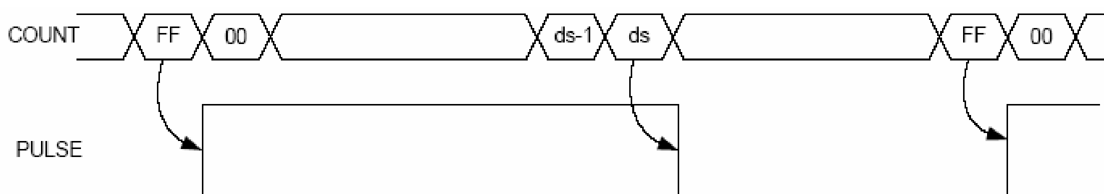



Figure 4.4 Pulse as a Function of the Count Value (ds = pulse width modulation-data)

### 4.4 VHDL MODELING OF THE FUNCTIONAL DESCRIPTION OF PWM DESIGN

As can be discussed in chapter 3 VHDL has got different levels of abstraction such as behavioral, structural and data flow. The highest level of abstraction supported in VHDL is called the behavior level of abstraction. When creating a behavioral description of a circuit, we can describe our circuit in terms of its operation over time.

A digital electronic system can be described as a module with inputs and/or outputs. The electrical values on the outputs are some function of the values on the inputs. In many cases, it is not appropriate to describe a module structurally. One such case is a module which is at the bottom of the hierarchy of some other structural description. Once the structure and behaviour of a module have been specified, it is possible to simulate the module by executing its bevioural description. This is done by simulating the passage of time in discrete steps. At some simulation time, a module input may be stimulated by changing the value on an input port. The module reacts by running the code of its behavioral description and scheduling new values to be placed on the signals connected to its output ports at some later simulated time.

In this paper an 8 bit PWM resolution of a VHDL description is used. We start the description of an entity by specifying its external interface, which includes a description of its ports. So the PWM defined as:

**Entity entity_name is**

**[generic_declaration]**
**[port_clause]**
**[entity_declarative]**

**End [entity name];[24]**

Ports: Provide communication with other components, must have signal name, type and mode.

- Port Modes :

- in (data goes into entity only)
- . out (data goes out of entity only and not used internally)
- . inout (data is bi-directional)
- . buffer (data goes out of entity and used internally)

Packages and Libraries: Libraries contain packages and Packages contain commonly used types, operators, constants, functions, etc. Both must be "opened" before their contents can be used in an entity or architecture.

**library ieee;**

**use ieee.std_logic_1164.all;**

The entity of the design of PWM syntax is given below, which includes the input and the output interface signal.

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

Entity pulse width modulation is

   Port (
          clock: in STD_LOGIC;
          Reset: in STD_LOGIC;
          Data_value: in STD_LOGIC_VECTOR (7 downto 0);
          Pulse width modulation: out STD_LOGIC);

End pulse width modulation;
```

The internal aspect of the design unit can be behavioral (RTL), and always associated with single entity. Single entity can have multiple architectures**.** An implementation of the entity is described

in an architecture body. There may be more than one architecture body corresponding to a single entity specification, each of which describes a different view of the entity. A behavioral description of the pulse width modulation could be written as:

**Architecture architecture_name of entity_name is**

**{ architecture_declarative_part }**

**Begin**

**{ architecture_declarative_part }**

**End { architecture_name } ;[24]**

Architecture **Behavioral** of pulse width modulation is

```
SIGNAL register_out: std_logic_vector (7 downto 0);
SIGNAL counter_out_intger: std_logic_vector (7 downto 0);
SIGNAL pulse width modulation_integer, rco_intger: std_logic;

Begin

Process (Clock, register_out, Reset)
BEGIN
IF (reset ='1') THEN
register_out <="00000000";
ELSIF (rising_edge (clock)) THEN
register_out <= data_value;
END IF;

END PROCESS;
```

An 8 bit data register is used to store the data value. The data values that stores in the register will determine the duty cycle of the pulse width modulation output. An 8 bit up and down counter. Counts up or down based on the pulse width modulation _int signal and generates terminal count whenever counter reaches the maximum value or when it transists through zero .terminal count is used to automatically load the data value to generate different pulse width modulation out with different duty cycle. Increment and Decrement is the two function used in the design for the counter up and down counting. They are defined in separate user package library.

```
PROCESS (clock, counter_out_intger, rco_intger, register_out)
BEGIN
IF (rco_intger = '1') THEN
  counter_out_intger <= register_out;
  ELSIF rising_edge (clock) THEN
    IF (rco_intger = '0' and pulse width modulation_intger ='1' and counter_out_intger
<"11111111") THEN
counter_out_intger <= INCRIMENT (counter_out_intger);
ELSE
    IF (rco_intger ='0' and pulse width modulation_intger ='0' and counter_out_intger
>"00000000") THEN
        counter_out_intger <= DECRIMENT (counter_out_intger);
        END IF;
        END IF;
        END IF;
        END PROCESS;
        PROCESS (counter_out_intger, rco_intger, clock, Reset)
        BEGIN
        IF (reset ='1') THEN
        rco_intger <='1';
        ELSIF rising_edge (clock) THEN
```

```
IF ((counter_out_intger ="11111111") or (counter_out_intger ="00000000")) THEN
rco_intger <= '1';
ELSE
rco_intger <= '0';
END IF;
END IF;
END PROCESS;
PROCESS (clock, rco_intger, Reset)
BEGIN
IF (reset ='1') THEN
pwm_intger <='0';
ELSIF rising_edge (rco_intger) THEN
Pulse width modulation_int <= NOT (pulse width modulation_int);
ELSE
Pulse width modulation_int <= pulse width modulation_int;
END IF;
END PROCESS;
Pulse width modulation <= pulse width modulation_int;

END Behavioral;
```

Signals and Variables are two fundamental types of objects used to carry data from place to place in a VHDL design description: Signals are assigned via “<=”! Variables are assigned via “:=”. Signals are used in the connection parts for VHDL entities. Variables are declared with in the process block, procedures, and functions. Signals can only be declared with in architecture bodies. They can be passed as parameters to functions and procedures. A process is a body of code which is executed whenever any of the signals it is sensitive to changes value. The Increment and Decrement function is defined in separate user package library. The syntax of the function is given below. [11]

```
Library IEEE;
Use IEEE.STD_LOGIC_1164.ALL;

        PACKAGE user_package is

        Function INCRIMENT(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
        Function DECRIMENT(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;

        END user_pkg;

        PACKAGE BODY user_pkg is

        Function INCRIMENT(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
        Variable declaration;
        BEGIN

End loop;

End INCRIMENT;

Function DECRIMENT(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
         Variable declaration:
         BEGIN

End loop;

End DECRIMENT;
End user_package;
```

# Chapter 5

## Simulation Results and Discussion

These chapters discuses about the synthesis and simulation of the design in xilinx Field Programmable Gate Array using Xilinx synthesis and simulation tool (XST). WebPack ISE design software offers a complete design suite based on the xilinx Foundation ISE series software. We are using the Spartan 3e FPGA device**.** Xilinx WebPack is the computer-aided design (CAD) environment which will Compile and synthesize a VHDL behavioral description for a specific target field programmable gate array (FPGA) architecture. Programmable logic design has entered an era in which device densities are measured in the millions of gates, and system performance is measured in hundreds of megahertz. Given these system complexities. Xilinx offers complete electronic design tools that enable the implementation of designs in Xilinx PLDs. These development solutions combine powerful technology with a flexible, easy-to-use graphical interface to achieve the best possible designs within the project. The availability of products such as WebPACK ISE software has made it much easier to design with programmable logic. Designs can be described easily and quickly using a description language such as VHDL or with a schematic capture package.

### 5.1 XST Synthesis

Logic synthesis is a process that starts from a high level of logic abstraction ( VHDL) and automatically creates a lower level of logic abstraction using a library containing primitives.

After design entry we can run synthesis. The ISE software includes Xilinx® Synthesis Technology (XST), which synthesizes VHDL to create Xilinx-specific netlist files known as NGC files. Unlike output from other vendors, which consists of an EDIF file with an associated NCF file, NGC files contain both logical design data and constraints. XST places the NGC file in

the project directory and the file is accepted as input to the Translate step of the Implement Design process. To specify XST as the synthesis tool, we must set the Synthesis Tool Project Property to XST.

The following figure shows the flow of files through the XST software.

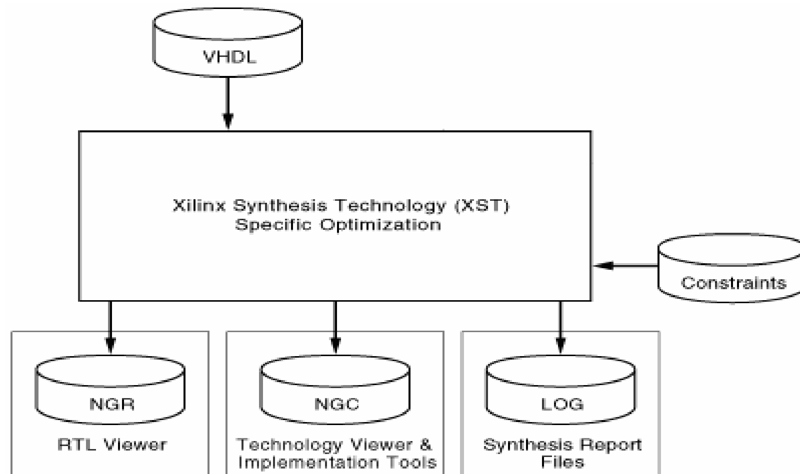



Figure 5.1 XST Design Flow Overview

## 5.2 Viewing a Synthesis Report

The synthesis report allows you to view the results of the netlist-generation synthesis process. In this report, we can see a summary of our synthesis, and a summary and analysis of the netlist generation. Viewing netlist in a graphical format enables to analyze how XST inferred components in the design. Schematics comprise one sheet or multiple sheets, depending on the size of the netlist and your Viewer Preferences settings.

There are two netlist viewers, the RTL Viewer and the Technology Viewer. Each viewer opens a different netlist file type, but many of the viewer functions are the same.

1. RTL schematic

This is a schematic representation of the optimized design shown at the Register Transfer Level (RTL). This representation is in terms of generic symbols, such as adders, multipliers, counters, AND gates, and OR gates, and is generated after the HDL synthesis phase of the synthesis process. The RTL Viewer enables you to view a Register Transfer Level (RTL) netlist as a schematic when you use Xilinx® Synthesis Technology (XST) as your synthesis environment

2. Technology schematic

This is a schematic representation of an NGC file shown in terms of logic elements optimized to the target architecture or "technology," for example, in terms of LUTs, carry logic, I/O buffers, and other technology-specific components. It is generated after the optimization and technology targeting phase of the synthesis process. Viewing this schematic allows to see a technology-level representation of HDL optimized for a specific Xilinx architecture.

The Technology Viewer enables you to view a Technology Level netlist as a schematic when you use Xilinx® Synthesis Technology (XST) as your synthesis environment.

**XST Detailed Design Flow**

The following figure shows each of the steps that take place during XST synthesis. The following sections describe each step in detail.

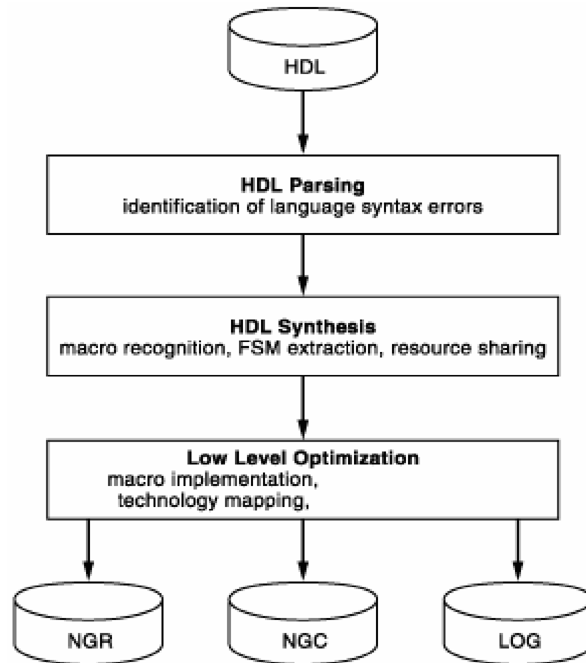



Figure 5.2 XST design flow

## HDL Parsing

During HDL parsing, XST checks whether your HDL code is correct and reports any syntax errors.

## HDL Synthesis

During HDL synthesis, XST analyzes the HDL code and attempts to infer specific design building blocks or macros (such as MUXes, RAMs, adders, and subtracters) for which it can create efficient technology implementations. To reduce the amount of inferred macros, XST performs a resource sharing check. This usually leads to a reduction of the area .

You can control the HDL synthesis step using constraints. You can enter constraints using any of the following methods:

- HDL source file

Enter VHDL attributes.

- XCF Enter global parameters and module-level constraints in the Xilinx constraints (XCF) file

Default property values are used for the Synthesize process, unless you modify them. .

## Low Level Optimization

During low level optimization, XST transforms inferred macros and general glue logic into a technology-specific implementation. The flow for FPGAs differs significantly at this stage as follows:

- FPGA Flow

The FPGA flow is timing-driven and can be controlled using constraints, such as PERIOD and OFFSET. During low level optimization, XST infers specific components, such as the following:

- Carry logic (MUXCY, XORCY, MULT_AND)
- RAM (block or distributed)
- Shift Register LUTs (SRL16, SRL16E, SRLC16, SRLC16E)
- Clock Buffers (IBUFG, BUFGP)
- Multiplexers (MUXF5, MUXF6, MUXF7, MUXF8)

The use of technology-specific features may come from a macro implementation mechanism or from general logic mapping. Due to mapping complexity issues, not all available FPGA features may be used. The FPGA synthesis flow supports advanced design and optimization techniques, such as Register Balancing, Incremental Synthesis, and Modular Design.

Compile Time Strategies: Use the following strategies to reduce synthesis compile time.

Incremental Synthesis

Incremental synthesis allows you to resynthesize only the modified portions of your design, which reduces your overall synthesis compile time. Following are the main types of incremental synthesis:

- Block level

  The synthesis tool resynthesizes the entire block if at least one modification was made in the block.

- Gate or LUT level

  The synthesis tool attempts to identify the exact changes made to the design and generates the final netlist with minimal changes.

XST supports block level incremental synthesis. To use the incremental synthesis flow with XST, you must divide your design into several partitions. XST considers each partition a separate unit and generates an NGC netlist file for each. If you make a change to one partition, XST only resynthesizes the partition you changed, and regenerates the corresponding NGC netlist file. During re-synthesis, XST is aware of all the partitions in the design and re-optimizes the changed block based on its context with the rest of the design.

Partitions

XST supports Partitions, which can help reduce runtime. By identifying the instances in your design as Partitions, your design is divided into separate logic groups. This allows you to change one Partition in your design while preserving other Partitions.

## 5.3 Simulation Overview

During HDL simulation, the simulator software verifies the functionality and timing of your design or portion of your design. The simulator interprets VHDL code into circuit functionality and displays logical results of the described HDL to determine correct circuit operation. Simulation allows you to create and verify complex functions in a relatively small amount of time. Simulation takes place at several points in the design flow. It is one of the first steps after design entry and one of the last steps after implementation, as part of verifying the end functionality and performance of the design. Simulation is an iterative process, which may require repeating until both design functionality is met. For a typical design, simulation comprises the following high-level steps:

1. Compilation of the simulation libraries
2. Creation of the design and test bench
3. Functional simulation
4. Implementation of the design

## 5.4 Test Benches

To simulate your design, you need both the design under test (DUT) or unit under test (UUT) and the stimulus provided by the test bench. A test bench is HDL code that allows you to provide a documented, repeatable set of stimuli that is portable across different simulators. A test bench can be as simple as a file with clock and input data or a more complicated file that includes error checking, file input and output, and conditional testing.

You can create the test bench using either of the following methods:

- Text editor

This is the recommended method for verifying complex designs. It allows you to use all the features available in the HDL language and gives you flexibility in verifying the design.

Although this method may be more challenging in that you must create the code, the advantage is that it may produce more precise and accurate results than using the Test Bench Waveform Editor.

To assist in creating the test bench, you can create a template that lays out the initial framework, including the instantiation of the UUT and the initializing stimulus for your design. Create this template as described in creating a Source File, selecting VHDL Test Bench as your source type.

Test Bench Strategies
Because the test bench becomes a part of the hierarchy in our code, the following is recommended:

- Make the test bench the top level of the code.

  The test bench instantiates the unit under test (UUT), and stimulus is applied from the top-level test bench to the lower-level design or portion of the design being tested.

- Use the instance name UUT for the instantiated unit under test.

  This is the default instance name that Project Navigator expects.

## 5.5 Xilinx Synthesis and simulation results

The functional description of VHDL model of an 8 bit PWM resolution is synthesized and simulated in xilinx Synthesis and Simulation Toll Environment .The Synthesis Tool (XST) contains View synthesis Report, View RTL schematic, View Technology Schematic and Check syntax.

Synthesis Summary of the PWM VHDL design

---- Target Parameters

Output File Name : "pwm_fpga"

Output Format : NGC

Target Device : xc3s100e-5-vq100

============================================================================
====
* HDL Compilation *
============================================================================
====
Compiling vhdl file "C:/Xilinx/pwm_fpga/pwm_fpga.vhd" in Library work.
Entity <pwm_fpga> compiled.
Entity <pwm_fpga> (Architecture <Behavioral>) compiled.
Package <user_package> compiled.
Package body <user_package> compiled.

============================================================================
====
* Design Hierarchy Analysis *
============================================================================
====
Analyzing hierarchy for entity <pwm_fpga> in library <work> (architecture <Behavioral>).

Building hierarchy successfully finished.

============================================================================
====
============================================================================
====
* HDL Analysis *
============================================================================
====
Analyzing Entity <pwm_fpga> in library <work> (Architecture <Behavioral>).
Entity <pwm_fpga> analyzed. Unit <pwm_fpga> generated.

==========================================================================
====
* HDL Synthesis *
==========================================================================
====

Performing bidirectional port resolution...

Synthesizing Unit <pwm_fpga>.
Related source file is "C:/Xilinx/pwm_fpga/pwm_fpga.vhd".
Found 8-bit comparator greater for signal <$cmp_gt0000> created at line 53.
Found 8-bit comparator less for signal <$cmp_lt0000> created at line 50.
Found 1-bit 4-to-1 multiplexer for signal <$mux0014> created at line 101.
Found 1-bit 4-to-1 multiplexer for signal <$mux0030> created at line 117.
Found 8-bit 4-to-1 multiplexer for signal <$mux0034>.
Found 8-bit register for signal <cnt_out_int>.
Found 1-bit register for signal <pwm_int>.
Found 1-bit register for signal <rco_int>.
Found 8-bit register for signal <reg_out>.
Summary:
inferred 10 D-type flip-flop(s).
inferred 2 Comparator(s).
inferred 10 Multiplexer(s).
Unit <pwm_fpga> synthesized.

.

HDL Analysis : The final report is given below

* Final Report *
==========================================================================
====
Final Results

RTL Top Level Output File Name : pwm_fpga.ngr

Top Level Output File Name : pwm_fpga

Output Format : NGC

Design Statistics
# IOs : 11

```
Cell Usage :
# BELS                          : 70
#      INV                      : 1
#      LUT2                     : 17
#      LUT2_D                   : 1
#      LUT3                     : 5
#      LUT3_L                   : 4
#      LUT4                     : 26
#      LUT4_D                   : 4
#      LUT4_L                   : 7
#      MUXF5                    : 4
#      VCC                      : 1
# FlipFlops/Latches             : 18
#      FDC                      : 9
#      FDCPE                    : 8
#      FDP                      : 1
# Clock Buffers                 : 1
#      BUFGP                    : 1
# IO Buffers                    : 10
#      IBUF                     : 9
#      OBUF                     : 1
=========================================================================
Asynchronous Control Signals Information:
----------------------------------------
-----------------------------------------------------+------------------------+-------+
Control Signal                                       | Buffer(FF name)        | Load  |
-----------------------------------------------------+------------------------+-------+
cnt_out_int_7__and0000(cnt_out_int_7__and00001:O)| NONE(cnt_out_int_7)    | 1     |
cnt_out_int_7__and0001(cnt_out_int_7__and00011:O)| NONE(cnt_out_int_7)    | 1     |
cnt_out_int_0__and0000(cnt_out_int_0__and00001:O)| NONE(cnt_out_int_0)    | 1     |
```

```
cnt_out_int_0__and0001(cnt_out_int_0__and00011:O)| NONE(cnt_out_int_0)    | 1     |
cnt_out_int_1__and0000(cnt_out_int_1__and00001:O)| NONE(cnt_out_int_1)    | 1     |
cnt_out_int_1__and0001(cnt_out_int_1__and00011:O)| NONE(cnt_out_int_1)    | 1     |
cnt_out_int_2__and0000(cnt_out_int_2__and00001:O)| NONE(cnt_out_int_2)    | 1     |
cnt_out_int_2__and0001(cnt_out_int_2__and00011:O)| NONE(cnt_out_int_2)    | 1     |
reset                                  | IBUF                   | 10    |
cnt_out_int_3__and0000(cnt_out_int_3__and00001:O)| NONE(cnt_out_int_3)    | 1     |
cnt_out_int_3__and0001(cnt_out_int_3__and00011:O)| NONE(cnt_out_int_3)    | 1     |
cnt_out_int_5__and0000(cnt_out_int_5__and00001:O)| NONE(cnt_out_int_5)    | 1     |
cnt_out_int_5__and0001(cnt_out_int_5__and00011:O)| NONE(cnt_out_int_5)    | 1     |
cnt_out_int_4__and0000(cnt_out_int_4__and00001:O)| NONE(cnt_out_int_4)    | 1     |
cnt_out_int_4__and0001(cnt_out_int_4__and00011:O)| NONE(cnt_out_int_4)    | 1     |
cnt_out_int_6__and0000(cnt_out_int_6__and00001:O)| NONE(cnt_out_int_6)    | 1     |
cnt_out_int_6__and0001(cnt_out_int_6__and00011:O)| NONE(cnt_out_int_6)    | 1     |
-----------------------------------------------+------------------------+-------+

Timing Summary:
---------------
Speed Grade: -5

   Minimum period: 6.617ns (Maximum Frequency: 151.132MHz)
   Minimum input arrival time before clock: 2.055ns
   Maximum output required time after clock: 4.604ns
   Maximum combinational path delay: No path found

Timing Detail:
--------------
All values displayed in nanoseconds (ns)

=========================================================================
====
Timing constraint: Default period analysis for Clock 'clock'
  Clock period: 6.617ns (frequency: 151.132MHz)
  Total number of paths / destination ports: 356 / 17
-------------------------------------------------------------------------
Delay:               6.617ns (Levels of Logic = 4)
  Source:            cnt_out_int_0 (FF)
  Destination:       cnt_out_int_6 (FF)
  Source Clock:      clock rising
  Destination Clock: clock rising
```

```
  Data Path: cnt_out_int_0 to cnt_out_int_6
                         Gate     Net
   Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
   ----------------------------------------  ------------
    FDCPE:C->Q          13   0.514   1.147  cnt_out_int_0 (cnt_out_int_0)
    LUT4_D:I0->O         3   0.612   0.774  _cmp_lt00002 (_cmp_lt00001)
    LUT4_D:I3->O         4   0.612   0.782  _and00001 (_and0000)
    LUT4:I3->O           1   0.612   0.684  _mux0034<6>57 (_mux0034<6>_map307)
    LUT4:I3->O           1   0.612   0.000  _mux0034<6>96 (_mux0034<6>)
    FDCPE:D                  0.268          cnt_out_int_6
----------------------------------------
   Total                          6.617ns (3.230ns logic, 3.387ns route)
                                   (48.8% logic, 51.2% route)

  Data Path: pwm_int to pwm_int
                         Gate     Net
   Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
   ----------------------------------------  ------------
    FDC:C->Q             8   0.514   0.921  pwm_int (pwm_int)
    INV:I->O             1   0.612   0.681  _not00051_INV_0 (_not0005)
    FDC:D                    0.268          pwm_int
   ----------------------------------------
   Total                 2.995ns (1.394ns logic, 1.601ns route)
                          (46.5% logic, 53.5% route)

Data Path: Data_value<0> to reg_out_0
                         Gate     Net
   Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
   ----------------------------------------  ------------
    IBUF:I->O            1   1.106   0.681  Data_value_0_IBUF (Data_value_0_IBUF)
    FDC:D                    0.268          reg_out_0
   ----------------------------------------
   Total                 2.055ns (1.374ns logic, 0.681ns route)
                          (66.9% logic, 33.1% route)
```

Device utilization summary:
---------------------------

Selected Device : 3s100evq100-5

| Logic Utilization | Used | Available | Utilization |
|---|---|---|---|
| Number of Slices | 34 | 960 | 3% |
| Number of Slice Flip Flops | 18 | 1920 | 0% |
| Number of 4 input LUTs | 65 | 1920 | 3% |
| Number of bonded IOBs | 11 | 66 | 16% |
| Number of GCLKs | 1 | 24 | 4% |

Table 5.1 Device utilization summary

Total memory usage is 140812 kilobytes

## 5.6 Simulation waveforms

For an 8 bit PWM resolution, resulting in $2^8$ different duty cycle states. Some of the duty cycle wave is shown below. The duty cycle of the PWM signal is controlled by the data value loaded to the up and down counter. The higher the data value the higher the duty cycle.

| Data value | Duty Cycles (%) |
|---|---|
| 00011001 | 10 |
| 01000000 | 25 |
| 10000000 | 50 |
| 11000000 | 75 |
| 11100110 | 90 |

Table 5.2: Some of the Data Values for different Duty Cycles (N=8)

The following figure shows the 50% and25% duty cycle modulated signal.

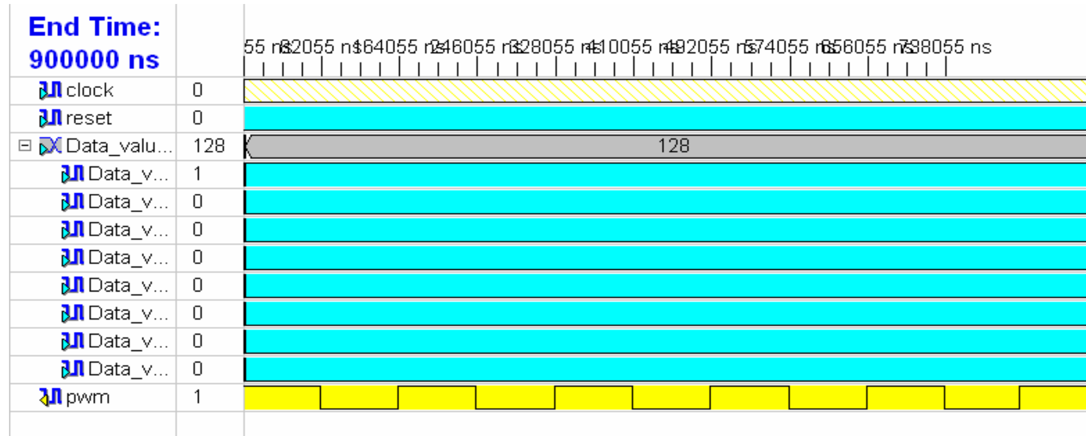



Figure 5.3. Waveform 1: pulse with 50% Duty cycle

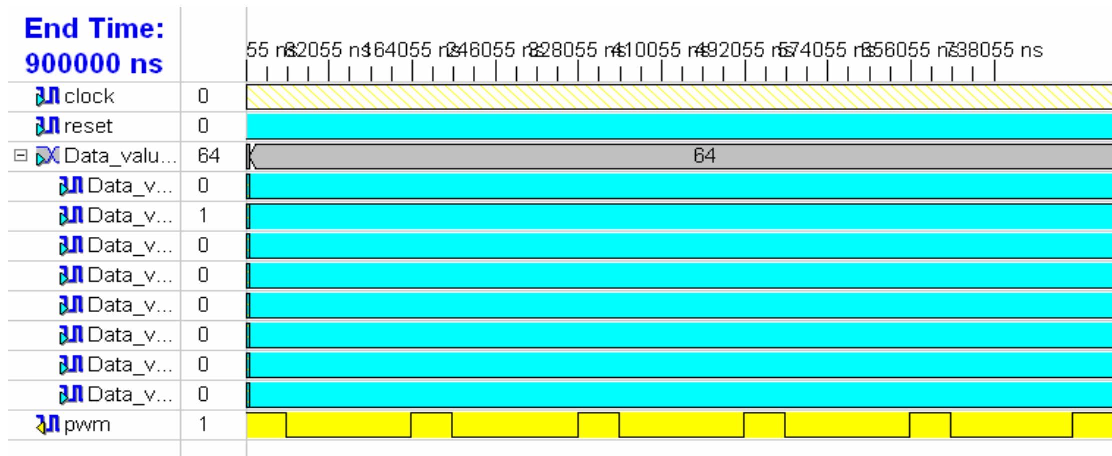



Figure 5.4. Waveform 2: pulse with 25% Duty cycle

| Signal | type | Description |
|---|---|---|
| Clock | Input | PWM clock. The duty cycle is a function of the clock. |
| Reset | Input | Reset counter and the pulse to 0 state. |
| Data_value | input | Set the duty cycle of the output signal |
| Pwm | output | Modulated signals. High time is set by data_value |

Table 5.3. PWM user interface signal description

## 5.7 View technology schematic results

The RTL Schematic Viewer and the Technology Viewer are shown below. Each viewer opens a different netlist file type, but many of the viewer functions are the same.

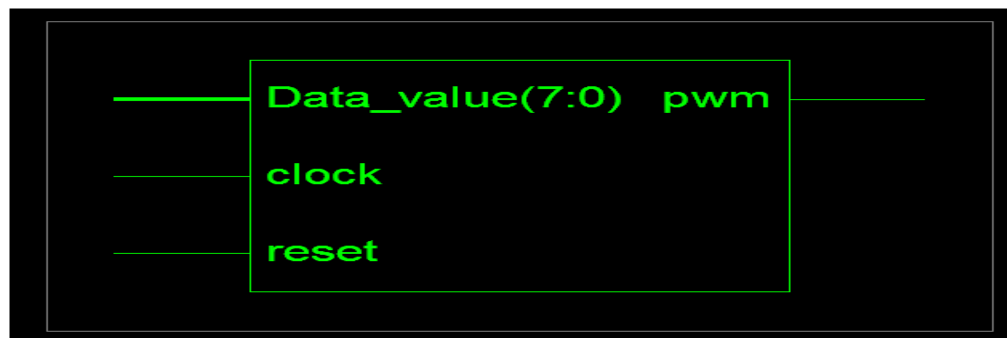



Figure 5.5. PWM user interface signal

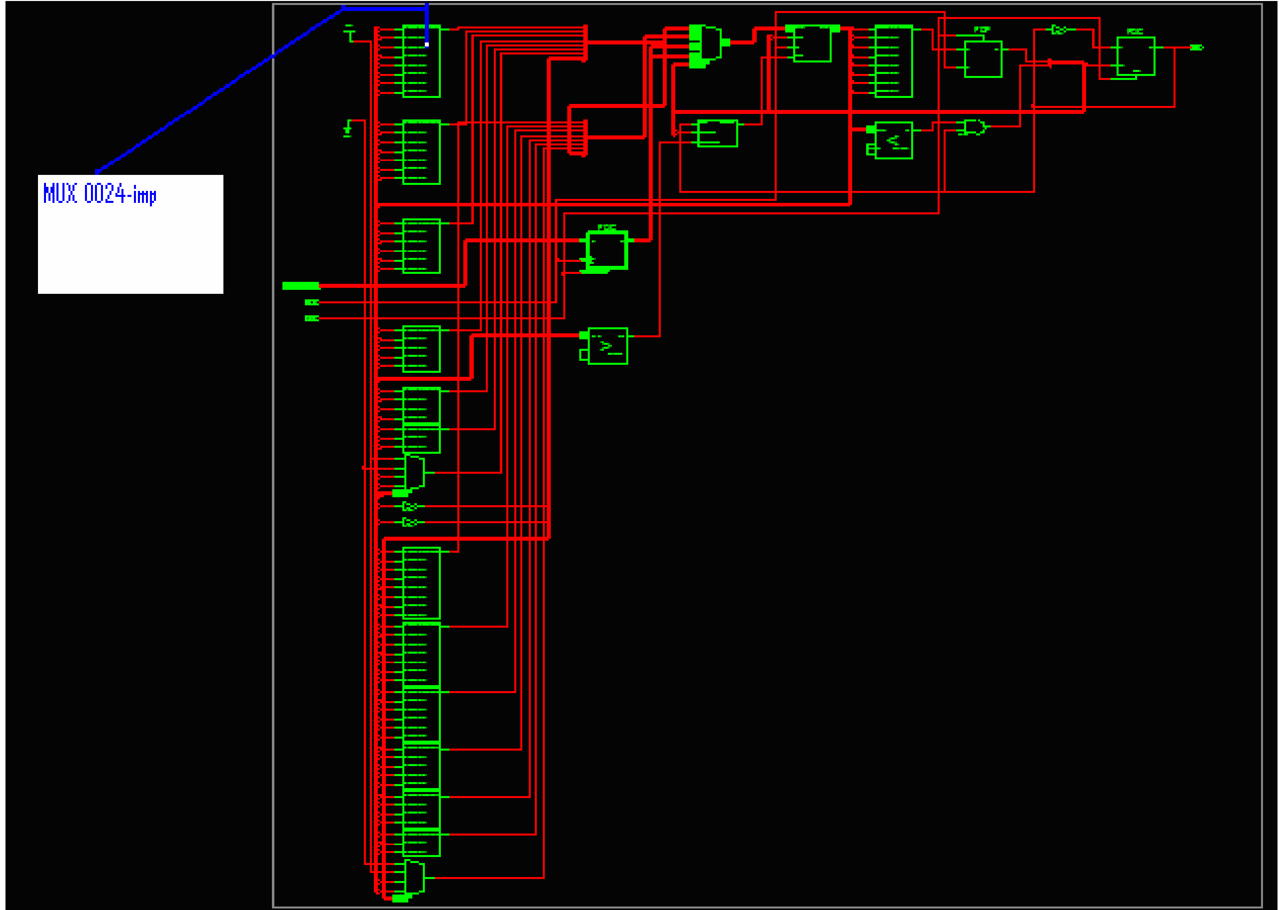
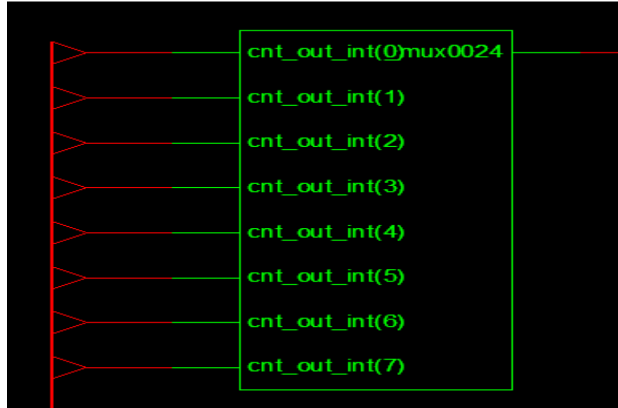
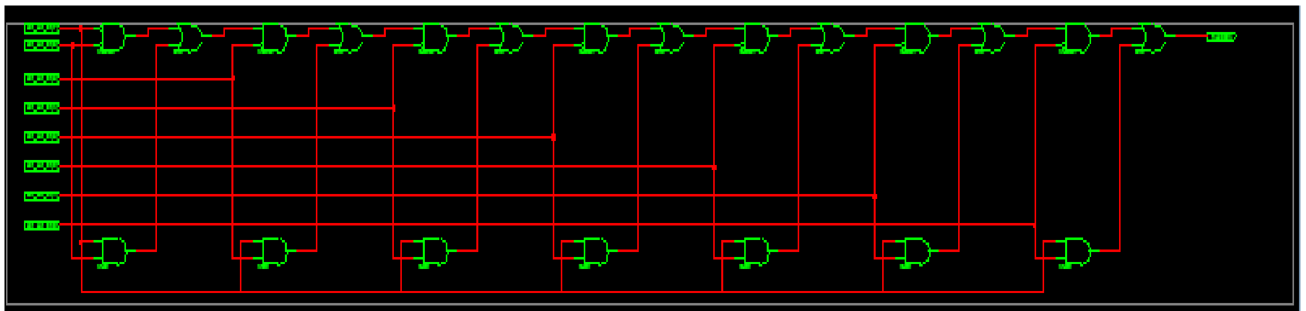



Figure 5.6. RTL schematic view of PWM

This RTL schematic view of PWM (Fig 5.6) representation is in terms of generic symbols, such as adders, multipliers, counters, AND gates, and OR gates. The first generic symbols of the RTL schematic view of PWM shown in MUX 0024-imp is equivalent to the next figure 5.7.



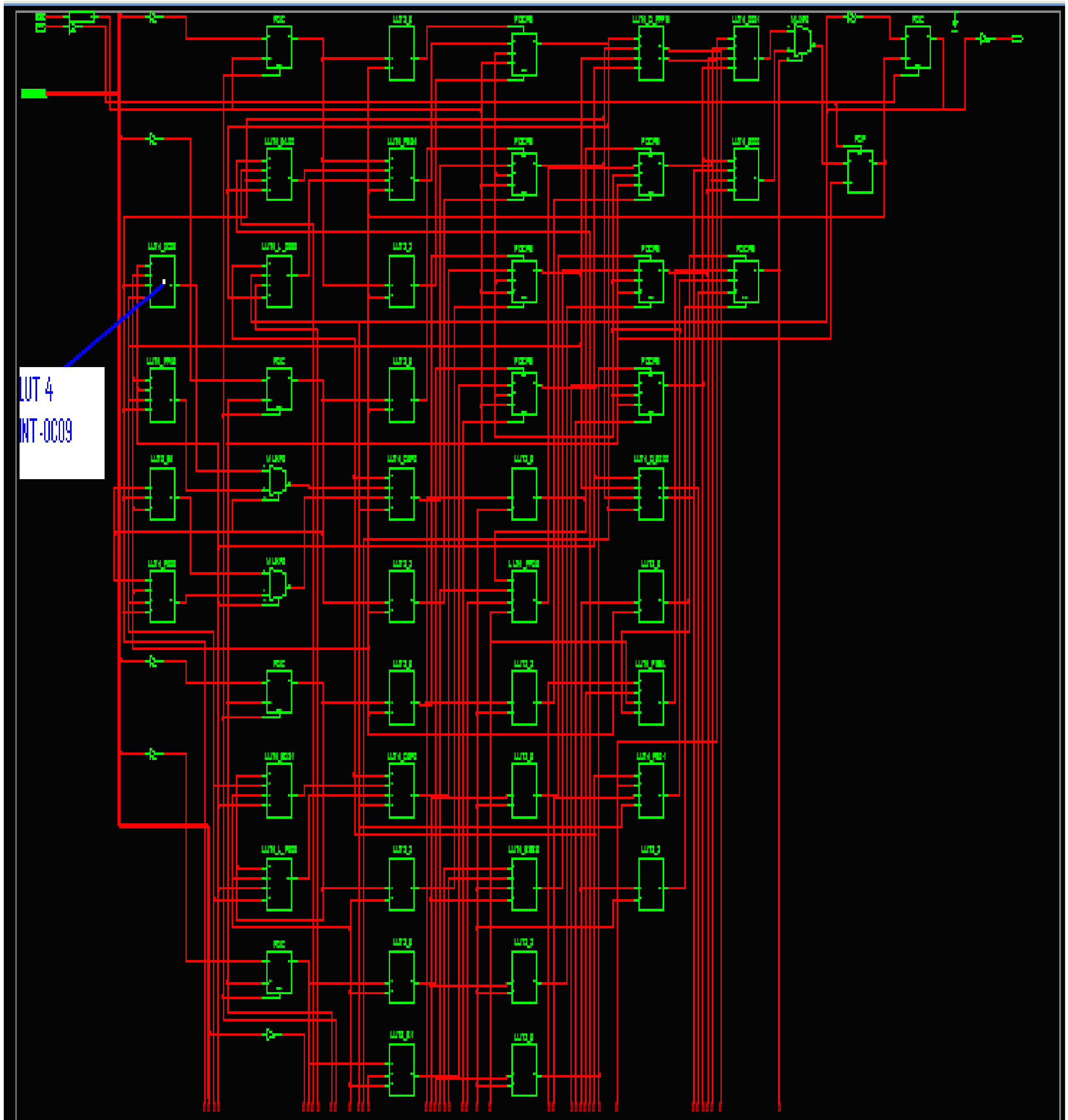

(a)



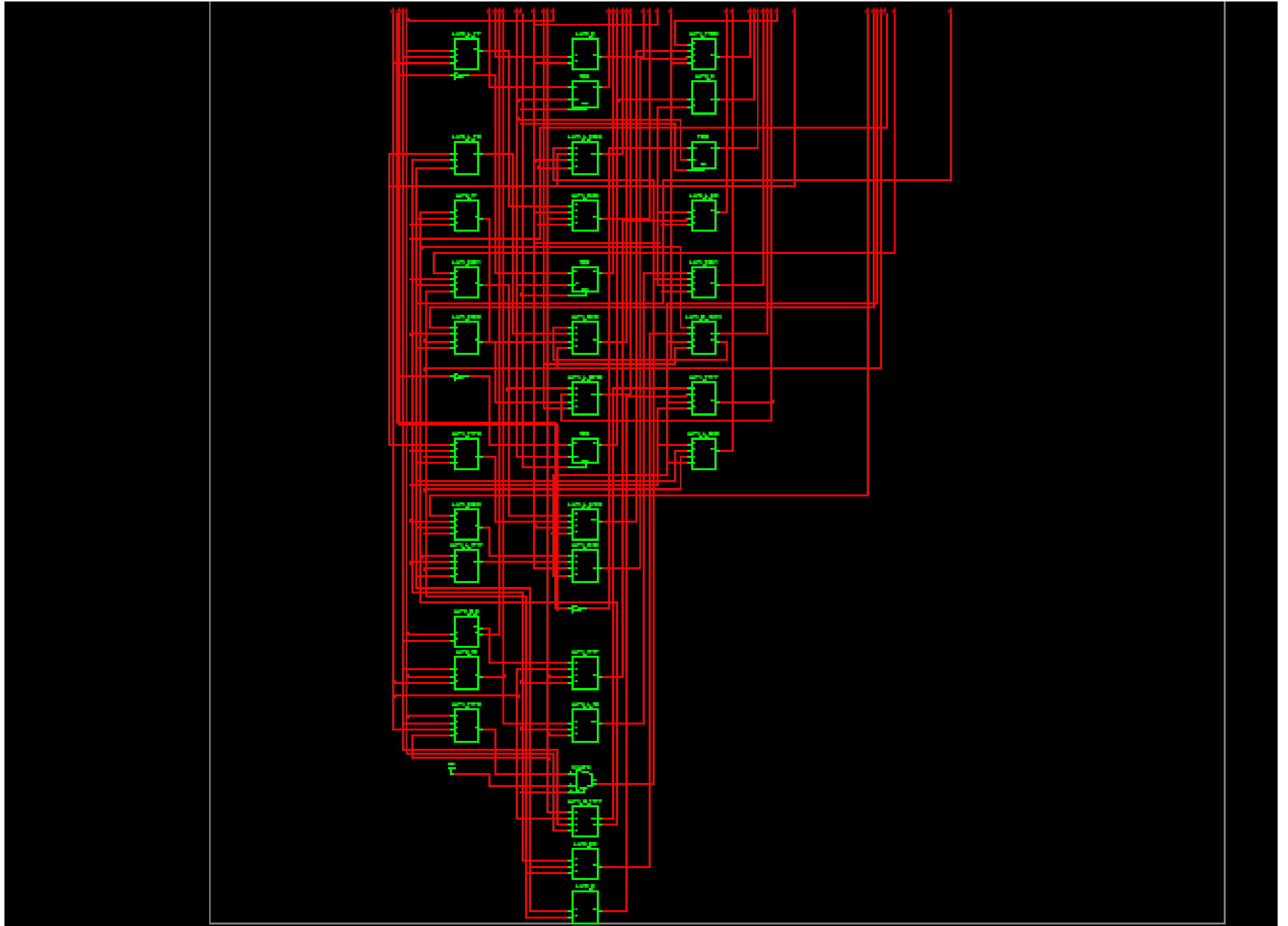
(b)

Figure5.7. (a) and (b) Equivalent Generic schematic of PWM of –MUX 0024-imp

The technology schematic viewers of the PWM in xilinx XST shows the design of PWM in terms of logic elements optimized to the target architecture in terms of LUTs, I/O buffers and others. The following figure 5.8 shows the technology schematic of the PWM design.




(a)



(b)

Figure 5.8. (a) and (b) Technology schematic view of PWM

As can be seen from figure 5.8 of the technology schematic of the PWM design the schematic view shows in terms of LUT. For example one of the equivalent views of Look-up Table of LUT 4 INT-OC9 shown above is given below in terms of function generators of logic gates.

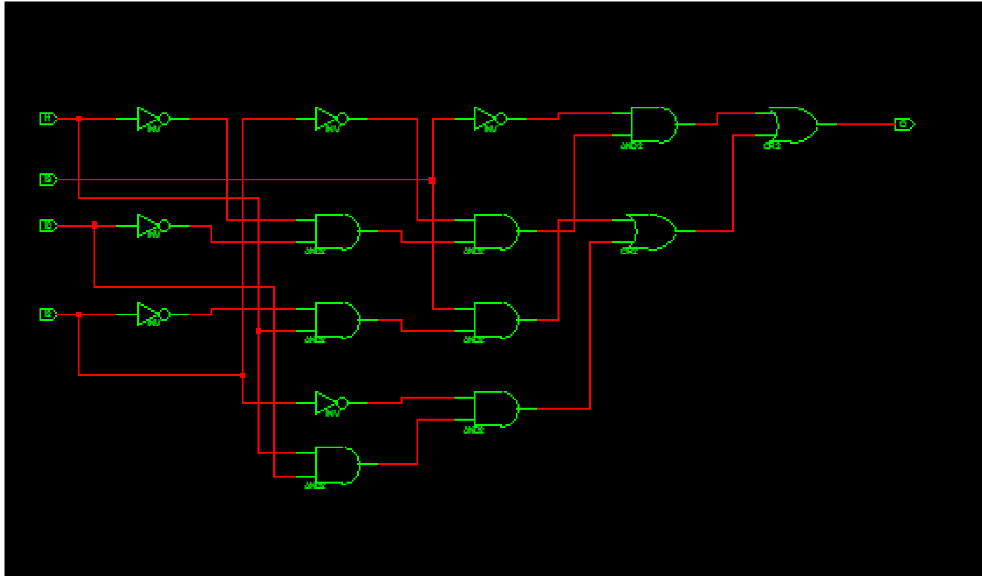


Figure 5.9.The equivalent Look-up Table function generators of LUT 4 INT _OC09 in fig 5.8.

# Chapter 6

# Conclusions & Future Work

## 6.1. Conclusions

Using Xilinx Integrated System environment 8.2i software we can develop the proposed PWM in Xilinx FPGA. Due to the need of design flexibility in FPGA, an 8 bit resolution PWM was developed using VHDL modeling in Xilinx Field Programmable Gate Array. A VHDL model was implemented on Spartan 3e and optimized for area. The simulation results prove that using the proposed method, PWM can be produced with a duty cycle resolution of 0.39%, which is adequate for most applications. The low percentage of the device logic blocks occupied by the PWM implementation permits the integration of multiple control operations in a single IC. The selection of the target device depends on the system cost and resolution requirements. In addition to this the VHDL modeling and the architectural features of XILINX FPGAs is studied. The PWM developed can be used in many diverse and complex applications like robotics, motor and motion control.

## 6.2 Future work

Depending upon the application the requirement of the resolution is different. it is evident that higher values of n provide better resolution of the duty cycle, but performance should be taken into consideration when doing so. Future work should include high resolution of the duty cycle

and high frequency application. Since Many FPGA vendors are available in the market, there fore the performance of XILINX FPGA with others using a powerful simulator should be studied.

# Appendix A

## VHDL Listing

```
1-- Company:
2-- Create Date:    19:37:12 11/25/2007
3-- Design Name: PWM in xilinx FPGA
4-- Module Name:    Pulse Width Modulation - Behavioral
5-- Project Name: Design of PWM
6-- Target Devices: xc3s100e-5v100
7-- Tool versions: ISE 8.2i
8 library IEEE;
9 use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.STD_LOGIC_ARITH.ALL;
11 use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13       Entity pulse width modulation is
14
15    Port (
16              clock: in STD_LOGIC;
17               Reset: in STD_LOGIC;
18               Data_value: in STD_LOGIC_VECTOR (7 downto 0);
19               Pulse width modulation: out STD_LOGIC);
20
21               End pulse width modulation;
22
23     Architecture Behavioral of pulse width modulation is
24
25     SIGNAL register_out: std_logic_vector (7 downto 0);
```

```
26   SIGNAL counter_out_intger: std_logic_vector (7 downto 0);
27   SIGNAL pulse width modulation_integer, rco_intger: std_logic;
28
29     Begin
30
31    Process (Clock, register_out, Reset)
32    BEGIN
33   IF (reset ='1') THEN
34   regester_out <="00000000";
35   ELSIF (rising_edge (clock)) THEN
36  regester_out <= data_value;
37  END IF;
38
39  END PROCESS;
40  PROCESS (clock, counter_out_intger, rco_intger, register_out)
41  BEGIN
42 IF (rco_intger = '1') THEN
43 counter_out_intger <= register_out;
44 ELSIF rising_edge (clock) THEN
45 IF (rco_intger = '0' and pulse width modulation_intger ='1' and counter_out_intger
46<"11111111") THEN
47 counter_out_intger <= INCRIMENT (counter_out_intger);
48 ELSE
49  IF (rco_intger ='0' and pulse width modulation_intger ='0' and counter_out_intger
50>"00000000") THEN
51       counter_out_intger <= DECRIMENT (counter_out_intger);
52       END IF;
53       END IF;
54       END IF;
```

```
55  END PROCESS;
56
57  PROCESS (counter_out_intger, rco_intger, clock, Reset)
58      BEGIN
59
50      IF (reset ='1') THEN
61      rco_intger <='1';
62      ELSIF rising_edge (clock) THEN
63      IF ((counter_out_intger ="11111111") or (counter_out_intger ="00000000")) THEN
64      rco_intger <= '1';
65      ELSE
66      rco_intger <= '0';
67      END IF;
68      END IF;
69
70      END PROCESS;
71
72      PROCESS (clock, rco_intger, Reset)
73      BEGIN
74
75      IF (reset ='1') THEN
76      pwm_intger <='0';
77      ELSIF rising_edge (rco_intger) THEN
78      Pulse width modulation_int <= NOT (pulse width modulation_int);
79      ELSE
80      Pulse width modulation_int <= pulse width modulation_int;
81
82      END IF;
83      END PROCESS;
```

```
84   Pulse width modulation <= pulse width modulation_integer;
85
86        END Behavioral;
87
88
89        PACKAGE user_package is
90
91        function INCRIMENT(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
92        function DECRIMENT(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR;
93
94       END user_package;
95
96       PACKAGE BODY user_package is
97       function INCRIMENT(X: STD_LOGIC_VECTOR) return STD_LOGIC_VECTOR is
98       variable XV: STD_LOGIC_VECTOR (X'LENGTH - 1 downto 0);
99
100      BEGIN
101       XV := X;
102      for I in 0 to XV'HIGH LOOP
103      IF XV (I) = '0' then
104     XV (I):='1';
105   exit;
106  else  XV (I) :='0';
107  end if;
108 end loop;
109
110   return XV;
111   end INCRIMENT;
```

```
112     Function DECRIMENT(X: STD_LOGIC_VECTOR) return   STD_LOGIC_VECTOR is
113       Variable XV: STD_LOGIC_VECTOR (X'LENGTH - 1 downto 0);
114       BEGIN
115       XV := X;
116           For I in 0 to XV'HIGH LOOP
117           IF XV (I) = '1' then
118     XV (I):='0';
119    exit;
120         else  XV(I) := '1';
121       end if;
122     End loop;
123
124      return XV;
125    End DECRIMENT;
126
127    End user_package;
```

----------------------------------------------------------------------------------------------------------

----------------------------------------------------------------------------------------------------------

# References

[1] Stephen Brown, “Fundamentals of digital logic with VHDL design,” July 2002.

[2] Gwaltney, “FPGA Implementation of controls,” 2003.

[3] Peter J. Ashendenal , “The VHDL cookbook ’’ First Edition, July 1990, South Australia.

[4] Karen Parenell, Nick Mehta,“ Programmable logic design Quick start Hand Book, ”Xilinx, Second edition, January 2002.

[5] Karen Parenell , Nick Mehta, “ Programmable logic design Quick start Hand Book, ” Xilinx, Third edition, January 2002.

[6] Angel Vpeterchev, “Digital pulse width modulation in power electronic circuits,” July2002.

[7] S.poorani, K.udays Kumar, “FPGA based Fuzzy Logic controller for electric vehicle,” Vol.45 issue 5, 2005 Journal of IOE, Singapore.

[8] Xilinx Integrated System Enviroment 2007 Manuel, Ver 8.2i.

[9] Joannis Sourdis, “Efficient and High speed FPGA based string matching for Packet Inspection,” Msc thesis, Chania, July 2004.

[10] Xilinx official Web site, http//www.xilinx.com.

[11]Dougleas Perry, “VHDL,” Third edition, 2003.

[12]Sunggu Lee, “Design of Computers and other complex digital devices,” July 2000.

[13] Tomas, “the low carb Very High Sped Integrated Circuit Hardware Description Language Tutorial,” 2003.

[14]Alexander Prodic,” Design and Implementation of a digital PWM controller for a High-frequency switching DC-DC Power Converter,” IECON’01, the 27th Annual conference of the IEEE industrial electronics society, 2001, USA.

[15] M. Wang, A. Ranjan, S. Raje, “Multi-Million Gate FPGA Physical Design Challenges”, ICCAD, pp. 891-898, 2004.

[16] J. M. Emmert, and D. Bhatia, “A Methodology for Fast FPGA Floorplanning”, Proc. FPGA, 1999.

[17] R. Ramos, X. Roset, A. Manuel, Implementation of fuzzy logic controller for DC/DC converters using field programmable gate array, in: Proc. 17th IEEE Instrumentation and Measurement Technology Conference, vol. 1, 2000, pp. 160–163.

[18] M.M. Islam, D. Allee, S. Konasani, A. Rodriguez, A lowcost digital controller for a switching DC converter with improved voltage regulation, IEEE Power Electronics Letters 2 (2004) 121–124.

[19] http://www.digilentinc.com (Xilinx I/O board).

[20]Diter Metzner, George Parz, “HDL based system engineering for Automotive Power Applications,” D-81609 Munchen, Germany, 2003.

[21] Patrick Pelgrims, Tom Tirens,” Learning about VHDL and FPGA’s,”Version 1.0,2003.

[22] http://www.quicklogic.com/support(official web site).

[23] Xilinx. Spartan 3E Platform FPGAs: Complete data sheet. DS031 v3.3, June 2005.

[24] P. J. Ashenden, “The Designer’s Guide to VHDL,” 2nd ed. Morgan Kaufmann Publishers, 2002.

[25] N. Mohan, T. Undeland, W. Robbins,” Power Electronics Converters, Applications and Design”, second ed., Wiley,1995.