VHDL Coding Style Guidelines and Synthesis:
A Comparative Approach

by

SHAHABUDDIN L. INAMDAR

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Electrical Engineering
Department of Electrical Engineering
College of Engineering
University of South Florida

Major Professor: Wilfrido A. Moreno, Ph.D.
James T. Leffew, Ph.D.
Sanjukta Bhanja, Ph.D.

Date of Approval:
October 25, 2004

Keywords: Altera, Xilinx, Portability, FPGA, VHDL Subset

© Copyright 2004, Shahabuddin Inamdar
ACKNOWLEDGEMENTS

My grateful acknowledgement is given to all those who were responsible for making this thesis possible. I would like to thank:

- My Major Professor, Dr. Wilfrido A. Moreno for his exceptional support and guidance. I would also like to sincerely thank Dr. James T. Leffew and Dr. Sanjukta Bhanja for being part of my examining committee.
- Mr. Harris Ioannidis, Mr. Trung Nguyen and other staff members of the Electrical Engineering Department for assisting with all the Synthesis tools and network support.
- Technical Specialist from Xilinx.com, Mr. David Dye. I am grateful for his valuable support and guidance.

I extend my gratitude to my parents, relatives and friends for standing by me on all occasions and encouraging me at every moment of my life.
TABLE OF CONTENTS

LIST OF TABLES iv

LIST OF FIGURES v

ABSTRACT vi

CHAPTER 1. INTRODUCTION 1

1.1 Introduction To VHDL Coding Styles And Synthesis 1
1.2 Design Flow During Synthesis 3
  1.2.1 Analysis 3
  1.2.2 Elaboration 3
  1.2.3 Initialization 4
  1.2.4 Simulation 4
  1.2.5 Synthesis 4
1.3 Post Synthesis Design Flow 4
  1.3.1 Translate 5
  1.3.2 Mapping 5
  1.3.3 Place And Route, (PAR) 5
  1.3.4 Bit Stream Generation 6
  1.3.5 Configuration 6
1.4 Synthesis Tools 7
1.5 Benefits Of Logic Synthesis 7
1.6 Thesis Organization 9

CHAPTER 2. VHDL CODING STYLES AND METHODOLOGIES 10

2.1 Understanding VHDL Coding Styles 10
2.2 Coding Style Requirements 12
2.3 General Coding Guidelines 12
2.4 Coding Style Guidelines For Synchronous Systems 14
2.5 Design Guidelines For Synthesis 18
CHAPTER 3. SYNTHESIS FEATURES AND INTERPRETATION

3.1 What Is Synthesis? 20
3.2 Inputs To Synthesis 20
3.3 VHDL Subset For Synthesis 21
   3.3.1 Supported VHDL Constructs 21
   3.3.2 Unsupported VHDL Constructs 22
   3.3.3 Ignored VHDL Constructs 22
3.4 Mapping Of VHDL Constructs Into Logic Network 23
   3.4.1 Signals 23
   3.4.2 Variable 24
   3.4.3 Wait Statements 24
      3.4.3.1 Wait On <Sensitivity List> 24
      3.4.3.2 Wait Until <Condition> 25
   3.4.4 Process Statement 25
   3.4.5 Conditional Signal Assignment Statement 25
   3.4.6 Selected Signal Assignment Statement 27
   3.4.7 If-Then-Else Statement 28
   3.4.8 If-Then-Elsif-Else Statement 29
   3.4.9 If Without Else Statement 30
   3.4.10 The Case Statement 31
   3.4.11 For-Loop Statement 32
   3.4.12 Three-State Logic 33
      3.4.12.1 D-Type Flip-Flop 35
      3.4.12.2 Synchronous Reset 36
3.4.13 Bi-Directional Three-State Buffer 34
3.4.14 Sequential Domain 35
   3.4.14.1 D-Type Flip-Flop 35
   3.4.14.2 Synchronous Reset 36

CHAPTER 4. COMPARISON OF DIFFERENT DESIGN METHODOLOGIES 37

4.1 Need For A Comparison 37
   4.1.1 Direct Competitor Comparison 39
4.2 Comparison Between Altera’s QuartusII And Xilinx’s ISE Webpack 39
   4.2.1 Advantages And Disadvantages Of Two Design Environments 40
      4.2.1.1 Advantages Of Altera’s QuartusII 40
      4.2.1.2 Disadvantages Of Altera’s QuartusII 41
      4.2.1.3 Advantages Of Xilinx’s ISE Webpack 41
      4.2.1.4 Disadvantages Of Xilinx’s ISE Webpack 43
   4.2.2 Portability Of The Synthesis Code 43
   4.2.3 Optimization Of VHDL Synthesis For A Target Technology 52
      4.2.3.1 Logic Elements Of The Flex10K Family 52
      4.2.3.2 Configurable Logic Block Of The SpartanII Family 53
      4.2.3.3 Implementation Of The Design 54
         4.2.3.3.1 Using Altera’s QuartusII 55
         4.2.3.3.2 Using Xilinx’s ISE Webpack 56
LIST OF TABLES

Table 3.1: Truth Table For A Bi-directional Three-State Buffer 34
Table 4.1: Different Vendors And Their Synthesis Tools 38
Table 4.2: Direct Competitor Comparison 39
Table 4.3: FLEX10K-EPF10K20 Device Performance Results 55
Table 4.4: SpartanII-XC2S100 Device Performance Results 56
Table 4.5: Xilinx ISE Performance With And Without Constraints 58
LIST OF FIGURES

Figure 1.1: VHDL Design Flow 6

Figure 2.1: A Race Condition 17

Figure 2.2: Solution for a Race Condition 17

Figure 3.1: Mapping of a Signal 23

Figure 3.2: Mapping of a Conditional Signal Assignment Statement 26

Figure 3.3: Mapping of a Selected Signal Assignment Statement 27

Figure 3.4: Mapping of an If-Then-Else Statement 28

Figure 3.5: Mapping of an If-Then-Elsif-Else Statement 29

Figure 3.6: Mapping of an If without Else Statement 30

Figure 3.7: Mapping of a Case Statement 31

Figure 3.8: Mapping of the For-Loop Statement 32

Figure 3.9: Mapping of Three State Logic 33

Figure 3.10: Mapping of a Bi-directional Three-State Buffer 34

Figure 3.11: D-Type Flip-Flop 35

Figure 3.12: Mapping of a Synchronous Reset 36

Figure 4.1: Logic Block of the FLEX10K Family 53

Figure 4.2: A Single Slice of the SpartanII CLB 54
ABSTRACT

With the transistor density on an integrated circuit doubling every 18 months, Moore’s law seems likely to hold for another decade at least. This exponential growth in digital circuits has led to its increased complexity, better performance and is quickly getting less manageable for design engineers.

To combat this complexity, CAD tools have been introduced and are still being continuously developed, which prove to be of great help in the digital industry. One of the technologies, that is rapidly evolving as an industry standard, is the Very High Speed Integrated Circuit Hardware Description Language, (VHDL), language. The VHDL standard language along with logic synthesis tools are used to implement complex digital systems in a timely manner.

The increase in the number of specialist design consultants, with specific tools accompanied by their own libraries written in VHDL, makes it important for a designer to have an in-depth knowledge about the available synthesis tools and technologies in order to design a system in the most efficient and reliable manner.
This research dealt with writing VHDL code in terms of hardware modeling, based on coding styles, in order to get optimum results. Furthermore, it dealt with the interpretation of VHDL code into equivalent optimized hardware implementations, which satisfy the constraints of a set of specifications. In order to obtain a better understanding of the different VHDL tools and their usefulness in different situations, a comparative analysis between Altera’s QuartusII and Xilinx’s ISE Webpack tools, was performed. The analysis compared their Graphics User Interface, VHDL Code Portability and VHDL Synthesis constraints. The analysis was performed by designing and implementing a screensaver circuit on an FPGA and displaying it on the VGA Monitor.
CHAPTER 1

INTRODUCTION

1.1 Introduction To VHDL Coding Styles And Synthesis

Staying competitive in today’s FPGA market means designing IC’s with greater functionality, higher speed and lower cost, [3]. Designing and implementing any digital circuit using a Hardware Description Language involves selecting a proper Synthesis Tool, writing VHDL code for the circuit, simulating the code, synthesizing the code, implementing the design in a target device and then programming the device by downloading the code into the device. Thus, writing VHDL code and selecting a synthesis tool represent the basics for a digital design at the present time.

The style of writing VHDL code is very important. Effective VHDL coding techniques can make all the difference between designs that meet tough synthesis targets and verification schedules versus those requiring re-spins, [3]. Vector independent code is important. Code reuse depends on the code being portable between different synthesis tools. Each synthesis tool vendor accepts a slightly different subset of VHDL. Hence the same code gives different results on different synthesis tools. Maintaining some basic coding guidelines will produce effective results on most of the tools. With respect to any section of code, the only trade-off available with a synthesis tool is timing versus area.
Some authors relate this trade-off in terms of low-power design. However, good code can result in a design that is both smaller in area and faster.

Synthesis can be defined as the process of converting the VHDL code into a Gate Level Netlist with the help of a synthesis tool. This conversion process may be performed manually with the help of the schematic tools. However, with the digital circuit devices becoming more and more complicated and the gate count reaching as high as 100,000 and above, it is impractical to continue using the same old conventional methods. A better method for synthesis is the use of electronic design automation, (EDA), tools that convert hardware description language into an optimized gate level design. One of the advantages of using the HDL is that it is very easy to define a system in terms of software and then use the synthesis tool to produce the desired hardware. The hardware language used in this research was VHDL, which provides a technology independent description of the electronic circuit.

Synthesis represents an important activity in today’s digital design environment. While designing digital circuitry for complex ASIC or FPGA / PLD the designer relies on the logic synthesis to accomplish the task. With the present pace of the micro-electronics industry, the market windows are getting shorter. Hence, the designs must be produced in a shorter time scale while maintaining the high quality. Synthesis is one of the efficient methods that help the designer to design at a higher level without involving the drudgery of manual gate level design. Therefore, the designer can concentrate on the more important aspects of the design such as architecture and algorithm development.
1.2 Design Flow During Synthesis

The detailed process of synthesis can be described as a sequence of the following events:

- Analysis,
- Elaboration,
- Initialization,
- Simulation,
- Synthesis.

1.2.1 Analysis

It consists of compiling the .vhd file and storing it in a design library. During compilation, the compiler checks the syntax and semantics of the .vhd file and converts it into an intermediate form, which is stored in a specific design library. Finally, during analysis, the analyzer maps the equation based form of design to a standard cell library.

1.2.2 Elaboration

During the elaboration phase, flattening of the design hierarchy takes place. Elaboration involves the following tasks:

- Initialization of signals, processes and variables,
- Components are bound to architectures,
- Memory is allocated for storage of various objects.

The final result of the elaboration process is a flat collection of the signal nets and processes.
1.2.3 Initialization

The initialization process follows elaboration. Initialization involves the following tasks:

- Simulation time is reset to 0 ns,
- Explicit signals of ports and signals of architectures are initialized, resolved and assigned values,
- Signals declared implicitly by attributes are assigned values,
- Processes are executed until they suspend.

1.2.4 Simulation

Simulation follows Initialization. During simulation execution of the processes takes place.

1.2.5 Synthesis

After the design has been successfully simulated, the synthesis stage converts the text-based design into a gate level Netlist. This Netlist is a non-readable file that describes the actual circuit to be implemented at a very low level.

1.3 Post Synthesis Design Flow

Once synthesis is complete the design is implemented on the chip by sequentially following the phases:
• Translation,
• Mapping,
• Placement and Routing, (PAR).

During the implementation phase the design Netlist and normally a constraints file is used to recreate the design using the available resources within the FPGA.

1.3.1 Translate

This stage prepares the synthesized design for use within the FPGA. It checks the design and ensures that the Netlist is consistent with the chosen architecture. The result is stored in a file in a tool specific binary format that describes the logic design in terms of design primitives such as latches, Flip-Flops and Function Generators.

1.3.2 Mapping

In this stage, the design is distributed to the resources in the FPGA. Thus, mapping assigns a design’s logic elements to the specific physical elements such as CLBs and IOBs that actually implement logic functions in the device.

1.3.3 Place And Route, (PAR)

During the placement phase of the PAR, the design blocks created during mapping are assigned specific locations in the FPGA. The Routing phase assigns the interconnect paths in the FPGA.
1.3.4 Bit Stream Generation

The output obtained from the Place and Route phase is converted into a bit file, which is used to configure the FPGA.

1.3.5 Configuration

During configuration, the bit file is downloaded from the programming file into the SRAM of the target device. The complete design flow can be represented graphically as depicted in Figure 1.1.

Figure 1.1: VHDL Design Flow
1.4 Synthesis Tools

Synthesis tools play a vital role in digital electronic design automation. It accepts the conceptual hardware description language design definition as the input and provides the corresponding physical or logical representation for the targeted silicon device as an output, [3].

To produce highly optimized results with a fast compile time and quick turn around time a state of the art synthesis engine is required, [3]. The synthesis engine must be integrated with the physical implementation tool and should have the ability to proactively meet the design timing requirements. Most of the modern synthesis tools are robust, reliable and produce high quality optimized circuits from the initial Register Transfer Level,(RTL), VHDL designs. All of the VHDL features such as Access Types and Files, which are abstractions that are too unconnected with the hardware, cannot be synthesized. Therefore, each synthesis tool builder defines a subset of the VHDL language for input to the product. The size of the VHDL subset forms the basic criteria for choosing the synthesis tool. A large subset gives more freedom when refining the system description but does not imply optimum efficiency of the result. Another criteria is the existence of a number of possible target libraries.

1.5 Benefits Of Logic Synthesis

- Logic synthesis increases designer productivity. A few lines of RTL code is equivalent to hundreds of logic gates. Therefore, designs can be created and verified more quickly and efficiently at the RTL level.
**Different Design Options:** High level constructs available in VHDL and friendly graphical interface tools help the designer to experiment with the design in order to obtain the best possible solution by using the various available options.

**Improvement in Quality:** State-of-the-art synthesis tools available are highly reliable and optimized to produce quality circuits that meet the design criteria in all respects.

**Consistency between Levels of Hierarchy:** Automating the transfer of the hierarchical circuit from the RTL level to the gate level during synthesis guarantees that the design remains functionally identical during the transfer. Thus, there is no chance for any error introduction or inconsistencies, which might occur during manual transfer.

**Ease of Layout Design:** The silicon layout of the design is an important factor in design implementation. Most of the design layout is generated by the automated tools using advanced placement and routing algorithms and standard cell libraries.

**Technology independence:** Prior to the final commitment of the design to the hardware, the synthesis tool allows the designer to experiment with alternative hardware technologies such as ASIC, CPLD or FPGA.

**Design re-use and sharing:** The same design can be used and reused to create circuits of varying sizes without requiring major changes in the original code. By using parameters like generics, designs can be made scalable. Additionally, standardized hardware languages can be ported from one design environment to another without involving any major changes.
1.6 Thesis Organization

The thesis is divided into two sections. The basic design guidelines for writing VHDL code for effective synthesis and comparing the two design methodologies provided by Xilinx’s ISE Webpack and Altera’s QuartusII synthesis tools for implementing VHDL code. Chapter 2 depicts some of the useful guidelines for writing efficient and reliable VHDL code with respect to synthesis. Chapter 3 explains the term synthesis in detail by investigating the interpretation of the VHDL code by the synthesis tool. Chapter 4 compares interpretation of specific VHDL code on two different synthesis tools from two different vendors. Chapter 5 presents some thoughts on the future challenges and areas of research in the FPGA industry.
CHAPTER 2

VHDL CODING STYLES AND METHODOLOGIES

2.1 Understanding VHDL Coding Styles

The term methodology can be literally defined as an art that represents an orderly algorithmic approach in accomplishing a particular task. It ultimately depends on an individual whether, for a given task, the methodology applied is good or bad because of the difference in the point of view of the individual. A methodology appearing orderly and consistent to one person may be viewed as disorderly, inconsistent, cumbersome or uneconomical by another person. However, a good methodology involves accomplishing a task with high quality or less effort or both.

Technically, rapid prototyping based FPGA design is the act of converting an idea or a specification into a physical design using VHDL as the design entry language. Specifically, the methodology of top down design requires transforming ideas from an abstract form into a physical form that can be implemented. However, transforming a physical form to an abstract form that is accurate, concise and consistent with specified design requirement would require knowledge of various RTL coding styles and their affect on synthesis. There may be more than one method to model a particular design part but only a few would yield better performance. The essence of VHDL coding lies in
understanding which style yields the ultimate performance under the given set of specifications. The key to higher performance is to avoid writing code that needlessly creates additional work for the HDL compiler and synthesizer, which, in turn, generates designs with greater number of gates.

Basically, any coding style that gives the HDL simulator information about the design that cannot be passed onto the synthesis tool is a bad coding style. Also, any synthesis switch that provides information to the synthesis tool that is not available to the simulator is bad. The designer must be aware of these facts and should be able to differentiate a bad coding style from a good and efficient one since violation of these guidelines will produce pre-synthesis RTL simulations that will not match the post synthesis gate level simulations. To make the situation worse, these mismatches can be very hard to detect if all the possible logic combinations are not fully tested, which may prove fatal to the design during the final stages of production. The solution, in order to prevent such a catastrophe, is to understand what coding styles or synthesis switches can cause the RTL to gate level simulation mismatches and how to avoid those constructs.
2.2 Coding Style Requirements

The VHDL coding style and methodology that abide by these VHDL rules stresses the following requirements, [10, 11]:

- Code must abide by the VHDL language rules,
- Code should have a common look in order to enhance code familiarity between different models,
- Code should be easy to read and maintain by the author as well as by others,
- Code must yield expected results whether the description is behavioral or synthesizable,
- Obsolete or outdated VHDL should be avoided,
- The VHDL code should be cohesive. The common functions should be lumped in common packages, partitions or architectures,
- Synthesizable code must abide by vendors synthesis rules.

2.3 General Coding Guidelines

Some general tips that a designer should keep in mind for VHDL designing are, [10, 11]:

- Keep behavioral coding separate from structural coding. During debugging, it helps since less time is consumed,
- For declarations, instantiations and mappings use one line for each signal. It maintains clarity as well as makes the code more readable and understandable,
- Always use named association since it prevents accidental mixing of signals that may produce havoc in the design,
- Use as much commenting as possible to clarify the intent of the code,
  - Use a header comment for each entity-architecture and package-package body pair to provide a brief description of functionality for each lower block that is instantiated,
  - Use header comments for processes, functions and procedures to describe the purpose of each block of code,
  - Use individual comments internal to processes, functions and procedures to describe what a particular statement is accomplishing,
- Proper indentation for blocks of the code ensures readability and reuse of code,
- Use all lowercase names with underscores. Do not use _in and _out suffixes for port signal names since it might prove confusing, especially at hierarchical boundaries,
- Append a suffix to signals that use a clock enable and will be part of a clock-enabled path. This helps in the designers’ ability to specify multi-cycle constraints,
- Use active high signals throughout the VHDL. This makes the code easier to debug, test and also reduces complication,
- Use std_logic_arith, std_logic_signed and std_logic_unsigned packages. This provides a standard set of casting functions to maintain consistency between designers,
- Never assign to a signal in more than one process except for three state signals,
- Use alias only when it clearly promotes readability without adding complex redirection.
• For combinatorial processes, never assign to a signal and read from the same signal in the same process. This will eliminate infinite loops when performing behavioral simulations,

• For sequential processes, never assign to a signal outside of the control of the rising_edge(clk) statement. Such action will infer a combinatorial signal.

• Within a combinatorial process, all signals that are read must be in the sensitivity list. Within a sequential process; only asynchronous set/reset and clock should be in the sensitivity list. If signals are also used it will slow down the simulation.

• Always make an assignment to a variable before it is read. Otherwise, variables will infer either latches or registers to maintain their previous value,

• Use subprograms wherever possible at a local level. Do not specify the width of the inputs and outputs and use the range attributes for getting the size of the object,

2.4 Coding Style Guidelines For Synchronous Systems

Almost every present day digital system uses synchronous logic with frequencies ranging from DC to GHz. Some of the reasons for using synchronous logic include, [10, 11]:

•Eliminates speed variation problem through different paths of logic,

•Not affected by system parameters such as voltage and process,

•Simplifies interfacing between two blocks of logic by defining standardized synchronous behavior and eliminates elaborate hand shaking and token passing mechanisms,
- Portability of design, which allows migration to new and improved technology because of the deterministic behavior of the synchronous designs.

One of the main areas of concern in synchronous logic is the distribution of clocks throughout the design. For a good clock distribution system design, the clock skew must be kept to a minimum so that the setup and hold times are not violated at any one device. Methods for skew minimization include equal length traces and zero delay PLL based buffers. Also, the clocks waveform must be as clean and deterministic as possible.

State machines are one of the important constructs for the synchronous design. Any form of state machine, Mealy, Moore or mixed is a combination of combinational logic and a number of registers with decision making capability. The output of the state machine may include latch enables, tri-state enables, register enables or other control signals such as counter enables. When the state registers and inputs of Mealy or Moore transitions settle, the combinatorial gates produce glitches. The glitches are caused by varying propagation delays. These glitches are undesirable since they might open latches or clock registers and produce erroneous results. One simple solution that will guarantee glitch free outputs of the state machine is to include an output register of D_FFs.

A reset signal is used to clear the inputs of the state machine register elements. However, this reset signal is used asynchronously. If reset is asserted, everything is cleared immediately. If reset is de-asserted, the state machine will change from the reset state to some other state. However, if the reset de-asserts close to a clock edge some of the static bits will assume their new states while others might remain unchanged. Thus, an error state is obtained. The solution to prevent such an error state is to synchronize the reset input so that it will be de-asserted long before the clock edge, which will allow all
register elements to transition correctly. As a matter of fact, each and every input of the state machine must be synchronous.

Generally, all the states within a state machine are not always used. Indeed some state assignments are left unassigned. For example, a 20-state state machine using a 5-bit state register leaves 12 unused state values. If the state machine happens to enter one of the unused states it might enter into a dead-lock state that might require a hard reset in order to recover.

A single bit register that is implemented with a latch may use just 60% of the gates that a conventional flip-flop requires. Therefore, implementing a design that has a large number of configuration registers, FIFO’s or has elaborate data paths might be very tempting. However, there are major drawbacks in using latches. These drawbacks include:

- **NOISY INPUTS**: With the latch enable on, the input is propagated to the output along with the glitches. Also, the latch needs the D input to be stable for two clock periods. If this D input is changed at the same edge that closes the latch, then a race-condition results.

- **NOISY LATCH ENABLE**: Even the latch enable may show glitches, which are generally due to asynchronous decode logic. If this occurs, the whole circuit operation may fail.

The solution is to use registers instead of latches. Register design will use more space than equivalent latch based design but the register design will be robust and produce a faster design since register based design will be free from Race Conditions and any glitches on D inputs will not affect circuit operation.
Race Conditions are one of the problems that should be dealt with in synchronous design. Figure 2.1 demonstrates the Race-Condition.

The circuit is same for A and B. When A’s output changes, it might violate the hold time of B’s input if there is a clock skew between A and B’s clock. For example if B’s clock lags A’s, then the output from A might propagate through B and the desired extra clock delay will not exist. The solution for the Race Condition is to insert a delay element between A and B as depicted in Figure 2.2. The delay element ensures that there is enough time for the B flip-flop to complete its transition before the result of A’s transition reaches B.

The system should be designed while taking into consideration design testability issues. Use of a counter implies a lot of test vectors to ensure its functionality. If the counter is too long, it will require a large number of test vectors to verify its functionality. One simple solution is to break a long counter so that the design provides:
The ability to partition a counter into multiple smaller counters,

Visibility of the most significant bit of each stage. This helps in verifying that the counter as a whole works as a unit by monitoring the most significant bit transition from high to low and vice versa for every counter stage.

Write FSMs using two processes. One process will handle sequential assignments to registers and the other will handle the combinational logic. Two process FSMs are more representative of how the actual hardware works and produces more readable and predictable code. The process sensitivity list should have CLK and required SET/PRESET signals. According to the IEEE standard, this is sufficient for synchronous processes since it increases the design speed.

2.5 Design Guidelines For Synthesis

Good design for synthesis will include some or all of the following guidelines:

- Do not use delay elements since they cause synthesis and timing verification problems. If it is the only choice, then both the best and worst-case timing should be considered,

- Try to avoid using Flip-Flop’s with negative clock edges since they cause both synthesis and timing verification problems,

- Do not use buffer type ports to read output values within the code. Buffer ports can be connected to buffer ports only and a buffer port, if used, will propagate throughout the entire design. The better option is to use the mode out and add another variable or signal and assign it to the same output,
• Try to use a configuration to map entities, architectures and components. This helps in tracing the different architecture changes in a single file,

• Include all signals that are read inside the combinational process in its sensitivity list. This helps in preventing latch inferences,

• Avoid using long if-then-else statements. Use of a case statement is a better option since it prevents inferring large priority decoders,

• Like if-then-else statements, for loops also account for priority encoded logic if not carefully designed. Therefore, a designer should be cautious while using for loops for creating logic and should look for another way to write the code to implement the same functionality with the logic implemented in parallel,

• A designer should avoid slicing signals. If a signal is sliced, vector optimizations cannot be applied and a single bit slice will propagate the unoptimized vector to all affected processes. Use of a temporary variable instead of a slice can provide the same functionality without the performance penalty,

• Use the std_logic type for external ports.

These critical coding-style rules are necessary to ensure design quality and control needed to prevent design flow bottlenecks in downstream tools, [3]. They optimize implementation area and performance for field programmable gate arrays, (FPGAs). This coding-style policy will also improve the team design process by guiding engineering coding styles, which will insure that they are more portable and compliant with established design reuse practices.
CHAPTER 3

SYNTHESIS FEATURES AND INTERPRETATION

3.1 What Is Synthesis?

The automated process of converting a RTL / Dataflow model of a digital system into a gate level circuit is called synthesis. Here, an abstract specification is interpreted in terms of an equivalent optimized hardware implementation without overriding the specified constraints.

3.2 Inputs To Synthesis

In order to synthesize a circuit, following information is required about the circuit.

- A HDL code specifying the functional description of the hardware circuit to be targeted.
- A set of constraints specification in order to guide the process of synthesis. These constraints include speed, area and power consumption and will govern the choice of an optimal solution.
- The main library where the behavior of the basic components, from which the target design is to be built, is described.
The result of the synthesis process is the gate level net list of components. This netlist is further processed by the back end tools to get the desired functionality in the target hardware.

3.3 VHDL Subset For Synthesis

An efficient and necessary methodology for using VHDL in an organization is to determine all the constructs that are specific to the target application and develop the corresponding VHDL libraries. This set of VHDL constructs is referred to as a subset. Once such an environment is established, most of the designers need only to reuse these predefined constructs.

3.3.1 Supported VHDL Constructs

- **IEEE Libraries:**
  - Std_logic_1164,
  - Std_logic_textio,
  - Std_logic_signed,
  - Std_logic_unsigned,
  - Numeric_bit,
  - Numeric_std.

- **Design Units:**
  - Entity,
  - Architecture,
  - Package,
  - Package Body,
  - Configuration Declaration.

- **Sub–Programs:**
  - Functions,
  - Procedures.

- **Ports of Mode:**
  - In,
  - Out,
  - Inout,
  - Buffer.
• Object Types:
  Signals,
  Constants,
  Variables.
• Composite Types:
  Arrays,
  Records.
• User defined Enumeration Types,
• Integer and Sub-Types: Natural and Positive,
• Operators: +, -, /, *, **, mod, rem, abs, not, =, /=, <, >, <=, >=, and, or, nand,
  nor, xor, sll, srl, sla, sra, rol, ror, &,
• Sequential Statements:
  Signal and Variable Assignments,
  Function and Procedure calls,
  Wait, If, Case, Loop, For, Return and Null.
• Concurrent Statements:
  Signal Assignment,
  Process,
  Block,
  Component Instantiation,
  Subprogram Call,
  Generate.
• Generic Ports in Entities,
• Predefined attributes:
  ‘Range,
  ‘Event.
• Aggregates and others clauses.

3.3.2 Unsupported VHDL Constructs

Certain VHDL constructs are not supported by synthesis tools. Synthesis tools do not, at the present time, support constructs such as:
• Access and File Types,
• Register and Bus Signals, which are rarely used,
• Guarded Blocks, which are rarely used,
• Next and Exit loop control statement,
• Real Type objects,
• User defined resolution functions.

3.3.3 Ignored VHDL Constructs

Certain VHDL constructs such as the Assert and Report Statements as well as the After Clause, at the present time, are ignored by the synthesis tool.
3.4 Mapping Of VHDL Constructs Into Logic Networks

During Synthesis, all the available VHDL constructs in the form of VHDL statements get transformed into an equivalent logic network. Examples of the transformation of some of the statements are presented next.

3.4.1 Signals

The signals can be represented either by a wire or by a latch. An example of a design entity that utilizes signals and the mapping to a network are depicted in Figure 3.1.

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity add is
Port (A, B, C:in integer;
S:out integer);
End entity add;

Architecture behavior of add is
Begin
P:Process (A, B, C) is
Begin
If A then S <= B + C;
End if;
End Process P;
End architecture behavior;
```

Figure 3.1: Mapping of a Signal, [2]

Under the following circumstances, a latch will represent a signal:
• If a signal is driven by a process containing a “Wait Until” statement,

• If a signal is driven by a process and updated by some and not all of the alternative paths of a conditional branch as shown in the Figure 3.1.

3.4.2 Variables

Variables are used to compute and/or keep the intermediate results within a process. Those variables which keep track of information across process execution are mapped into latches/registers. Any others are mapped into wires or are optimized away.

Under the following circumstances, a variable will be mapped into a memory element:

• If a variable is read before being updated,

• If the enclosing process contains a “Wait Until” statement,

• If within a process having a sensitivity list/Wait On statement, the variable assignment occurs within a branch of a conditional statement.

3.4.3 Wait Statements

Wait statements can be in several ways. These constructs are discussed in the following sub-sections.

3.4.3.1 Wait On <Sensitivity List>

The Wait on Statement with a sensitivity list should be the last statement of the process. It lists the signal to which the logic network is sensitive. Since a logic network is sensitive to all of its inputs, it is compulsory to include all the signals read by the process in the sensitivity list of the Wait on statement.
3.4.3.2 Wait Until <Condition>

The Wait until statement acts as synchronization primitive, dictating the condition under which signals driven by the process acquire their new values. As a result, all the signals driven within a process having a “Wait Until” statement will be latched. As a rule of thumb, any signal after the “Wait Until” statement should be read. Also, the wait statement must not be embedded within any conditional or loop statement. After clauses are not allowed in Wait statements since the timing constraints implied are not generally easy to implement. Even if they are used, they are simply discarded.

3.4.4 Process Statement

The Process statement is the most general construct in the VHDL language for describing the behavior of hardware. A process usually has a sensitivity list containing all of the inputs to the block of logic. Though including all of the inputs in the process statement is not mandatory, most of the synthesis tools will issue a warning if missing inputs are detected. In addition, there will be a mismatch between pre and post synthesis simulation results as a result of the missing inputs.

3.4.5 Conditional Signal Assignment Statement

Consider the design entity ‘mux (archmux)’, which uses a 3-bit select input ‘S’ to choose between 4 inputs ‘a’, ‘b’, ‘c’ and ‘d’. The conditional signal assignment statement assigns decreasing priority to the conditions corresponding to the order they appear in the statement. The value of ‘S’ selects inputs ‘a’, ‘b’ or ‘c’ with ‘a’ having the highest priority, while input ‘d’ is selected if ‘S’ does not satisfy any of the possible
values. The code corresponding to a design entity utilizing the Conditional Signal Assignment Statement and the synthesized circuit are presented in Figure 3.2.

```vhdl
Library ieee;
Use ieee.std_logic_1164.all;

Entity mux is
Port (a, b, c, d:in std_logic;
    S:in std_logic_vector(2 downto 0);
    X:out std_logic);
end entity mux;

Architecture archmux of mux is
Begin
    X <= a when ( S = "000" ) else
         b when ( S = "101" ) else
         c when ( S = "110" ) else
         d;
End architecture archmux;
```

Figure 3.2: Mapping of a Conditional Signal Assignment Statement, [23]

The three conditions for inputs ‘a’, ‘b’ or ‘c’ are detected by three AND gates while an OR gate is used to detect the otherwise condition when none of the conditions are true. The outputs of these gates enable one of the data inputs to pass through the data selector by asserting the corresponding enable (e) input.
3.4.6 Selected Signal Assignment Statement

The selected signal assignment statement is similar to the condition signal assignment statement except that in the selected signal assignment statement, no particular choices selection has a higher priority than any of the others. The code and its synthesized circuit are presented in Figure 3.3.

```vhdl
Library ieee;
Use ieee.std_logic_1164.all;

Entity mux is
  Port (output_signal: out std_logic;
        a, b, c, d: in std_logic;
        Sel: in std_logic_vector(2 downto 0));
End entity mux;

Architecture with_select_when of mux is
Begin
  With sel select
    Output_signal <= a when "111",
                   b when "010",
                   c when "101",
                   d when "100",
                   'X' when others;
end architecture with_select_when;
```

Figure 3.3: Mapping of a Selected Signal Assignment Statement, [23]

In the code of Figure 3.3, the design entity “mux with architecture with_select_when” selects between four data inputs using a 3-bit select input. It requires
less selection logic due to the presence of the “‘X’ when others” clause. This clause is interpreted as a ‘don’t care’ condition and is exploited during the optimization phase to minimize the number of gates required by the synthesized circuit. In this circuit, none of the inputs has higher priority than any of the others. The OR-gate produced in the previous case is not produced due to the absence of the prioritized conditions.

3.4.7 If-Then-Else Statement

Consider the process with two different if-then-else statements. The code and its synthesized mapping are presented in Figure 3.4. The variable ‘v’ is assigned either to ‘a’ or ‘b’ depending on the state of input ‘c1’.

```
P : Process (c1, c2, c3, a, b) is
Variable v : bit;
Begin
    If c1 = '0' then v := a;
    else v := b;
    End if;
    If c2 = '1' and c3 = '1' then v := not v;
    End if;
    F <= v;
End process P;
```

![Figure 3.4: Mapping of an If-Then-Else Statement, [23]](image-url)
The first if-then-else statement results in the left hand data selector. The second if-then-else statement results in the second data selector, on the right hand side, where ‘v’ is inverted if ‘c2’ and ‘c3’ are both ‘1’. Thus, output ‘f’ is being driven by ‘v’ or ‘not v’.

Therefore, an if-then-else statement results in a two input data selector module.

### 3.4.8 If–Then–Elsif–Else Statement

A multi-way decision if-then-elsif-else statement tests each condition in strict order of priority. A true condition causes execution of the corresponding statements following the “then” keyword and control then passes to the statement immediately following the “end if” statement. Consider a process which selects the output ‘a’, ‘b’, ‘c’ or ‘d’ depending upon the inputs ‘c0’, ‘c1’ or ‘c2’ as logic ‘1’. The first condition c0= ‘1’ corresponds to the right hand data selector which has the highest priority. The code and its mapping are presented in Figure 3.5.

```vhdl
P: Process (c0, c1, c2, a, b, c, d) is
Begin
  If c0 = '1' then f <= a;
  elsif c1 = '1' then f <= b;
  elsif c2 = '1' then f <= c;
  else f <= d;
end if;
End Process P;
```

Figure 3.5: Mapping of an If-Then-Elsif-Else Statement, [23]

The final else part of the above statement maps to the upper input of the left hand data selector, which connects input ‘d’ to output ‘f’ only if all the control inputs are ‘0’.

Thus, if-then-elsif-else statement results in a set of cascaded data selectors.
3.4.9 If Without Else Statement

If a design requires a latch for proper functionality then there is an appropriate design unit. Signal objects in VHDL have implicit memory. The currently assigned value is retained until the signal is updated by another assignment statement. Consider a process, which shows how a data latch can be described by using an if statement without an else part. When the ‘enable’ signal transitions from ‘0’ to ‘1’, the process is triggered and Q is updated. However, if the condition ‘enable = 1’ is false, Q is not updated. Since Q is a signal, it will retain the value previously assigned to it, thus latching the value.

Figure 3.6 presents the code and the synthesized circuit appropriate to the code.

```vhdl
P: Process (enable, data) is
Begin
    If enable = '1' then
        Q <= data;
    End if;
End process P;
```

Figure 3.6: Mapping of an If without Else Statement, [23]

If the latch is undesirable in the circuit, then the best solution would be to include the else part to complete the if-statement, which will eliminate the possibility of producing a latch.
3.4.10 The Case Statement

A more concise way of implementing the multi-way decision logic is by using a Case statement. This statement executes only that condition statement where the expression between ‘case’ and ‘is’ matches one of the choices following when. The Case Statement mapping of the code to a circuit for a process describing a 4 by 1 multiplexer is presented in Figure 3.7.

```
P: Process (sel, a, b, c, d) is
   Begin
   Case sel is
   when “00” => f <= a;
   when “01” => f <= b;
   when “10” => f <= c;
   when “11” => f <= d;
   when others => f <= ‘X’;
   end case;
   end process P;
```

![Figure 3.7: Mapping of a Case Statement, [23]](image)

Even though all the four possible input combinations are included, a final ‘when others’ clause is needed since the other combinations associated with the 9-value logic were not included. The case statement must always be enclosed within a process statement. The case statement results into a straightforward parallel data selector as depicted in Figure 3.7.
3.4.11 For-Loop Statement

The For-loop statement in VHDL is used for iteration or repetition of logic. As far as logic synthesis is concerned, For-loops are acceptable provided they do not include a next or exit statement within the process. Consider a process using a for-loop statement. The loop body is executed 4 times with ‘i’ ranging from 0 to 3. Let a. represent repetitive logic and b represent iterative logic. The code and the code mapping are presented in Figure 3.8.

P: Process (a, b) is
variable v: std_logic;
Begin
v:= 0;
For i in 0 to 3 loop
f(i) <= a(i) and b(3-i);
V := v xor a(i);
end loop;
g <= v;
end process P;

Figure 3.8: Mapping of a For-Loop Statement, [23]
The first statement in the for-loop creates repetitive logic since four 4-input AND gates are produced as shown in Figure 3.8(a). During synthesis, the loop unwinds producing a circuit for each pass through the loop body. The second statement in the for-loop creates an iterative logic circuit consisting of three 2-input XOR gates as shown in Figure 3.8(b).

3.4.12 Three State Logic

In addition to the 2-logic values, ‘0’ and ‘1’, the third logic value ‘Z’ representing the high impedance state can be implemented by a synthesis tool. This is accomplished by including a package called ‘std_logic_1164’, which is contained in library IEEE.

Consider a design entity for a single bit 3-state buffer device. This particular buffer is unidirectional since it drives the output port in one direction. The value of the output is the value of the input port or ‘Z’ depending on the enable input. The code and the code mapping of the Three State Logic are presented in Figure 3.9.

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

Entity tristate is
  Port (input, enable:in std_logic;
        output:out std_logic);
end entity tristate;

Architecture Arch of tristate is
  Begin
    output <= input when enable = '1'
    else 'Z';
  end Architecture Arch;
```

Figure 3.9: Mapping of Three State Logic, [23]
3.4.13 Bi-Directional Three-State Buffer

Consider the design entity for a single bit bidirectional 3-state buffer with an XOR gated input. A port with mode INOUT is used to allow the data to flow in either direction. The Truth Table for the Bi-directional Three State Buffer is presented in Table 3.1.

<table>
<thead>
<tr>
<th>Input</th>
<th>enable</th>
<th>temp</th>
<th>Output</th>
<th>bidir_port</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 / 1</td>
<td>0</td>
<td>0 / 1</td>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>0 / 1</td>
<td>1</td>
<td>0 / 1</td>
<td>Input XOR temp</td>
<td>Input</td>
</tr>
</tbody>
</table>

The synthesized circuit along with the accompanying VHDL code is presented in Figure 3.10.

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

Entity bidir is
Port (A, enable, x: in std_logic;
y: out std_logic;
    bidir_port: inout std_logic);
end entity bidir;

Architecture tri_state of bidir is
Begin
    bidir_port <= A when enable = '1' else 'Z';
    y <= bidir_port xor x;
end entity tri_state;
```

Figure 3.10: Mapping of a Bi-directional Three-State Buffer, [23]
3.4.14 The Sequential Domain

In sequential synchronous logic, a global clock is utilized to control the execution of the statements within the process body. Signal values are mapped to the output of memory elements called Flip-Flops.

3.4.14.1 D Type Flip-Flops

Consider the design process presented in Figure 3.11. The sensitivity list may contain just the clock, preset and clear signals. The process gets triggered every time an event occurs on the clock signal. The output ‘q’ is updated only on the rising clock edge, (‘0’ to ‘1’). The corresponding D Type Flip-Flop that results from synthesizing the process is also presented in Figure 3.11.

```vhdl
P: Process (clk) is
   Begin
      if rising_edge(clk) then
         q <= data;
      end if;
   end process P;
```

Figure 3.11: D Type Flip-Flop, [23]
3.4.14.2 Synchronous Reset

The sequential logic circuits may use asynchronous reset. However, use of synchronous reset is recommended. Consider the process presented in Figure 3.12. All statements execute on the rising edge of the clock with the reset input being tested first. The synthesized circuit for the given process is also presented in Figure 3.12.

```vhdl
P: Process is
   Begin
      Wait until rising_edge(clk)
      if reset = '1' then q <= '0';
      else q <= data;
      end if;
   end process P;
```

Figure 3.12: Mapping of a Synchronous Reset, [23]

Depending on the VHDL construct in the code, the synthesis tool maps it to the corresponding hardware. Therefore, every construct should be used very discretely while keeping in mind its hardware mapping and resource utilization.
CHAPTER 4

COMPARISON OF DIFFERENT DESIGN METHODOLOGIES

4.1 Need For A Comparison

With the rapid increase in technology, digital systems are becoming more and more advanced and hence more complicated. Such complicated systems are less comprehensible to the human mind. To tackle this complexity many CAD tools are being developed and put into use. Currently one of the most important technologies in the field of digital design is the VHDL Language, whose use is increasing daily. In order to implement complex digital systems using VHDL, various VHDL CAD tools called Logic Synthesis Tools have become available in the market along with the simulation tools that transform the VHDL description and simulate its operation. VHDL Synthesis tools have become crucial for implementing digital systems.

With various VHDL Synthesis Tools available, it may turn out to be a time consuming task to select a proper one to implement a particular digital system. Some of the tools available in the market and, which are well recognized all over the world are presented in Table 4.1.
Table 4.1: Different Vendors And Their Synthesis Tools

<table>
<thead>
<tr>
<th>Synthesis Tool</th>
<th>Vendor</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISE Webpack</td>
<td>Xilinx Inc.</td>
</tr>
<tr>
<td>QuartusII</td>
<td>Altera Corporation.</td>
</tr>
<tr>
<td>Actel</td>
<td>Actel Corporation.</td>
</tr>
<tr>
<td>View Synthesis</td>
<td>ViewLogic</td>
</tr>
<tr>
<td>AutoLogic</td>
<td>Mentor Graphics.</td>
</tr>
<tr>
<td>ISPHDL</td>
<td>Lattice Semiconductor Corporation</td>
</tr>
<tr>
<td>Synergy</td>
<td>Cadence Design Systems.</td>
</tr>
<tr>
<td>LeonardoSpectrum</td>
<td>Exempler Logic Inc.</td>
</tr>
<tr>
<td>Synplify</td>
<td>Synplicity Inc.</td>
</tr>
<tr>
<td>Warp II</td>
<td>Cypress Semiconductor Corporation</td>
</tr>
</tbody>
</table>

All of the tools, presented in Table 4.1, perform VHDL synthesis and implement the digital system onto a chip. However, each software system has its own advantages and disadvantages. Some are quite user friendly but take a lot of time to implement a small system. Some are pretty fast but have a poor Graphic User Interface. Some won’t handle highly complex systems. In general each available system will have properties that are both good and bad with respect to a given design. With so many things to be taken care off, within the constraints of the design desired specifications, it becomes a challenge to select the right synthesis tool to accomplish a particular task.

The two design environments that were selected for comparison, in this research, were ISE Webpack from Xilinx Inc. and QuartusII from the Altera Corporation. The selection was largely based on the popularity of these two tools in the field of digital design that targets implementation in Programmable Logic Chips. Both companies have enjoyed a strong earnings growth for almost the last half decade. In addition to this, extra synthesizers are available such as FPGA Express with Altera and Leonardo Spectrum with Xilinx. Finally, both of these synthesizing software systems are available on the internet, as educational software, that is free of cost to the user.
4.1.1 Direct Competitor Comparison

Table 4.2 presents a direct competitor comparison among the various chip-making companies that are competing today in the FPGA market. Table 4.2 shows clearly that Altera and Xilinx form the backbone of the programmable chip industry.

Table 4.2: Direct Competitor Comparison, [14]

<table>
<thead>
<tr>
<th></th>
<th>XLNX</th>
<th>ACTL</th>
<th>ALTR</th>
<th>LSCC</th>
<th>Industry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Market Cap</td>
<td>9.41B</td>
<td>369.90M</td>
<td>7.30B</td>
<td>478.52M</td>
<td>272.43M</td>
</tr>
<tr>
<td>Employees</td>
<td>2,770</td>
<td>543</td>
<td>1,995</td>
<td>1,048</td>
<td>488</td>
</tr>
<tr>
<td>Rev. Growth (ttm)</td>
<td>20.90%</td>
<td>11.60%</td>
<td>16.20%</td>
<td>-8.50%</td>
<td>9.10%</td>
</tr>
<tr>
<td>Revenue (ttm)</td>
<td>1.51B</td>
<td>157.72M</td>
<td>938.75M</td>
<td>215.80M</td>
<td>142.10M</td>
</tr>
<tr>
<td>Gross Margin (ttm)</td>
<td>63.60%</td>
<td>61.01%</td>
<td>68.78%</td>
<td>56.35%</td>
<td>38.68%</td>
</tr>
<tr>
<td>EBITDA (ttm)</td>
<td>396.34M</td>
<td>10.04M</td>
<td>321.56M</td>
<td>-19.64M</td>
<td>2.76M</td>
</tr>
<tr>
<td>Oper. Margins (ttm)</td>
<td>25.78%</td>
<td>4.68%</td>
<td>29.90%</td>
<td>-42.94%</td>
<td>5.75%</td>
</tr>
<tr>
<td>Net Income (ttm)</td>
<td>352.00M</td>
<td>8.54M</td>
<td>222.97M</td>
<td>-86.42M</td>
<td>N/A</td>
</tr>
<tr>
<td>EPS (ttm)</td>
<td>0.983</td>
<td>0.317</td>
<td>0.577</td>
<td>-0.768</td>
<td>N/A</td>
</tr>
<tr>
<td>PE (ttm)</td>
<td>27.57</td>
<td>45.33</td>
<td>33.78</td>
<td>N/A</td>
<td>33.42</td>
</tr>
<tr>
<td>PEG (ttm)</td>
<td>1.06</td>
<td>1.45</td>
<td>1.38</td>
<td>4.27</td>
<td>0.97</td>
</tr>
<tr>
<td>PS (ttm)</td>
<td>6.60</td>
<td>2.41</td>
<td>8.13</td>
<td>2.35</td>
<td>2.09</td>
</tr>
</tbody>
</table>

XLNX: represents Xilinx Inc.
ACTL: represents Actel Corp.
ALTR: represents Altera Corp.
LSCC: represents Lattice Semiconductor Corp.
Industry: represents Semiconductors.

4.2 Comparison Between Altera’s QuartusII And Xilinx’s ISE Webpack

The comparison was performed on the basis of 3 aspects:

I. Advantages and disadvantages of the two design environments,

II. Portability of VHDL code between the two design environments,

III. Optimization of the VHDL synthesis to a target technology.
4.2.1 Advantages And Disadvantages Of The Two Design Environments

4.2.1.1 Advantages Of Altera’s QuartusII

Altera’s QuartusII design software system offers a number of properties that enhance and shorten the design cycle. The more important of these properties are:

- The technology leadership of QuartusII design software combined with a broad portfolio of design ready Intellectual Property, (IP), Cores provides designers unparalleled levels of performance, ease of use, reduced time to market and total cost of ownership, [26],
- Altera’s QuartusII software offers advanced technology such as Design Space Explorer, (DSE), and Physical Synthesis, which insures that the designer gets the best possible performance from their designs,
- Armed with industry leading tools and features such as SOPC builder, Chip Editor with change Manager, Tcl Scripting and automatic Timing Analysis, the QuartusII tool accelerates design completion and time-to-market,
- The Graphic User Interface for the QuartusII software is an integrated environment where all the tools are available within the click of a button,
- Line numbers and the color combination for the syntax make the code more readable,
- Almost all of the integrated tools are supplied, at no additional charge, with the QuartusII Tool.
4.2.1.2 Disadvantages Of Altera’s QuartusII

Altera’s QuartusII design software system also provides certain problems for the digital system designer. The more frustrating of these problems are:

- The Project Navigator window does not have the facility for tabs, which makes it difficult to use multiple files,
- Error messages are vague and difficult to understand,
- A long start-up time before compilation,
- Help documentation is presented in an unorderly manner,
- The QuartusII software tool is difficult to work with especially for file manipulation between projects,
- During compilation or simulation, the QuartusII software may suspend and not proceed to the next module if a menu or a modal dialog box is open at the time the current module finishes its execution,
- QuartusII allows implementation of a design only into Altera chips. Hence, it is not useful for future design implementation into a newer technology,
- There exists no explicitly available power analyzer tool.

4.2.1.3 Advantages Of Xilinx’s ISE Webpack

Xilinx’s ISE Webpack design software system also offers a number of properties that enhance and shorten the design cycle. The more important of these properties are:

- The main user interface screen is a congregation of various panes/tabs such as sources pane, processes pane, Transcript window pane and the log tab.
Therefore, with only the main screen open, different tasks can be handled at the click of a button without cluttering the screen. This provides an environment that allows the designer to rapidly create or debug a design and achieve timing closure,

- The integrated environment enhances the learning process and accelerates time to productivity,
- A snapshot is a read-only copy of a current project including all of the projects files and process properties. This provides a way to save multiple project revisions. Additionally, snapshots can be retrieved for examination or editing at a later time,
- ISE Webpack’s integrated environment provides language templates in order to use and modify the design code,
- Floor Planning is included and well supported in the Xilinx ISE Webpack,
- Other utilities such as a simulator, power analyzer and a timing analyzer are included within the integrated environment,
- Simply double clicking on the error messages, the line where the error occurred is highlighted. Also line numbers are provided in the coding space to reach the error line fast and without any confusion,
- The error messages are clear and understandable,
- A special tool for Power Analysis available.
4.2.1.4 Disadvantages Of Xilinx’s ISE Webpack

Xilinx’s ISE Webpack design software system also provides certain problems for the digital system designer. The more frustrating of these problems are:

- Since the Xilinx ISE Webpack involves all the tools in one environment, it is more complicated for the novice to learn.
- Online help involves long search times.
- The Xilinx ISE Webpack system only allows implementation of a design in Xilinx chips. Hence, it is not useful for future design implementation in a newer technology.

4.2.2 Portability Of The Synthesis Code

The term portability refers to the ability of code to run on different platforms. Portability of code demands a technology independent coding style. This means that a designer can start a particular design using VHDL and can select the implementation vehicle later when the details about design utilization are available. This saves time, money and the possibility of having to modify the whole code to conform to a new environment at a later time if the decision is made to migrate to a new solution. Standard VHDL code will port easily to any design environment. However, different VHDL vendors use their own subset of VHDL. Therefore, VHDL code will not necessarily be portable between all the design environments.

In order to compare the portability factor between the two selected design environments, compilation and synthesis of some combinational and sequential VHDL code was attempted on Altera’s QuartusII and Xilinx’s ISE Webpack tools. Most of the
code compiled successfully in both design environments indicating the flexibility provided by the portability of the code. However, as mentioned earlier, the code portability between different vendor tools depends upon the VHDL subset chosen by the vendor and hence absolute compatibility is not assured. To illustrate this, a Screensaver Design Project was selected, [25]. This project had been implemented by Xilinx on the XESS40 board. An attempt was made to re-implement the circuit on Altera’s UP1 board. However, because of a frequency mismatch the screensaver display was not visible on the VGA monitor. The VHDL code for the top entity of the Screensaver project, “a0top.vhd”, is presented in the following paragraphs. The component’s Netlist was intentionally not created at the beginning. The VHDL code follows:

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity A0TOP is
  port(clock,rsetz: in std_logic;
       Vsyn, Hsyn:out std_logic;
       OEz,WEz,CEz:out std_logic;
       R,G,:out std_logic_vector(1 downto 0);
       AdBUS: out std_logic_vector(14 downto 0);
       D aBUS: inout std_logic_vector(7 downto 0));
end entity A0TOP;

architecture TOP_arch of A0TOP is

  component Top_fuego is
    port(clock, rsetz, Vsync, Enable:in std_logic;
         H:in std_logic_vector(2 downto 0);
         DataBUSInFuego:in std_logic_vector(7 downto 0);
         AddrBUSBR:out std_logic_vector(6 downto 0);
         AddrBUSFU: outstd_logic_vector(12 downto 0);
         DataBUSOutFuego:out std_logic_vector(7 downto 0);
         EBrasas, CalcFuego, Lectfuego, Escrfuego:out std_logic);
  end component top_fuego;
```
component SecuenciadorGeneral is
  port(resetzt, clk:in std_logic;
       c:out std_logic_vector(1 downto 0);
       Ei:out std_logic_vector(3 downto 0));
end component SecuenciadorGeneral;

component DriverVGA is
  port(clk, resetz:in std_logic;
       Vsync, Hsync:out std_logic;
       C:in std_logic_vector(1 downto 0);
       BusDatos:in std_logic_vector(7 downto 0);
       R, G, B:out std_logic_vector(1 downto 0);
       H:out std_logic_vector(3 downto 0);
       BusDirecciones:out std_logic_vector(14 downto 0));
end component DriverVGA;

component FSMDelay is
  port(Clk, Resetz, Vsync:in std_logic;
       Flag:out std_logic);
end component FSMDelay;

signal CalcFuego, CalcBrasas, uno, cero, Hsyncro, Vsyncro:std_logic;
signal contr, Eifuego, EnableBrasas, CalcFuegoAux:std_logic;
signal LectfuegoAux, EscrfuegoAux, WFlag, Bras:std_logic;
signal Caux, Caux2:std_logic_vector(1 downto 0);
signal Raux, Gaux, Baux:std_logic_vector(1 downto 0);
signal Eiaux, Haux:std_logic_vector(3 downto 0);
signal AddrBUSauxVGA:std_logic_vector(14 downto 0);
signal AddrBUSauxVGA2:std_logic_vector(14 downto 0);
signal DataBUSauxINVGA:std_logic_vector(7 downto 0);
signal AddrBUSauxBR:std_logic_vector(6 downto 0);
signal AddrBrasas:std_logic_vector(14 downto 0);
signal AddrBUSauxFU:std_logic_vector(12 downto 0);
signal AddrFuego:std_logic_vector(14 downto 0);
signal DataBUSauxINFuego:std_logic_vector(7 downto 0);
signal dabus_sal, Brasa:std_logic_vector(7 downto 0);
signal DataBUSauxOUTFuego:std_logic_vector(7 downto 0);

begin
  uno <= '1';
  cero <= '0';

  Secuenciador:SecuenciadorGeneral
    port map(resetzt, clock, Caux2, Eiaux);
  Video:DriverVGA
    port map(clock, resetz, Caux, Vsync, Hsync, Raux, Gaux, Baux, Haux, AddrBUSauxVGA, DataBUSauxINVGA);
Efecto_Fuego:Top_fuego

**port map** (clock, rsetz, Vsyncro, Haux(3 downto 1), Eifuego, EnableBrasas, AddrBUSauxBR, AddrBUSauxFU, DataBUSauxINFuego, DataBUSauxOUTFuego, CalcFuegoAux, LectFuegoAux, EscrFuegoAux);

Retraso:FsmDelay

**port map** (clock, rsetz, Vsyncro, WFlag);

Caux <= Caux2;
R <= Raux;
G <= Gaux;
B <= Baux;
Eifuego <= Eiaux(0);
Vsyn <= Vsyncro;
Hsyn <= Hsynrono;
AddrBUSauxVGA2(14 **downto** 6) <= AddrBUSauxVGA(14 **downto** 6);
AddrBUSauxVGA2(5 **downto** 0) <= AddrBUSauxVGA(5 **downto** 0)+"110000";
AddrBrasas(6 **downto** 0) <= AddrBUSauxBR;
AddrBrasas(14 **downto** 7) <= "00111111";
AddrFuego(12 **downto** 0) <= AddrBUSauxFU;
AddrFuego(14 **downto** 13) <= "00";
Bras <= (AddrBUSauxBR(6) xor uno);
Bras(0) <= Bras;
Bras(1) <= Bras;
Bras(2) <= Bras;
Bras(3) <= Bras;
Bras(4) <= Bras;
Bras(5) <= Bras;
Bras(6) <= Bras;
Bras(7) <= Bras;

buses:*process*(Haux, uno, cero, AddrBUSauxVGA2, DaBUS, Eifuego, CalcFuegoAux, EnableBrasas, EscrFuegoAux, WFlag, Brasa, AddrBrasas, LectFuegoAux, AddrFuego, DataBUSauxOUTFuego) is

begin

DaBUS_sal <= (others=>'0');
Contr <= '0';
if (Haux(1)='0' **and** Haux(0)='0') then
  CEz <= cero;
  OEz <= cero;
  WEz <= uno;
  AdBUS <= AddrBUSauxVGA2;
  DataBUSauxINVGA <= DaBUS;
  DataBUSauxINFuego <= (others=>'Z');
elsf (Eifuego='0') then
  CEz <= uno;
OEz <= uno;
WEz <= uno;
AdBUS <= (others=>'Z');
DataBUSauxINVGA <= (others=>'Z');
DataBUSauxINFuego <= (others=>'Z');

elsif (CalcFuegoAux = '0' and EnableBrasas = '0') then
    CEz <= uno;
    OEz <= uno;
    WEz <= uno;
    AdBUS <= (others=>'Z');
    DataBUSauxINVGA <= (others=>'Z');
    DataBUSauxINFuego <= (others=>'Z');

elsif (CalcFuegoAux='0' and Haux(1)='0' and Haux(0)='1') then
    CEz <= cero;
    OEz <= uno;
    WEz <= uno;
    AdBUS <= AddrBrasas;
    DataBUSauxINVGA <= (others=>'Z');
    DataBUSauxINFuego <= (others=>'Z');
    Contr <= '0';
    DaBUS_sal <= Brasa;

elsif (CalcFuegoAux='0' and Haux(0)='0') then
    CEz <= cero;
    OEz <= uno;
    WEz <= Wflag;
    AdBUS <= AddrBrasas;
    DataBUSauxINVGA <= (others=>'Z');
    DataBUSauxINFuego <= (others=>'Z');
    Contr <= '1';
    DaBUS_sal <= Brasa;

elsif (CalcFuegoAux='0') then
    CEz <= cero;
    OEz <= uno;
    WEz <= uno;
    AdBUS <= AddrBrasas;
    DataBUSauxINVGA <= (others=>'Z');
    DataBUSauxINFuego <= (others=>'Z');
    Contr <= '1';
    DaBUS_sal <= Brasa;

elsif (Haux(1)='0' and Haux(0)='1') then
    CEz <= cero;
    OEz <= uno;
    WEz <= uno;
    AdBUS <= AddrFuego;
    DataBUSauxINVGA <= (others=>'Z');
    DataBUSauxINFuego <= (others=>'Z');
Contr <= '0';
DaBUS_sal <= DataBUSauxOUTFuego;

elsif (EscrFuegoAux='1' and Haux(0)=0') then
  CEz <= cero;
  O Ez <= uno;
  WEz <= Wflag;
  AdBUS <= AddrFuego;
  DataBUSauxINVGA <= (others=>'Z');
  DataBUSauxINFuego <= (others=>'Z');
  Contr <= '1';
  DaBUS_sal <= DataBUSauxOUTFuego;

elsif (EscrFuegoAux='1') then
  CEz <= cero;
  O Ez <= uno;
  WEz <= uno;
  AdBUS <= AddrFuego;
  DataBUSauxINVGA <= (others=>'Z');
  DataBUSauxINFuego <= (others=>'Z');
  Contr <= '1';
  DaBUS_sal <= DataBUSauxOUTFuego;

elsif (LectFuegoAux='1' and Haux(0)=0') then
  CEz <= cero;
  O Ez <= cero;
  WEz <= uno;
  AdBUS <= AddrFuego;
  DataBUSauxINVGA <= (others=>'Z');
  DataBUSauxINFuego <= DaBUS;

elsif (LectFuegoAux='1') then
  CEz <= cero;
  O Ez <= uno;
  WEz <= uno;
  AdBUS <= AddrFuego;
  DataBUSauxINVGA <= (others=>'Z');
  DataBUSauxINFuego <= DaBUS;

else
  CEz <= uno;
  O Ez <= uno;
  WEz <= uno;
  AdBUS <= (others=>'Z');
  DataBUSauxINVGA <= (others=>'Z');
  DataBUSauxINFuego <= (others=>'Z');
end if;
end process buses;

zeta: process (contr,dabus_sal) is
begin
if (contr='1') then
    dabus <= dabus_sal;
else
    dabus <= (others=>'Z');
end if;
end process zeta;
end architecture Top_arch;

When an attempt was made to compile this design in the Altera’s QuartusII tool, the process came to an abrupt end with an error message stating “component entity top_fuego.vhd not found”. When the same code was ported and compiled in the Xilinx’s ISE Web pack tool, a similar error was expected. However, no such error occurred. Instead, a warning message was produced, which stated “Generating a Black Box for component top_fuego” and also for all other similar components.

The reason for such a difference in behavior between the two tools is that FPGA designs may involve a foreign Netlist, which is required during the translation phase and not during synthesis. Hence, if the Xilinx tool identifies a condition where there is no foreign Netlist, it automatically uses a black box and sends a message to the user. Therefore, the design doesn't fail. With the warning message produced during this process, it is possible to confirm that:

- Black boxes have been recognized by the Xilinx tool and the appropriate hooks have been implemented,
- HDL sub modules have not been found and must be added to the project before the design is re-synthesized.

A synthesis tool can work in one of two mutually exclusive ways when encountering a hole in the code. It can either automatically allow the hole and warn the
user about the missing code or it can post an error and require attributes for a black box. Xilinx chose the first option and Altera chose the second.

Another example will further illustrate the difference in portability between the two tools. The VHDL design entity code for the component “mux4a1generico.vhd” of the screensaver project’s top entity a0top.vhd code was compiled first in Altera’s QuartusII tool where it compiled successfully without producing any error. The actual code that successfully compiled in Altera’s QuartusII tool is given in the following paragraphs.

QuartusII VHDL code for component “mux4a1generico.vhd”

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity mux4a1Generico is
  generic (size:integer := 1);
  port (Selection:in std_logic_vector(1 downto 0);
        Entrada1 :in std_logic_vector(size-1 downto 0);
        Entrada2 :in std_logic_vector(size -1 downto 0);
        Entrada3 :in std_logic_vector(size -1 downto 0);
        Entrada4 :in std_logic_vector(size -1 downto 0);
        Salida :out std_logic_vector(size -1 downto 0));
end entity mux4a1Generico;

architecture mux4a1Generico_arch of mux4a1Generico is
begin
  p1:process(Selection, Entrada1, Entrada2, Entrada3, Entrada4) is
  begin
    case "Selection" is
    when "00" => Salida <= Entrada1;
    when "01" => Salida <= Entrada2;
    when "10" => Salida <= Entrada3;
    when "11" => Salida <= Entrada4;
    when others => Salida <= Entrada1;
    end case;
  end process p1;
end architecture mux4a1Generico_arch;
```
When an attempt to compile the design entity code for component “mux4a1generico” was made in Xilinx’s ISE Webpack tool, an error message was produced stating “Selection is not a valid expression as a 'case' selector.” After a modification was made in the case syntax, the code ran successfully. The VHDL design entity code for the component “mux4a1generico.vhd” that successfully compiled in the ISE Webpack tool is presented in the following paragraphs.

ISE Webpack VHDL code for component “mux4a1generico.vhd”

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity mux4a1generico is
  generic(size:integer := 1);
  port(Selection:in std_logic_vector(1 downto 0);
       Entrada1 :in std_logic_vector(size-1 downto 0);
       Entrada2 :in std_logic_vector(size -1 downto 0);
       Entrada3 :in std_logic_vector(size -1 downto 0);
       Entrada4 :in std_logic_vector(size -1 downto 0);
       Salida :out std_logic_vector(size -1 downto 0));
end entity mux4a1generico;

architecture mux4a1generico_arch of mux4a1generico is
begin
  p1:process(Selection, Entrada1, Entrada2, Entrada3, Entrada4) is
  begin
    case Selection is
    when "00" => Salida <= Entrada1;
    when "01" => Salida <= Entrada2;
    when "10" => Salida <= Entrada3;
    when "11" => Salida <= Entrada4;
    when others => Salida <= Entrada1;
  end case;
end process p1;
end architecture mux4a1generico_arch;
```
4.2.3 Optimization Of VHDL Synthesis For A Target Technology

Synthesis compilers convert HDL code into a new format that can be implemented in silicon. The resulting figure of merit is based on factors such as the amount of silicon required to implement the design, which is measured in terms of the total number of gates and the design speed or design frequency, [27].

Prior to starting the implementation process, the designer must consider the FPGA families on which the implementation is going to take place. The design family selected to implement the Screensaver Design Circuit was the EPF10K20, Flex Family, from Altera and the XC2S100, Spartan II Family, from Xilinx.

The interconnect resources and the array of logic cells, which are also called logic blocks, form the basic building blocks of an FPGA. Interconnects are used to connect the different logic blocks while logic blocks provides the facility to implement the intended functionality. With the technology continually advancing, the number of logic blocks in an FPGA device is reaching millions, which allows the implementation of very complex functions into a device.

4.2.3.1 Logic Elements Of The FLEX10K Family

The configuration of the logic element in the FLEX10K family is presented in Figure 4.1. Each of the logic elements consists of a four input Look-Up Table, (LUT), and a function generator that can quickly compute any function of four variables. In addition, each logic block contains a Flip-Flop with a synchronous clock enable, a carry chain and a cascade chain. With respect to combinatorial functions, the Flip-Flop is bypassed and the LUT’s output drives the outputs of the logic block. The carry chain is
used mainly in high-speed counters and adders while the cascade chain is used to implement wide input functions with a minimum delay. Carry and cascade chains are generally used only for critical high-speed applications.

4.2.3.2 Configurable Logic Block Of The SpartanII Family

In the SpartanII family, a logic block is called a Configurable Logic Block, (CLB). A CLB consists of 4 Logic Cells, (LC). The LCs are arranged in two similar slices of two LCs each. A single slice of a CLB is presented in Figure 4.2. Each LC consists of a 4-input function generator, carry logic and a storage element. A function generator is implemented as a 4-input LUT. The function generator can act as a 16x1 bit synchronous RAM. Similarly, two function generators within a slice can act as a 16x2 bit synchronous RAM, a 32x1 bit synchronous RAM or a 16x1 bit dual port synchronous RAM. In addition, a SpartanII CLB contains Input-Output Blocks (IOBs) to provide the interface between the package pins and the internal logic.
Clock DLLs are provided for clock distribution delay compensation and clock domain control. Furthermore, there are Flip-Flops for implementing sequential logic.

### 4.2.3.3 Implementation Of The Design

The Screensaver design was implemented simultaneously using Altera’s QuartusII and Xilinx’s ISE Webpack synthesis tool. The design had already been implemented by Xilinx on the XESS40 Board using Xilinx’s ISE Webpack tool, [24]. As an extension of the Xilinx work, an attempt was made to re-implement the screensaver design on the Altera UP1 board using Altera’s QuartusII synthesis tool. Simultaneously, the design was synthesized again using Xilinx’s ISE Webpack in order to make a comparative study between the two synthesis tools.
4.2.3.3.1 Using Altera’s QuartusII

The Screensaver design circuit was implemented first in an EPF10K20 using Altera’s QuartusII tool. The design was again compiled and then synthesized using default timing parameters for the EPF10K20 device. The corresponding results are presented in the Table 4.3. The results depict the circuit performance in terms of frequency Fmax. Design optimization was performed first with respect to speed and then with respect to area. Also shown is the number of logic units, which included 4-input LUT’s as well as registers utilized while optimizing the design for speed and area.

<table>
<thead>
<tr>
<th>Optimization Goal</th>
<th>Performance, Fmax in MHz</th>
<th>No. of 4-input LUT’s( % utilization)</th>
<th>No. of Registers i.e. F/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>47.85</td>
<td>413 / 1152 (35 %)</td>
<td>206 / 2400</td>
</tr>
<tr>
<td>Area</td>
<td>46.08</td>
<td>386 / 1152 (33%)</td>
<td>206 / 2400</td>
</tr>
</tbody>
</table>

Table 4.3 shows that under speed optimization, the circuit ran at a maximum frequency of 47.85MHz. The number of LUT’s utilized was 413, which accounted for 35% of the total LUT’s available while 206 registers were used. Similarly for area optimization, the circuit ran at a maximum frequency of 46.08MHz. The number of LUT’s utilized was 386, which accounted for 33% of the total LUT’s while 206 registers were used.

Thus, between the speed and area optimization above, there existed a difference of only 2% in logic element utilization with register usage the same for both. Also, there was not much difference in circuit performance was there was a change of just 1 MHz. This data shows that in order to optimize speed, area is sacrificed and vice-versa.
4.2.3.3.2 Using Xilinx’s ISE Webpack

The Screensaver circuit was implemented in an XC2S100 device of the SpartanII family using Xilinx’s ISE Webpack 6.2i tool. The design was compiled and synthesized using default timing parameters for the XC2S100 device. The corresponding results are presented in Table 4.4.

<table>
<thead>
<tr>
<th>Optimization Goal</th>
<th>Performance, Fmax in MHz</th>
<th>No. of 4-input LUT’s( % utilization)</th>
<th>No. of Registers i.e. F/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>113.5</td>
<td>328 / 2400 (13 %)</td>
<td>212 / 2400</td>
</tr>
<tr>
<td>Area</td>
<td>99.8</td>
<td>326 / 2400 (13%)</td>
<td>208 / 2400</td>
</tr>
</tbody>
</table>

Table 4.4 shows that under speed optimization, the circuit ran at a maximum frequency of 113.5MHz. The number of LUT’s utilized was 328, which accounted for 13% of the total LUT’s while 212 registers were used. Similarly for area optimization, the circuit ran at a maximum frequency of 99.8MHz. The number of LUT’s utilized was 326, which accounted for 13% of the total LUT’s while 208 registers were used.

Between speed and area optimizations there was an almost negligible difference in logic element utilization with the register usage being almost identical. However, there was a noticeable difference in circuit performance since there was a change of approximately 15 MHz in Fmax. This data shows that in order to optimize speed, area is sacrificed and vice-versa.

4.2.3.4 Speed Vs Area Comparison Between Individual Tools

With respect to the individual synthesis tool comparison in terms of speed and area, the results depended upon the context of the design, which determined the selection of the particular parameter for the optimization goal. If speed was the prime factor of the
design with area as less significant factor, then speed was the parameter optimized. The same was true if area was deemed the more significant parameter. In most of the designs, a proper setting between the speed and area optimization was considered appropriate.

4.2.3.5 Speed Vs Area Comparison Between QuartusII And ISE Webpack

The data presented in Tables 4.3 and 4.4 clearly show that for speed optimization, the number of LUT’s and registers utilized by the circuit using the Altera QuartusII tool and Xilinx’s ISE Webpack tool were almost identical. However, circuit performance in terms of Frequency, Fmax, using Xilinx’s ISE Webpack was 113.5MHz, which was superior to the 47.85 MHz performance obtained using Altera’s QuartusII. With respect to area optimization, the number of LUT’s and registers utilized by the circuit using Altera’s QuartusII tool and Xilinx’s ISE Webpack tool were almost the same. However, circuit performance in terms of Frequency, Fmax, using Xilinx’s ISE Webpack was 99.8MHz, which was again superior to the 46.08 MHz performance produced by Altera’s QuartusII. Therefore, the data force the conclusion that the circuit performance using Xilinx’s ISE Webpack was better than that using Altera’s QuartusII. However, according to Altera, [28], a comparison is not fair if a design is compared with the default set of parameters set for each of the two tools. The reason specified is that different synthesis tools have different default timing analysis and optimization behaviors. The QuartusII tool, by default, performs a complete and thorough analysis of all permutations of the path and assumes the worst possible case when reporting timing analysis, [28]. The Xilinx ISE tool, by contrast, omits certain structures during timing analysis, which reduces the total net delay and improves performance. However, even after applying all
the modifications mentioned in [28], for making a fair comparison, there was no major difference in the results. The Xilinx ISE tool still produced a circuit that yielded improved performance when compared to the Altera QuartusII tool under same set of constraints. Another factor, which favored Xilinx’s ISE Webpack, was that application of timing constraints further improved the Xilinx circuit performance. However, there was no such improvement in performance with the Altera QuartusII circuit. The data obtained from the comparisons with and without constraints is presented in Table 4.5.

<table>
<thead>
<tr>
<th>ISE Webpack Synthesis</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx ISE Synthesis without Constraints</td>
<td>Fmax = 99.8MHz (Period: 10.015 ns)</td>
</tr>
<tr>
<td>Xilinx ISE Synthesis with Constraint, Fmax=8ns</td>
<td>Fmax = 126.8MHz (Period: 7.886 ns)</td>
</tr>
</tbody>
</table>

The comparison between the two synthesis tools with respect to the Graphic User interface and VHDL portability issue showed little difference between the tools. Xilinx’s ISE Webpack was better than Altera’s QuartusII with respect to circuit optimization for synthesis. The results of Xilinx’s ISE Webpack were far better than those for Altera’s QuartusII.
CHAPTER 5

FUTURE RESEARCH

5.1 The Future Of FPGA Design Verification

With the increase in digital circuit complexity, FPGA users have to deal with more than one language in their designs such as the original design sources written in VHDL while the IP cores in Verilog. To speed up test bench simulation, patches written in C/C++ are used frequently, [12]. In addition, embedded systems are entering the FPGA domain. The latest trend for the FPGA is the inclusion of specialized hardware in the form of hard cores, which reduce cost and development time. A new concept called hybrid device is under development that involves embedding FPGA logic inside an ASIC, which will enable the majority of the design to be optimized and frozen while smaller sections of the design can be changed in the field, [12]. With such features available in the FPGA domain, the new development tools for FPGA’s will need to be up-to-date with state-of-the-art technology in order to take advantage of the new features during design optimization. With processor incorporated FPGAs, [11], the tools need to take software into account in order to optimize at a higher level of abstraction. Future tools will need to have the capability to synthesize and optimize software in a manner similar to how they now synthesize and optimize hardware. These intelligent tools will work with libraries of pre-tested hardware objects and software functions and leave low
level “C” and VHDL design necessary only for unique, specialized sections of hardware or software, [11].

The latest release of the QuartusII tool from Altera is easy to use and efficient software for designing with CPLDs, FPGAs and structured ASICs. QuartusII is an upgraded version of MaxPlusII that delivers better performance, possess more features and provides access to exciting new CPLD and FPGA device families, [26]. Since it is more reliable and user friendly most MaxPlusII users are making the move to the QuartusII tool.

According to Mr. David Dye from Xilinx.com, Xilinx will be releasing its new version of ISE Webpack, which will be labeled version 6.3i, at the end of September 2004. Very little information is available about the newer version but it will have support for the new architecture family Virtex-IV. All the individual tools, especially PACE and Architecture Wizard, will have incremental improvements.

Synthesis vendors are currently working with FPGA suppliers in the hope of advancing the tools to a level where they automatically exploit all of a device’s architectural features, [29]. As the size of FPGA’s grow, some more powerful and user friendly tools will appear in the market that will help the designers meet their challenges and give a new dimension to the FPGA industry.
REFERENCES


5. Robert W. Hix Jr. and Dr. Roger L. Haggard, “Comparative Design Methodologies For FPGA Based Computer Arithmetic”, Tennessee Technological University, Cookeville, TN 38505


   http://finance.yahoo.com/q/co?s=XLNX

   reuse.com/articles/article6404.html

    Standard”,
    http://www.synthworks.com/papers/VHDL_RTL_Synthesis_Standard_HDLCON
    _2002.pdf


18. Ray Weiss, “Probing The Limits Of Logic Synthesis”, http://www.reed-
    electronics.com/ednmag/archives/1994/031794/06dfcov.htm

19. “Analysis Of VHDL Models”, http://mikro.e-technik.uni-ulm.de/vhdl/anl-
    engl.vhd/html/node12.html

20. “VHDL Coding Style Guidelines”, Education Services, Xilinx Global Services,
    from Xilinx at Xilinx.com

    http://www.rulabinsky.com/cavd/text/chap04-1.html

    http://www.eedesign.com/story/OEG20031007S0053

23. “VHDL for Synthesis”, AMI, Advanced Electronic Design Automation, Unit 8,
    http://216.239.41.104/search?q=cache:XYozK1C6LWUJ:www.ami.ac.uk/course
    ware/adveda/ch8/adveda08notes.html+adveda+unit+V&hl=en

    http://www.esi2.us.es/%7Eaguirre/LabMic_eng.html


26. “Performing Equivalent Timing Analysis Between The Altera QuartusII And
    Xilinx ISE Software” Altera, White Paper, WP-TMANAL, version 1.0, August 2

27. “Careful HDL Coding Maximizes Performance In LUT Based FPGA’s”,
    http://www.elecedesign.com/Articles/ArticleID/7531/7531.html
APPENDICES
### APPENDIX-A; PORTABILITY RESULTS

<table>
<thead>
<tr>
<th>Appendix-A1</th>
<th>Compiling a0top.vhd Using Altera’s QuartusII</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tool</td>
<td>Altera’s QuartusII</td>
</tr>
<tr>
<td>Design</td>
<td>Screensaver Design Project</td>
</tr>
<tr>
<td>Comment on Result</td>
<td>Error was thrown as an undefined component entity.</td>
</tr>
</tbody>
</table>
APPENDIX-A; PORTABILITY RESULTS (Continued)

Appendix-A2 : Compiling a0top.vhd Using Xilinx’s ISE Webpack
Tool : Xilinx’s ISE Webpack
Design : Screensaver Design Project
Comment on Result : No Error thrown. A lack box was generated automatically for each undefined component entity.
APPENDIX-A; PORTABILITY RESULTS (Continued)

Appendix-A3: Compiling mux4a1generic Using Altera’s QuartusII
Tool: Altera’s QuartusII
Design: Screensaver Design Project
Comment on Result: No Error thrown.
Appendix-A4 : Compiling mux4a1generico Using Xilinx’s ISE Webpack
Tool : Xilinx’s ISE Webpack
Design : Screensaver Design Project
Comment on Result : An invalid case selector syntax error was thrown.
APPENDIX-B; SYNTHESIS RESULTS

Appendix-B1: Altera’s QuartusII Fitter Report For Speed Optimization
Tool: Altera’s QuartusII
Design: Screensaver Design Project
Optimization Goal: Speed
Comment on Result: Fitter Report (a0top.fit.rpt)
APPENDIX-B; SYNTHESIS RESULTS (Continued)

Appendix-B2 : Altera’s QuartusII Timing Report For Speed Optimization
Tool : Altera’s QuartusII
Design : Screensaver Design Project
Optimization Goal : Speed
Comment on Result : Timing Analysis Report (a0top.tan.rpt)
Appendix-B3 : Altera’s QuartusII Fitter Report For Area Optimization
Tool : Altera’s QuartusII
Design : Screensaver Design Project
Optimization Goal : Area
Comment on Result : Fitter Report (a0top.fit.rpt)
APPENDIX-B; SYNTHESIS RESULTS (Continued)

Appendix-B4: Altera’s QuartusII Timing Report For Area Optimization
Tool: Altera’s QuartusII
Design: Screensaver Design Project
Optimization Goal: Area
Comment on Result: Timing Analysis Report (a0top.tan.rpt)
APPENDIX-B; SYNTHESIS RESULTS (Continued)

Appendix-B5 : Xilinx’s ISE Webpack Map Report For Speed Optimization
Tool : Xilinx’s ISE Webpack
Design : Screensaver Design Project
Optimization Goal : Speed
Comment on Result : Map Report (a0top.mrp)
APPENDIX-B; SYNTHESIS RESULTS (Continued)

Appendix-B6 : Xilinx’s ISE Webpack PAR Report For Speed Optimization
Tool : Xilinx’s ISE Webpack
Design : Screensaver Design Project
Optimization Goal : Speed
Comment on Result : Text Based Post Place & Route Static Timing Report (a0top.tw1)
Appendix-B7 : Xilinx’s ISE Webpack Map Report For Area Optimization
Tool : Xilinx’s ISE Webpack
Design : Screensaver Design Project
Optimization Goal : Area
Comment on Result : Map Report (a0top.mrp)
APPENDIX-B; SYNTHESIS RESULTS (Continued)

Appendix-B8: Xilinx’s ISE Webpack PAR Report For Area Optimization
Tool: Xilinx’s ISE Webpack
Design: Screensaver Design Project
Optimization Goal: Area
Comment on Result: Text Based Post Place & Route Static Timing Report (a0top.tw1)