A VHDL-BASED DIGITAL SLOT MACHINE IMPLEMENTATION USING A COMPLEX PROGRAMMABLE LOGIC DEVICE

by

Lucas C. Pascute

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science of Engineering

in the

Electrical Engineering

Program

YOUNGSTOWN STATE UNIVERSITY

December, 2002

A VHDL-BASED DIGITAL SLOT MACHINE IMPLEMENTATION USING A COMPLEX PROGRAMMABLE LOGIC DEVICE

Lucas C. Pascute

Signature:

[illegible] 12-3-02

Lucas C. Pascute, Student Date

Approvals:

[illegible] 12/3/02

Dr. Faramarz Mossayebi, Thesis Advisor Date

[illegible] 12-3-02

Dr. Robert H. Foulkes, Committee Member Date

[illegible] 12/03/02

Dr. Salvatore Pansino, Committee Member Date

[illegible] 12/12/02

Dr. Peter J. Kavinsky, Dean of Graduate Studies Date

ABSTRACT

The intent of this project is to provide an educational resource from which future students can learn the basics of programmable logic and the design process involved. More specifically, the area of interest involves very large scale integration (VLSI) design and the advantages associated with it such as reduced chip count and development time. The methodology used within is to first implement a design; using small and medium scale integration (SSI/MSI) packages in order to have a baseline for comparison. The design is then translated for use with the very high speed integrated circuit hardware description language (VHDL) and implemented onto a complex programmable logic device (CPLD). A discussion of this implementation process as well as VHDL lessons is provided to serve as a tutorial for the interested reader. This thesis concludes with a summary of the project results and ideas for future research topics.

ACKNOWLEDGEMENTS

I would first and foremost like to thank Dr. Mossayebi for his patience and time that he gave to me throughout this entire process. I greatly appreciate the fact that he often went beyond what was required to help me complete this research in the time frame I desired. Also, I must graciously thank Dr. Pansino and Dr. Foulkes for serving on my thesis committee on such short notice. The time spent reviewing my thesis during the busy holiday season is well appreciated. The other professors that I have had in the department over my academic career (Dr. Jalali, Dr. Munro, and Dr. Rost) must also be recognized because they have taught me many of the principles I needed to be able to undertake a project of this magnitude. In addition, I have to thank all of my friends and family who kept me as sane and motivated as possible during some of the tough times experienced while doing this research. Last but not least, a huge thank you goes out to Carolyn for helping me with all of the odds and ends that I have needed during my five and a half years here, she always was willing to help even when it seemed I was in there more than once a day!

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The topic of digital design is the fastest changing and most diverse branch of electrical engineering. This diversity stems from the fact that some aspect of almost all topics of electrical engineering can relate to digital design. On the most basic level, electrical components such as resistors, transistors, etc. can be interconnected in a manner to produce various logic functions. This basic level also requires knowledge of electromagnetic field theory since these fundamentals are inherent to any type of circuit; analog or digital. Once logic gates (functions) are built through electrical components, they can be used as the building blocks for digital design. Next, the gates can be connected to form registers, which in turn can form chips. The highest level of digital systems is a collection of many chips and is called the process memory switch level. This level refers to hardware devices such as computer central processing units (CPUs). This hierarchy demonstrates how the topics can vary from basic electromagnetic field theory to computer engineering as the levels grow to increasingly more macro levels. For this thesis, the areas of interest focus mainly on the gate, register, and chip levels since one of the project's intents is to demonstrate various digital logic components and design processes.

The gate, register, and chip levels consist of integrated circuits (ICs) of various sizes and were originally measured by the number of gates present on a single chip. The smallest category of these ICs is referred to as small-scale integration (SSI). The traditional definition of SSI is an integrated circuit that contains between one and twenty gates. An example of SSI would be any of the 7400 series single function logic gates or flip-flops. These chips are typically contained within a 14-pin, dual-inline-pin (DIP) package.

The next size of IC is called medium-scale integration (MSI). An example of this size of integrated circuit would be a multiplexer, counter, or decoder. For an IC to be considered MSI, it needs to contain anywhere from twenty to two hundred gates. The SSI and MSI ICs are used as building blocks to combine and form LSI and VLSI integrated circuits.

Large scale integration (LSI) refers to digital components such as programmable logic devices (PLDs) and encompasses ICs with between two hundred and two hundred thousand gates. Originally, LSI was the largest category used to describe an integrated circuit. However, as the number of transistors on a single chip continues to double every eighteen months, a new category has been formed.

This newly titled category is very large scale integration (VLSI). VLSI can be applied to any IC that contains at least one million transistors. This level of integration is currently the most researched and fastest growing and includes field-programmable logic arrays (FPGAs) and complex programmable logic devices (CPLDs).

The main intent of this project is to investigate VLSI and demonstrate the ability of these devices to reduce chip count and design time. A second goal is to present the information in such a manner that an interested student could learn enough information to build his or her own design without having to delve through many additional resources. In order to accomplish these objectives, two main sections are presented within that describe and demonstrate the functionality of FPGAs and CPLDs.

The first section encompasses Chapters II, III, and IV. Chapter II describes the evolution of logic devices into the types currently being used today. Also, this chapter contains information about the various architectures as well as how software packages are used in conjunction with hardware devices to create and optimize designs. Chapter III provides specifications about the hardware and software that were used during this project. These components were graciously provided by the Altera Corporation as part of their university program design laboratory kit. An explanation of the details of this program can be found in [1]. Chapter IV introduces the VHSIC (Very High Speed Integrated Circuits) hardware description language (VHDL) that can create circuits for implementation onto CPLDs or FPGAs. The development history of this language is provided as well as a series of six informational sheets to aide future students with learning this intricate programming language.

The second section of this thesis describes the design of a digital slot machine through various stages until final implementation onto a CPLD. Chapter V discusses the origin of the design idea, many of the thought processes involved, and the final design parameters. Chapter VI presents a visual representation of this design using Microsoft Excel. Industry often uses models in order to communicate the objectives of a project to all team members. This model was developed in the spirit of this idea so that the reader can best follow and visualize the desired outcomes. Chapter VII describes a representation of the digital slot machine design using SSI and MSI parts. This chapter is included to give the reader a comparison of the design using previous technology with respect to VLSI. Chapter VIII details the process of creating and implementing the design onto the university program 1 (UP1) education board using the VHDL language with the MAX+PLUS® II software. The information is presented in such a manner that the reader could follow the details and create and implement a design of his or her choosing with this package. A conclusion containing a discussion of the results as well as ideas for future research is presented in Chapter IX.

CHAPTER II

PROGRAMMABLE LOGIC BASICS AND DEVICES

2.1 Introduction to Programmable Logic

The topic of digital design began with the use of small unchangeable IC logic packages to create logic circuits. Engineers would design an appropriate interconnection of these packages in order to achieve the desired function. The particular ICs being referred to are similar to the 7400-series TTL logic chips that can presently be found in every undergraduate digital laboratory. This method allowed simple digital circuits to be designed quickly and with low cost. However, as the circuits and logic grew larger, the designs also grew increasingly difficult and the time to build and troubleshoot them grew increasingly long. For this reason (and the advent of the personal computer), devices such as application-specific integrated circuits (ASICs) and programmable logic devices (PLDs) were created.

The first device credited with being the father of ASICs is called the Micromosaic and was introduced by Fairchild Semiconductor in 1967. This device allowed the user to program a desired logic function through use of a computer program. The computer program would connect the previously un-connected transistors that were present on the device in order to achieve the designed function.

After the creation of the Micromosaic, the traditional ASIC (as it is referred to today) began to take form. An ASIC is a device that is designed by the customer to produce a specific function. This new product allowed circuits that previously required several IC packages to be built on a single chip, thus making a significant reduction in the amount of power used and space consumed. However, the process of troubleshooting ASICs is difficult and the development costs can be great. This is due to the fact that the device is actually created by an outside manufacturer, even though it is designed by the customer. These disadvantages led to the development of the programmable logic device (PLD) in the mid 1970's.

The PLDs provided the same advantages as the ASIC but allowed the end-user to configure them instead of a separate manufacturer. This feature also allowed greater information security since the design would not need to be passed to another company for manufacturing purposes. For these reasons (and others), the "boom" of programmable logic began.

Programmable logic refers to an IC device that is configurable by the user after it is manufactured and is able to implement digital logic functions. There are three common programmable architecture categories presently used that are pertinent to this discussion. These architectures include Simple Programmable Logic Devices (SPLDs), Complex Programmable Logic Devices (CPLDs), and Field Programmable Gate Arrays (FPGAs).

2.2 Simple Programmable Logic Devices (SPLDs)

Simple Programmable Logic Devices include such devices as the Programmable Logic Device (PLD), Programmable Array Logic device (PAL), Programmable Logic Array device (PLA), and the Generic Array Logic device (GAL). The SPLDs are just as the name suggests; the smallest and cheapest architecture available for programmable logic. A single device of this architecture is capable of replacing a handful of the 7400-series TTL logic packages.

PLDs have wide varying characteristics including: the electronic technology implementation (bipolar, CMOS, etc.), implementation of the fusible links, erasure capability, and the ability (or lack thereof) to reprogram the device. The main deterrent to replacing 7400-series built circuits with PLDs in the past was related to slowness in propagation times. However, as PLD technology has matured, they have become as fast as the quickest TTL chips on the market. From all the sub-categories of SPLDs, the term "Programmable Logic Device" is used as a generic term that encompasses the others. The most popular and basic forms of PLDs are PAL devices, PLA devices, and GAL devices. It is beneficial to discuss each of these briefly so that the reader can get a better understanding of terms, such as macrocells, for future topics.

PLAs were the first widely-used programmable logic devices. The development of this device grew out of research done by IBM and Texas Instruments during the late 1960s and early 1970s and was first introduced by Signetics in 1975. This device allows the user (unlike with the ASICs) to program the connections to be made and consists of a two-level AND-OR array (sum of products) of terms. The basic structure of a four input, four output PLA is given as Figure 2.1. Through observation of this figure, many of the basic characteristics of a PLA can be observed. First of all, each input and its complement are available for use as an input to any of the AND gates. Also, each product term in the array can be connected to any of the OR gates present on the chip. The possible connections, which are represented by the dots in Figure 2.1, can be implemented in various ways including fuses or memory cells.

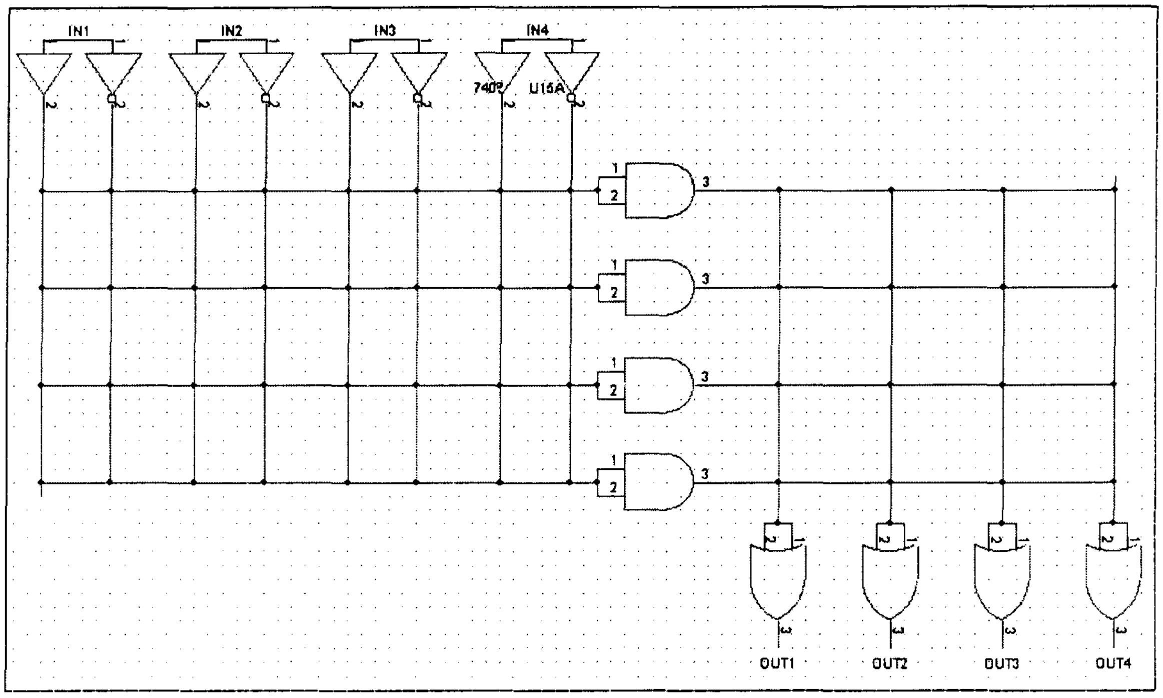


Figure 2.1 – The basic architecture of a PLA device

Using the basic PLA device architecture as a model, Monolithics Memories Inc. (MMI) produced the first PAL device in 1976. The introduction of this device helped PLDs gain extensive acceptance throughout the digital design community and were the first PLD to use versatile bi-directional input/output pins. The PAL device is similar to a PLA device as it contains an AND/OR array, however, the difference is that the OR array is not variable but instead is fixed to specific product terms. The architecture of a four input, two output PAL device is given as Figure 2.2. This figure clearly shows how each product term is not available to every OR gate as was the case for the PLA device. The logical question to ask is how is this technology even beneficial without the previous flexibility? The answer is that a PAL device is cheaper and faster than its counterpart and can implement functions close to the same size. This is why the PAL device is the most commonly used form of the PLD in the market today.

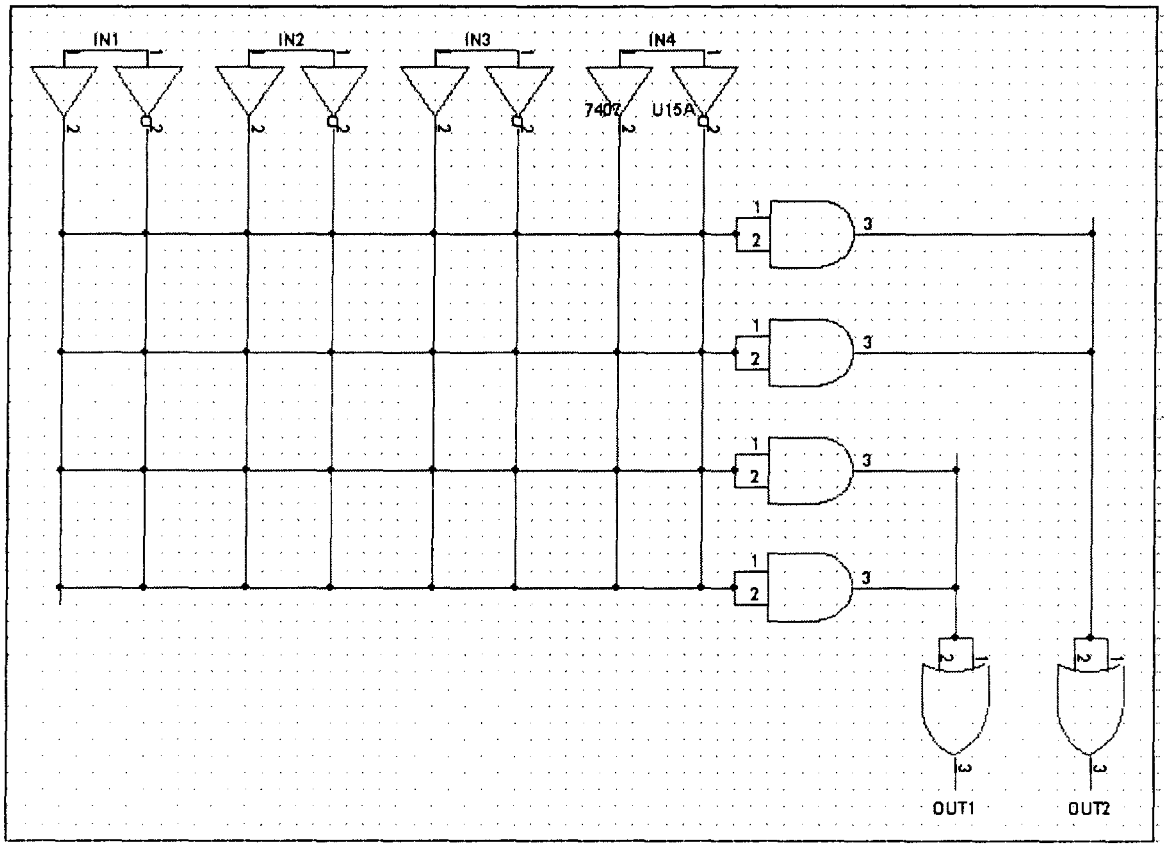


Figure 2.2 – The basic architecture of a PAL device

From the PAL architecture, a newer PLD was created by Lattice Semiconductor and was called a Generic Array Logic device (GAL). The GAL device is virtually the same as a PAL device because it was produced in such a way that it could emulate any combinational or sequential PAL device through use of AND/OR arrays and flip-flops. It even contains the same fixed product term structure as in the PAL device. Also, GAL devices were built so that they could be electronically erased and are thus re-programmable, making it valuable for prototyping and trial runs. There are many GAL device architectures that have been introduced and are on the market today; [2] and [4] give a thorough discussion of them and many examples of GAL devices.

2.3 Complex Programmable Logic Devices (CPLDs)

As logic circuits grew larger, the size of PLDs also continued to grow. The problem with this phenomenon is that both the physical and the AND/OR array size on the PLD began to be inefficient. These problems were also in addition to an increase in capacitive effects and a decrease in speed. To remedy these difficulties, the complex programmable logic device (CPLD) was developed. The basic CPLD architecture consists of three main parts: logic blocks, an interconnect structure, and input/output (I/O) blocks. A CPLD is very flexible since it allows the individual logic blocks and interconnect structure to be programmed. A typical configuration of a CPLD is depicted in Figure 2.3. Since the three main parts of the CPLD are also what distinguishes different variations of these logic devices, they will be investigated further in depth individually below.

The architecture of the CPLD shown in Figure 2.3 contains four logic blocks similar to PLDs but these devices can contain as few as two or up to as many as sixty-four or more of these blocks on a single chip. Each one of these logic blocks is then broken down further into sections called macrocells. A macrocell typically consists of a

2-level AND/OR array, a flip-flop, a multiplexer, and various other small logic circuitry. These macrocells allow for wide fan-in and typically between four to sixteen product terms. Depending on the particular architecture being examined, a single logic block generally contains up to sixteen interconnected macrocells. One useful feature on some architectures is the ability of the macrocells to share product terms. This is particularly valuable for architectures that contain only four product terms per macrocell since it allows for great flexibility. One disadvantage of this feature is that the propagation time is slightly slower and thus must be considered (as any variable should) when choosing the best architecture for the designer's specific need.

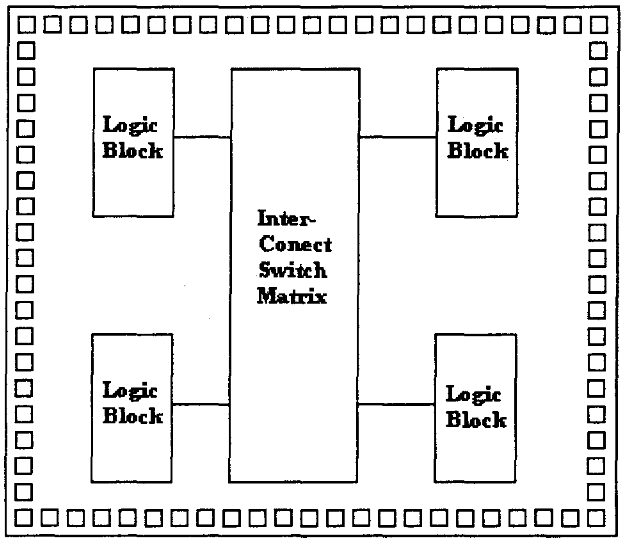


Figure 2.3 – The basic architecture of a CPLD.

The second part of the architecture which can be seen in Figure 2.3 is the interconnect switch matrix. The function of this switch matrix is to connect the logic blocks within the CPLD just as the macrocells are connected inside of them. There are two basic categories into which switch matrices fall into: partially populated and fully populated. A fully populated switch matrix can connect all the logic blocks present on the CPLD in any and all possible combinations whereas a partially populated one lacks the capability of providing some of these combinations.

Since there are more connections in a fully populated switch matrix, designs are more often easier to route than in a partially populated one. This routing process is done through use of a software package and is thus considered programmable. The ease of routing is important since the design is often constrained by a fixed pinout that must be used. It obviously makes more sense to be able to keep this fixed pinout then to redesign the peripheral circuitry around the CPLD to be compatible with a new layout. Also, partially populated switch matrices have problems with delays since they are not as predictable as their counterparts are. It almost seems as if there is no reason to ever use a partially populated switch matrix from the previous comments but the main variable in business to always remember is cost; partially populated matrices are cheaper. So in order to find the most appropriate device for a design, all variables such as speed, area, and cost should be considered.

The last component of a CPLD is the input/output blocks. These I/O blocks are used to receive the input signals from the outside or to send output signals. As for the output signals, there are often many conditions to which they can be set. For example, they typically have the ability to be set to active 'high' or 'low', a disconnect or open state, or connected to the global enables that are present on the CPLD. Also, as stated in [2], I/O block architectures now contain various analog controls in addition to the digital ones. Some of these controls include slew-rate control to manage the speed at which output signals vary, a pull-up resistor (and sometimes a pull-down resistor) to prevent situations in which a float condition is undesirable, and a user-programmable ground to best handle high changing currents on the device.

Most modern CPLDs are manufactured using electrically eraseable programmable read-only memory (EEPROM) technology. With this technology, the CPLD is non-volatile (saves it memory contents even while power is removed) and re-programmable. Since the memory is not lost through the removal of the power source, the logical assumption would be that the CPLD could only be programmed once and thus be unchangeable. This is not a valid assumption, however, as the name of the device states, it can be erased electrically.

2.4 Field Programmable Gate Arrays (FPGAs)

A desire for performance improvement to that of the CPLDs led to the development of field-programmable gate arrays (FPGAs) by the Xilinx Corporation in 1984. Instead of using a few PLD-like logic blocks with a central and fairly simple interconnect switch matrix, FPGAs use many small programmable logic blocks with a complex interconnect structure. A general architecture of FPGAs is shown as Figure 2.4 on Page 15. All FPGAs contain the same basic parts as the CPLD (logic blocks, interconnect, and I/O blocks) but each is substantially different, so each structure will be examined individually.

The logic blocks separate FPGAs into two separate categories: coarse-grained and fine-grained. A coarse-grained architecture is the most common and the best performing type used and consists of larger logic blocks as well as a few look-up tables and flip-flops. The fine-grained architecture has a larger number of logic blocks but they are smaller in size. These logic blocks typically contain a single flop-flop and a small AND/OR array. Depending on the architecture, a FPGA can contain anywhere from tens to tens of thousands of logic blocks on a single chip with even more flip-flops available. The key goal for FPGA logic blocks is getting as much capability as possible through use of the least smallest sized logic blocks as possible.

The decision of which particular architecture should be used depends mainly on the user's design. Coarse-grained FPGAs are good for applications that have a few fairly independent large logic structures. The reason for this is that the large space available in these logic blocks tends to be quite wasted when used for small logic functions. These small logic functions are best suited for fine-grained FPGAs. The main problem with fine-grained architectures is that they require a significantly larger number of interconnect paths. This increase in interconnection circuitry can lead to a considerable amount of delay and a decrease in the amount of available space. For these reasons, it is important to examine the design chosen for implementation and choose the most appropriate architecture in order to best optimize one's design.

The next part of the FPGA architecture is the programmable interconnect structure. As evident in Figure 2.4, there is no central device that controls the way in which the logic blocks are connected, it is instead done through an intricate scheme. This scheme involves connecting each of the logic blocks that are present on the FPGA with a set of wires containing programmable connections. It is important that the interconnect scheme is well thought out since the large number of wires present can result in long delays. This is especially true for fine-grained architectures that require more interconnections and thus have more potential for delays. In order to minimize this delay, computer-aided design packages use a variety of optimization tools to best "route" the design, where routing is described as the process of interconnecting the resources in a manner that meets project specifications. It is beyond the scope of this project to describe the interconnect structure or optimization methods to great detail due to the extreme variance between the many available architectures from each particular vendor. The interested reader should consult references [2] and [12] for additional information regarding these topics.

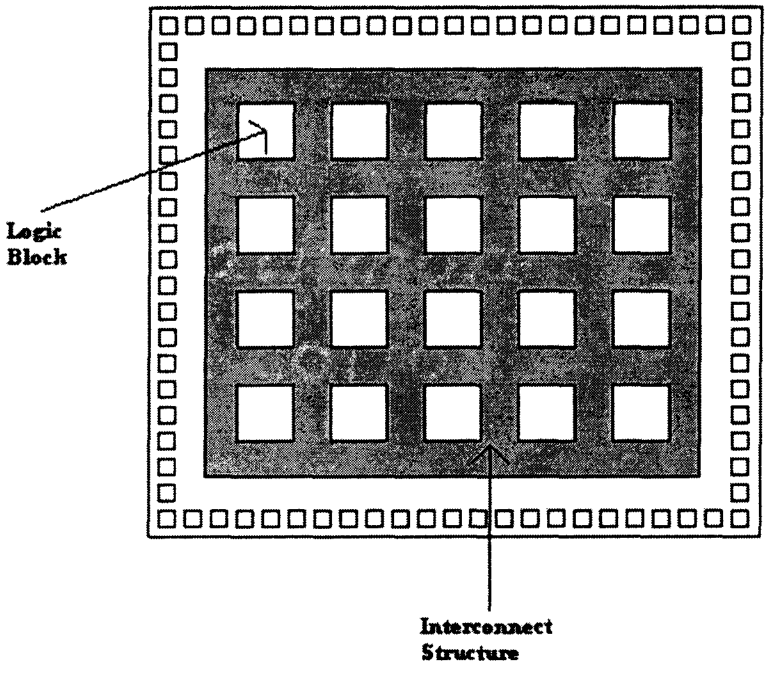


Figure 2.4 – The basic architecture of a FPGA.

The last part of the FPGA architecture is the input/output blocks. These I/O blocks are the components most similar in nature to their counterpart on a CPLD. They can be defined as not only inputs or outputs but also as three-state or bi-directional. In addition to the same analog components availabie on the CPLD such as pull-up resistors, pull-down resistors, and slew rate control, some FPGAs allow the user to provide edge-triggered flip-flops or latches to the I/O blocks. Reference [2] provides a discussion of the XC4000 FPGA (available from the Xilinx Corporation) and notes that an additional feature it has is "a delay element to guarantee that the input will have a zero hold-time requirement with respect to the external system clock".

Most of the FPGAs built today are done so using a technology called static random access memory (SRAM). SRAM technology's main advantage is its re-programmability and can be found in a similar fashion in microprocessors. Some disadvantages inherent to SRAM are that it is volatile (which means that the memory contents are lost when power is removed) and that it typically requires high power during operation. The way that SRAM works is through use of an external configuration memory source, which stores the integral program information. This includes the direction definitions of the I/O blocks, the functionality of the logic blocks, and the manner these blocks are connected together. This information is then "downloaded" or programmed onto the FPGA for design implementation. A thorough discussion of SRAM technology can be found in [2].

2.5 Computer-Aided Design

The computer-driven environment present in the world today has integrated itself into the field of digital design and has become a very important tool in it. Computer-Aided Design (CAD), also called Computer-Aided Engineering (CAE), allows the designer to create high-quality logic circuits through use of a wide variety of software tools. Typical software packages sold today contain various methods for the entry of the design, simulation tools, timing analyses, and synthesizing capabilities once the creator is confident of the correctness of the design. The discussion to follow gives a look into these main components of CAD.

In order to create a design with the objective of physical implementation using CAD, it must first be entered into the software package in some sort of fashion. Most packages today allow a great deal of flexibility to the designer when it comes to the manner of entry. The first type of these methods to be discussed and perhaps the most basic and common is schematic entry. This method allows the schematics of logic circuits to be drawn onto an editor on the computer by placing various logic symbols. More advanced software packages include a large catalog of devices available for use on the schematic as well as various troubleshooting capabilities. Schematic entry is most useful for small circuits since placing a large number of symbols and making sure they are connected properly can become cumbersome. This complication, along with the need to work out the exact design using basic logic symbols as well as documenting the design ultimately led to the development of Hardware Description Languages (HDLs).

These languages (such as ABEL, VHDL, and Verilog) allow the designer to model and design various sizes of functions, chips, and digital systems. This is done through use of programming techniques similar to those of high-level languages (Visual Basic, C++) to model the desired behavior instead of the actual combinational or sequential logic circuit required. The editors used with these HDLs are very similar to those of the other programming languages since they often provide certain tools to aide in debugging the code for successful compilation.

In addition to these entry methods, there are various other techniques that are available in software packages. Some of these methods include: state diagrams to describe the behavior of a desired state machine, truth tables describing the corresponding outputs for assorted inputs, and waveform editors to describe the timing behavior of inputs, outputs, and the relationships between the two.

Once the design has been entered in the desired manner, it can then be compiled and simulated. Once the compilation process is successful, the program will create data that can be used to verify the functionality of the circuit. Some of the tools available to the designer are various test benches and timing analyzers. A test bench uses items called "test vectors" in order to check the behavior of the circuit. These vectors can range from a single source of inputs created by the designer to a complete set of possible input combinations generated by the program itself. The timing analyzer then works together with the test bench in order to describe the time dimension and functionality of the design. Time is a very important variable to consider in the design of logic circuits since races and glitches are very difficult errors to predict and are common complications in initial design attempts. Using the combination of these available tools, the designer is able to perfect the design to his or her content without ever physically building it.

The last stage of the design development, called logic synthesis or programming, is the process of transferring the compiled and tested design onto an actual physical circuit. This procedure is often complicated by the limitations that arise on both the software package and on the hardware device being used. These constraints vary from the total number of gates and space available on the hardware device to the amount of computing power available to the software. Because of these constrictions, modern compilers contain optimization capabilities so that the physical circuit is not only functional but is programmed in a manner so that variables such as the cost and amount of circuitry needed are minimized.

A common feature available in modern compilers is a list of possible architectures for use as the targeted hardware device. Each of the architectures that are then listed contains algorithms specifically written for the optimization of designs for that device. The major logic manufacturers taken this path to encourage the users of their software to also purchase their hardware.

CHAPTER III

ALTERA'S PROGRAMMABLE LOGIC COMPONENTS

3.1 Introduction

The aim of this chapter is to provide a more specific extension to the information presented in the previous chapter. While Chapter 2 discussed the general structure of the hardware and software used in programmable logic, Chapter 3 converses about the package that was used to build the design presented within this work. This is one of the standard packages developed by the Altera Corporation to help aide in teaching digital logic design. This package, which is called the university program design laboratory package, contains a version of the company's MAX+PLUS® II software, a UP1 education board containing a EPF10K20 FPGA device and a EPM7128S CPLD, and a ByteBlasterMV^{TM} parallel port download cable. This package is available to any accredited university that actively instructs students in digital logic design. These materials provide all the necessary tools to create and implement basic digital logic designs. The capabilities of this package's hardware and software will be summarized in the following sections, which heavily rely on the user guides available from the Altera website [16].

3.2 The Hardware

The main piece of hardware in the university package is the UP1 education board. This board contains many features including two VLSI devices. In addition to these components, there is a 25.175 MHz crystal oscillator that can be used as a global clock input. Also, there four push-buttons and sixteen switches that are connected to pull-up resistor to provide input values. To view the outputs of a design, sixteen LEDs illuminated by active-low logic, and two dual-digit, seven segment displays are provided. More detailed information about each of these features as well as others seen on the board depicted in Figure 3.1 (taken from [33]) can be found in reference [14].

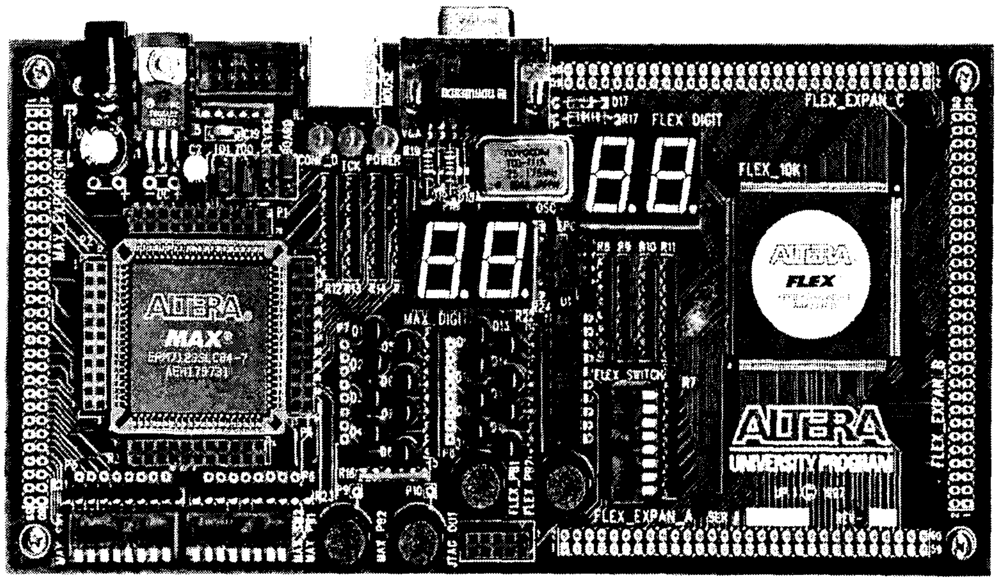


Figure 3.1 – The Altera UP1 education board

The FPGA included in our particular package is the EPF10K20 device and is part of the Altera Corporation's FLEX 10K device family. This particular device is contained within a 240-pin power quad flat pack package and is based on reconfigurable SRAM technology. It contains 1,152 logic elements and six embedded array blocks containing 2K bits of memory. A logic element is Altera's term for a small bit of logic circuitry similar to a macrocell discussed in Chapter 2. Each PLA-like block (called Logic Array Blocks by Altera) contains eight logic elements and thus one hundred forty four logic array blocks on the EPF10K20 device. The embedded array blocks can be used as RAM or ROM for the device and support synchronous and asynchronous operation. These They can also be used to implement common functions such as microcontrollers, multipliers, and state machines.

The other device on the UP1 educational board is the EPM7128S CPLD, which is part of the Altera Corporation's MAX 7000 device family. As opposed to the FPGA device, which used SRAM technology, this particular device uses EEPROM elements instead. It is packaged within an eighty four pin plastic J-lead chip carrier and contains one hundred twenty eight macrocells. These macrocells contain the normal features described in Chapter 2 as well as a "configurable register with (an) independently-programmable clock" as stated in the user's guide [14]. For the interested reader, the Altera website [16] provides a great deal of information about not only these devices, but also the architectures of the other packages that they offer as well.

3.3 The Software

Even though the MAX+PLUS® II university development software was included in the package, the version used for the design of this project was the commercial version This was due to the fact that the commercial version was already installed and contains additional resources unavailable in the student version. MAX+PLUS® II offers a very flexible design environment that supports almost all of the Altera programmable logic device families and will run on most modern operating systems. As stated in [15], the design process for MAX+PLUS® II consists of four phases: design entry, design compilation, design verification, and device programming.

The MAX+PLUS® II software provides three main design entry editors (graphic, text, and waveform) as well as a symbol editor to complement the others. Perhaps the most basic and easiest way to enter a design is through what the company calls graphic design entry. This method involves placing logic symbols (of SSI/MSI devices) and wiring them up in the desired manner with appropriate labels. The software provides over 300 typical symbol functions as well as the ability to create user-defined symbols in the symbol editor. Also, this flexible software allows schematics from other companies' software (such as Cadence's OrCAD) to be imported into the schematic screen.

Another popular entry method is text design entry. MAX+PLUS® II supplies a text editor in which the user can input VHDL, Verilog HDL as well as AHDL (Altera's Hardware Description Language) constructs or codes. Even though the software can read in coding that was written within any common text editor, the particular one available within the software contains features such as syntax coloring to easily observe keywords and basic templates for all the supported languages.

Also, the software provides a waveform editor that can be used to generate waveforms for input vectors as well as view the simulation output results after compilation. This design entry method is often popular when using state machines. Another nice feature of this software is that the these above-mentioned methods can be combined in a "mix or match" fashion to allow the greatest amount of flexibility to the user.

The next phase of the design process is the design processing stage. After the design is entered by one of the methods described above, it is then processed through the MAX+PLUS® II compiler. The compiler takes in these various files, checks them for errors, synthesizes the logic, and creates output files that can be used for programming, simulation, and timing analysis. Also, the complier optimizes the efficiency of the project so that minimal resources are required. The flexibility provided in the software allows synthesis settings to be adjusted, timing requirements entered, and unused pins and logic blocks specified.

The complier also consists of a feature called the message processor. This feature checks the files being compiled for errors, such as connection or syntax ones, and displays appropriate error or warning messages. Also, it provides the ability to automatically open the design file that is the source of the error or warning by a simple double-click. These features are invaluable to the digital logic designer in search of an elusive troublesome error.

Once the design project has been compiled, the next step of the process is design verification. The MAX+PLUS® II software includes two main features for verification: a simulator and timing analyzer. The simulator is used to test the logic integrity of the design as well as the internal timing. The input for the simulator is typically produced using a set of waveforms, which are generated in the waveform editor. Once the design is simulated, the results can be viewed by opening the waveform editor and adding a trace for the desired output. Also, there are options available that allow the design to monitored for glitches, oscillations, and other potential problems. The main reason for simulating a design is to ensure that it is working as expected before spending the time and money to actually program the device. Additional features available in the MAX+PLUS® II simulator include the ability to specify the expected logic levels, define the time interval of what constitutes a glitch, and insert breakpoints.

The purpose of the timing analyzer is to determine propagation times of the internal signals and all of the critical paths. This software feature provides the user with three possible types of analyses: delay matrix, setup/hold matrix, and the registered performance display. The delay matrix calculates all possible combinations of paths between the source and destination nodes and displays the shortest and longest of these paths. The setup/hold matrix displays the minimum required setup and hold times for devices such as latches and flip-flops. The registered performance display allows the user to define variables within the analysis and displays the results of a registered performance analysis. Once an analysis has terminated, the software provides a few more useful tools to the user. One of these tools is the ability to choose any of the source nodes and destination nodes and view the corresponding delay pattern. Also, the software provides a message processor, which lists all possible paths for any specified node.

The last step of the design process is the programming phase. In order to program the hardware device, it is first connected to the computer through the ByteBlasterMV^{TM} parallel port download cable, which is included in the university package. The programmer module in the software then uses certain files that were generated during the compilation stage to program the hardware. Also, the MAX+PLUS® II programmer allows the user to configure, verify, examine, blank-check, and functionality test the hardware in addition to programming it. Lastly, the software provides (sometimes) helpful error messages that are generated if any problems occur while programming the device to aide the user in troubleshooting.

CHAPTER IV

AN OVERVIEW OF VHDL

4.1 History of VHDL

VHDL is an abbreviation for the VHSIC (Very High Speed Integrated Circuits) Hardware Description Language. The development of VHDL was initiated in 1980, when the Department of Defense introduced what was called the VHSIC program. It should not be a surprising fact that this language was started by the government since many new technologies such as the internet were created in this manner. The reason for this is that the government has the power and money to implement certain regulations to push the completion of projects. This particular project was derived out of the need to create a common descriptive language in the field of digital design. This need was the direct result of the government having many different vendors supplying integrated circuits to them. Since there was no standard at that time, each of these suppliers had their own descriptive language that was used for design and development. In order to get this project started, the government funded it with $16 million for VHDL design tools as well as an additional $17 million allotted for direct VHDL development.

The next step in the evolution of the language was the Woods Hole Workshop, which was held in Massachusettes during June 1981. The workshop brought together members of industry, government, and academia who were involved and knowledgeable in the design of VHSIC chips. This workshop was used for discussion of VHSIC goals and the initial planning for VHDL began.

The next major step in development occurred in July 1983 when a team consisting of the IBM, Intermetrics, and TI organizations were awarded a contract to develop the baseline structure of the language. In order to get the best outcome, these three companies did not keep the process development results closed but instead allowed public review and suggestions. This constant review and improvement lead to the development of VHDL versions 1 through 7.1.

The final modifications before official release were implemented into version 7.2 in August of 1985. Once this initial launch occurred, the government immediately passed military standard 454, which mandated that documentation was required for any and all government contracts that contained an ASIC (Application Specific Integrated Circuit). An ASIC is "an integrated circuit that is designed using standard logic blocks to reduce the time to market of a new chip." [25]. In order to best adhere to this standard, there was a push for suppliers to use the VHDL language throughout the entire design process.

This final release, however, did not conclude the evolution of VHDL. In February of 1986, all rights to the language were transferred to IEEE (Institute of Electrical and Electronic Engineers) with the responsibility of updating and maintaining it. Upon transfer, IEEE formally endorsed it. This lead to a proposal the following month for a new and improved version to extend the capabilities of the existing language as well as to change and fix any identified problems. After a year of improvements and modifications, VHDL became classified as IEEE standard 1076-1987 (also called 1076.1

or VHDL-87) in December of 1987. This version of VHDL was at first intended not as a design tool, but instead to provide a precise model of circuitry. Along with this new standard, the IEEE VHDL language reference manual was published. Soon thereafter, in 1988, this IEEE standard became approved by the American National Standards Institute (ANSI).

The next significant changes in the language occurred around September 1993. It is stated under the IEEE standard guidelines, that all standards have to undergo a review every five years to determine their future relevance to the industry. Because of this review, VHDL was re-standardized in order to further fix and enhance the language and was later passed as IEEE standard 1076-1993 (also known as VHDL-93). This new standard included improved file handling, predefined standard packages, expanded functions, more consistent syntax, and larger keyword libraries and attributes.

Since this revision, VHDL has become a draft international standard by the International Electrotechnical Commission (IEC). Also, VHDL is no longer just a description language as its initial design was intended, but instead is a design tool with increased modeling capability. These new features allow the simulation as well as the testing of complete digital systems before they are manufactured. This allows companies to save time and money, since wasteful and costly prototypes are virtually unnecessary. VHDL has become one of two widely used hardware description languages. The other popular one is Verilog, which was published by Cadence in 1991 and remained a proprietary language for numerous years thereafter. This business strategy actually allowed VHDL to "catch on" since all of the other vendors were unable to use or modify

Verilog and thus put all of their resources behind VHDL. A good discussion comparing the development of the two languages can be found in [24].

4.2 Basics of VHDL

VHDL is an intricate language that takes a good amount of time to learn and perhaps years to master. This fact is additionally complicated when the person learning it has no previous programming experience. Most publications do not typically take this into account and instead assume prior programming skills. Since this is obviously not always the case, a series of six lessons are presented in Appendix A that cover the basics of VHDL on the level of a novice programmer. These lessons, when accompanied by a short lecture and textbook (to provide further examples) would allow the baseline language to be covered quite thoroughly within a three to four week time frame.

These lessons are presented in an order that would likely be most beneficial to the student. Here is a listing and order of the topics that are covered by the reports and located within Appendix A:

1. ***Introduction & Examples of VHDL***
2. ***Introduction to Architecture Bodies***
3. ***Arrays and Operators***
4. ***Structural Modeling***
5. ***Dataflow Modeling***
6. ***Behavioral Modeling***

CHAPTER V

BASIS OF SLOT MACHINE DEVELOPMENT

5.1 Introduction

The design concept for this thesis originated during a trip to Las Vegas in December of 2000. The need to find a topic for an upcoming undergraduate senior electrical engineering project combined with the overwhelming amount of "one-armed bandits" present in Las Vegas inspired the author and a colleague to further investigate these machines. While the initial senior capstone project concentrated on the mathematical model as well as the many surrounding aspects of gambling such as ethics, business, psychology, etc.; the following discussion is solely on the design and implementation using various software and hardware technologies.

The design description gives a brief history of slot machines to give the reader a feel to how the problem of building one would be attacked. Also, project specifications are discussed and the mathematical model given. This is in addition to other specifications mentioned in order to give a basis for which the various designs will be built and compared upon.

5.2 History & Origin of Slot Machine Technology

Charles Fey (1862-1944) is recognized as the inventor of slot machines. Fey was an American entrepreneur who began manufacturing these devices inside of his San Francisco workshop in 1894. When asked about his new invention, he described them as such, "Take a coin chute for the people to put their money in, and a cash box for the money to go into, and put something in between that will interest the people, and you've invented a slot machine". Fey was a true pioneer of the gaming field and was the true originator of the three-wheel slot (that is commonplace today), which he built in 1898. This slot machine that he invented, known as the "Liberty Bell", was so well thought out and revolutionary that the same basic design is being used in today's high tech gaming facilities.

Fey began marketing these devices throughout San Francisco's Barbary Coast in the late 1890's. However, there was one huge obstacle that remained in his way: California laws during that time period prohibited any type of gambling machines that paid out monetary jackpots. As any great business person knows how to do, Fey looked for a loophole in this law. Instead of paying out a certain monetary sum for lining up three of the same symbols, the owner of the machine would pay out the equivalent in cigars for instance. This is where the symbols originated that are commonplace today. Such popular symbols like the cherry, plum, orange, etc. were used as the designation of the flavor of chewing gum the player would win if he or she matched three of them in a row. However, gambling was like prohibition, when the police were not around, these gaming machines most likely turned back into having monetary payouts. The main difference between the machines of that era and the ones of today are in the general makeup. The machines back then were mechanically driven using springs, wheels, and gears and consisted of three reels that normally would hold 20 symbols. Each symbol would have as likely a chance of being hit on each of the three reels. Nowadays, these machines have evolved into microprocessor-controlled devices in which the mathematical makeup can be programmed and can carry up to five wheels or more.

It wasn't until 1931 that the first state in the United States legalized gambling. It should be no surprise that this state was Nevada. This legislature created the first legal American market for these devices. The 1930's saw these machines spread across America, and in the late 1940's, the famous Bugsy Segal inserted the machines into his Flamingo Hilton, which is still standing in Las Vegas, Nevada. The mindset into which these machines were marketed was quite different back in that era. Slot machines were first used to just entertain the wives and girlfriends of the high rollers until the revenues began supplanting that of the table games. The mechanical machines continued to thrive until Bally's created the first electromechanical slot machine during the early 1960's. This machine titled "Money Honey" was a huge success.

The next milestone in the evolution of slot machines came in 1981 when computerized reel-spinning slot machines were introduced along with video-display gaming devices. Also around this same time, the popularity of slot machines became equal to that of the table games, which at one time was thought to be impossible.

The next decade introduced two new huge developments. The first was that legalized gambling expanded beyond the traditional areas of Las Vegas and other Nevada cities, which opened the way for riverboat casinos. The reason for this legislation was that governments at both the state and local levels were investigating ways in which to increase tax revenue. Seeing the huge impact on the Las Vegas area, they began to explore the possibilities of creating legalized gambling facilities inside their own boundaries. The second development that had a huge impact on the industry of slot machines were federal rulings that allowed for the expansion of gaming on Native American soil. As odd as that sounds, the world's largest casino called Foxwoods is owned by the Pequot tribe and located in Leyward, Connecticut. These two developments helped generate a significant amount of money for the slot makers, which allowed them to enhance their research and development departments. Nowadays, these machines make up 67% of a typical casino's revenue and are almost all microprocessor-based that either spin virtual reels or produce graphic pixels onto a video screen [9]. The trend is now to market these machines as multimedia machines of an enhanced entertainment value than those of yesteryear.

With these technological changes, the internal architecture of these devices had to be varied as well. The traditional architecture was ROM-based, with all of the game code and multimedia effects residing on programmable, read-only memory modules called EPROMs. This architecture, which is still being used in some machines today, has the advantage that the internal composition cannot be changed once it is burned into the EPROM. This is important to maintain the integrity of these games since enormous amounts of money can be at stake. However, some limitations are that EPROMs work at very slow clock speeds, contain very little memory and have only the basic graphical support available. C compilers and DOS-prompt linkers are used to produce these platforms but are constrained severely by the aforementioned limitations. In order to produce the best gaming experience, the security of the EPROM needed to be combined with some technology that would allow for a greater enhancement of multimedia capabilities.

This breakthrough came about in 1997 when the state of Nevada passed legislation that allowed a new platform of gaming devices to use personal computer styled hardware. These new gaming machines actually contained a Pentium processor, a hard drive, and a graphical interface just as a typical PC would have. The technology was not advanced enough to store the game code unto EPROMs but was instead encrypted and contained onto a protected hard disk. These new machines offered significantly more entertainment value than those of previous architectures but were still looked upon by many gaming areas as being security risks. For this reason, a company called Casino Data Systems, developed a fully integrated system in which EPROMs were used to carry all of the game control functions and the PC-based devices were used for only the execution and storage of the multimedia functions. This technology is being increasingly implemented today with amazing results. At a recent trip to Las Vegas, it was observed that single machines could play video poker (with many variations), blackjack, and also other popular games. This is extremely valuable to a casino owner since it allows for greater versatility and provides a more customized experience to the consumer.

Along with the parts of the machine that the consumer can see, there are many other interacting components beneath the surface. I/O functions are used to handle many of the internal utilities such as coin and bill handling, integrity checks, etc. and are programmed to go into "tilt" mode if anything is found to be wrong. Also, the machines are required to be able to handle any unexpected shutdowns quite easily since most casinos do not use uninterruptible power supplies to these machines. The most interesting internal feature is the fact that the complete game history is kept on an EPROM chip so that customer disputes can be easily handled and verified by casino personnel.

Lastly, many of the present day machines are networked to one another. This enables casinos to monitor the activity of not only the machines but also the players. Some increasingly popular things found at casinos today are slot clubs. A person belonging to a slot club would get a card, which looks like a credit card. Whenever the person wants to play a slot machine, he or she would enter the money as normal but also enter this card into a designated slot. The casino is then able to track the amount of activity by this player and offers benefits such as free slot pulls, meals, shows, etc. The consumer benefits since he or she is rewarded with merchandise for just merely playing and the casino benefits from the loyalty toward their particular casino and valuable marketing information. This networking also allows many machines to "communicate" with each other within the casino and also other casinos. This "communication" allows for such things as progressive jackpots throughout the city that grow once a credit is inserted into one of the many machine locations.

5.3 Discussion of Mathematical Models

Now that the actual history and makeup of these machines have been discussed, the mathematical aspects surrounding gaming and slot machines will be examined. The mathematics and probability concepts of slot machines are perhaps the most important in the development process. Even though the outcomes are virtually random, the math model is what ultimately determines how much the house or player is going to win over a long stretch of time. However, there is a thin line between making the machine profitable for the casino yet entertaining for the user.

The first thing that must be determined before the model can even be created is the bottom line payout. There are laws that govern the minimum percentage that a slot machine must payout in a legalized gambling state. For example, in the state of Nevada, the minimum payout is 75%; in New Jersey, this number is 83%. However, despite these small minimum required payout percentages, most modern games pay out well over 90%. The reason for this is to give the user a better sense that he or she is winning. This in turn will prompt the player to play this certain machine for a longer period of time and thus will be more profitable for the casino than the machine in which the player only puts two or three credits into and walks away.

This leads to the second main statistical number in the math model: the hit frequency. The hit frequency is the percentage associated with the number of times the user hits a winning combination on a particular turn in relation to the number of times the user has played. It is usually inversely related with the size of the jackpot; the bigger the jackpot, the less frequent the player will win. This is another tradeoff that must be considered while designing a slot machine. The jackpot must be big enough to warrant the player to try his or her luck at this machine and the hit frequency should be high enough to keep him or her there, all while still being profitable for the casino. As one can see, there are many variables that must all be considered by a statistician while trying to keep the player's psychology in mind.

Other variations that can be considered are different types of jackpots. There are two main types of jackpots: flat tops and progressive. A flat top jackpot is one that will always pay the winner a predetermined set amount. A progressive jackpot is one that grows through a percentage of the amount of coins being wagered. These progressive jackpots can be networked throughout many different machines and casinos and can grow to over millions of dollars. It is important to remember though that the hit frequencies and sometimes the payout percentage will be lower on these machines since the money will need to be obtained from somewhere to pay these huge amounts. For example, on the Megabucks slot machine, the jackpot can often reach over five million dollars but the payout percentage is only 87% as opposed to other similar machines with not quite as high a jackpot but with a payout of 95%.

Another variation on jackpots can be on what is necessary to win it. On some machines, a user must play multi-coins at a time to even be eligible to win the shown jackpot. This is very effective in getting a player to play more than one coin at a time since hitting the potentially winning combination while only playing a single coin can be very, very frustrating!

Once the bottom line numbers are determined for the gaming devices, it is time to effectively implement them onto the slot machine. Even though some decisions have been made, there are plenty more to make! The first of these is most likely how many wheels to use on the machine. The most traditional slot machines almost always had three spinning wheels of characters. However, it seems as if this trend has become less and less popular. The advantage with having more than the traditional three wheels is that there is a great deal more freedom for the designer when it comes to the odds of the game. More wheels enable greater and more creative payouts. However, the increase of wheels also makes the user feel that there is an increased difficulty in the game play and gives the sense that it is harder to hit big when five cherries must be hit in a row as opposed to three on another machine.

Another similar decision is the determination of how many characters or "stops" to put onto each of these wheels. The pros and cons for these are the same as in determining the number of wheels to use. However, the digital age has brought on a unique twist to this aspect. In previous years, one stop on a gear corresponded to one visible character on the screen. Nowadays, random numbers are produced by random number generators in order to determine which character is shown on the screen. This enables a manufacturer to "weight" certain characters. This process is as follows:

1. Three random numbers (or however many wheels there are) are generated by the random number generating device.
2. This number is divided by a fixed number (which is a power of two for simplicity) and modulo math is performed. For example, a fixed number of 64 would be used to divide the random number and would enable the possibility of the numbers (0-63) for each wheel.
3. These 64 possibilities are then assigned to actual characters on the wheel. A typical wheel might only contain 18 real characters so that some characters might have several of the random possibilities assigned to them while another character might just have one.

Once this process is determined, the mind games for the manufacturer still continue. A typical machine would use this process in order to weight non-paying characters such as blanks more heavily than paying characters. Also, such places as blanks right above and below high-paying spots on the wheel are typically "heavily weighted" to give the illusion that a big jackpot was nearly missed. This methodology is also used in making the first reel the loosest of them all. A high paying character would come up often on the first reel but be increasingly unlikely to come up on following reels. This weighting system was such a great innovation that it has been actually patented. A company by the name of IGT had the patent until 2001 on using fixed numbers greater than or equal to 64 for modulo math.

Once again, this emphasizes that there are many variables taken into consideration while determining the math makeup of one of these machines. Once all of these determinations have been made, it is time to implement the math model into the machine. This can be done using a computer package similar to Macromedia Director, which integrates a sketched math model into a computer format. Once approval is granted for the project, this computer format would be polished and finalized using C++ code or another high-level language unto an EPROM. This would conclude the process for the math segment of the gaming device and then would be "livened" up using an artwork software package such as Photoshop.

5.4 Project Specifications

Using the information presented in Section 5.3, some specifications for the design were decided upon. The variables that needed to be addressed included the following: number of wheels, range of random numbers to be generated per wheel, number of physical symbols and blanks per wheel, payout percentage, jackpot payout, and the hit frequency. These variables and their corresponding specification range are presented in Figure 5.1 and are discussed in the text that follows.

Variable	Spec. Range
Number of Columns/Wheels	3 - 5
Range of Random Numbers Per Column	8 - 64
Number of Symbols Per Column	4 - 10
Number of Blanks Per Column	4 - 10
Payout Percentage	83% - 98%
Hit Frequency	8% - 15%
Jackpot Type	Fixed
Jackpot Payout	250 - 5000

Figure 5.1 – Typical range of slot machine variables

A typical slot machine today has three to five wheels so this is the range I used for the actual model. From the discussion in Section 5.3, the relationship between number of wheels and mathematical freedom of the designer is directly proportional and inversely proportional to consumer confidence of hitting the jackpot.

The range of random numbers generated per wheel determines the flexibility of the mathematical model as well as the number of symbols and blanks per wheel. Obviously, there cannot be more total symbols (including blanks) per wheel then there are “spots” produced by the random number generator. Since International Gaming Technology (IGT) had a patent on the maximum number of “spots” allowed to be produced by a slot machine, which was sixty-four, this is the number used as the maximum specification for this design. A minimum of eight “spots” was determined since it is the lowest number that could be used while still having a somewhat flexible use for the designer.

The number of physical symbols and blanks per wheel are typically the same number. This is due to the fact that two symbols are separated by a blank on the conventional slot machines. However, in this design case, traditional circular reels did not necessarily need to be used so this normality was not a limitation. Also, as stated in the above paragraph, the total number of symbols and blanks must not exceed the number of “spots” produced by the random number generator. A ratio between the number of “spots” and the number of total symbols produces a number that is directly proportional to the flexibility of the mathematical model. Thus, the higher the ratio, the greater the flexibility allowed.

The previous variables discussed the physical construction of the slot machine whereas the next three variables are more related to the mathematical structure. Both sets of variables are interrelated, however, as evident by the previous discussion. The first of these variables and also the one most commonly referred to, is the payout percentage. The minimum payout percentage for this model was determined to be 83% since that is the lowest percentage allowable by law in New Jersey. Nevada has its minimum set at 75%, but the choice of 83% was used to make this model able to be used in all legalized gambling states. A maximum of 98% was set by looking through slot magazines that listed actual ranges of new slot machines on the market. This percentage pays back a good deal of what it takes in but still allows for a 2% profit for the casino. High percentage payouts are found more frequently on high denomination machines since obviously a greater amount of money is being bet and being lost over the long run.

The hit frequency variable refers to the odds of the player hitting a winning combination each time the "wheels are spun". This is believed to be the most important parameter since it directly correlates to player retention. By consistently hitting winning combinations, the player feels that he or she is winning, even if these jackpots are small ones. For this purpose, I have determined that I would like to set the specifications of this parameter to be in the high range of 8%-15%. This figure is often advertised in a misleading fashion today to be around the 50% mark for some machines. The figures that these companies use is based upon maximum credit play (which is often up to 90 credits per play). A winning combination of 50% includes all possible winning combinations even ones that include as few as 1 credit payback. It can obviously be seen how misleading this statistic can be since even though there is a 50/50 chance of hitting a winning combination, the player is far from winning by placing a 90 credit bet and receiving 1 credit back in return. The range of 8%-15% in this design is accurate and not misleading since there will only be one winning line possible per play.

The last variable related to the mathematical structure is the jackpot payout. This variable refers to the highest payout possible on one spin of the wheels. A fixed jackpot was decided upon since they are still the most popular jackpot design. A high jackpot increases the visibility of the player but also typically decreases the above-mentioned hit frequency since there will be less lower jackpots to compensate for the large amount of revenue lost when the jackpot is hit. High "spots" to total symbols ratios allow for higher jackpots since the design flexibility is increased. A minimum specification of 250 was decided upon because anything lower would discourage players from playing this particular machine since most slot players are looking to "get rich quick" and depending on the denomination played, less than 250 times the initial bet does not usually make someone rich. A maximum of 5000 was used to allow for the highest hit frequency as possible since these variables are typically inversely correlated.

5.5 Actual Model Parameters

After much trial and error while using the project specifications as a guideline for the slot machine design, the actual mathematical model was determined. The characteristics of this model are given in Figure 5.2. It is impossible to list or describe the entire thought process of how this particular model came about with all of the many variables present. However, a description to defend the validity of each decision will be presented in addition to a thorough investigation of all of the important mathematical statistics involved.

The decision of three wheels for the model was determined for two reasons. The first reason is that most current "mechanical looking" slot machines have three wheels while the video slots normally have five. Since the ultimate goal here is to implement the model onto hardware rather than a pure software version, three wheels would be more typical a choice. Also as stressed before, the complexity of this design has been simplified as much as possible for the benefit of the reader. The next variable in Figure

5.2 refers to the range of random numbers per wheel. Fifteen was the number of choice mainly due to the nature of the "random number generator" used in Section 7.2 of this report. This random number generator (in the chip design) consists of a shift register that produces a sequence of sixteen numbers and in this case the number '0000' is being omitted for reasons discussed later in Section 7.2. It is important to note that each of the wheels in this model will have a random number generator assigned to it.

Variable	**Spec. Value**
Number of Columns/Wheels	3
Range of Random Numbers Per Column	15
Number of Symbols Per Column	5
Number of Blanks Per Column	5
Payout Percentage	93.48%
Hit Frequency	12.27%
Jackpot Type	Fixed
Jackpot Payout	500

Figure 5.2 – The actual model parameters

Once this variable was set at fifteen, the next task was to determine the total amount of physical symbols (including blanks) to use in the design. It is obvious in this case that the number could be no higher than fifteen or else there would be unused symbols with no random number associated to them. The resolution was to provide nine total symbols (five physical symbols and four blanks) to allow six extraneous "spots" for design flexibility as well as to provide a realistic number of symbols on each reel.

These variables dealt mostly with the physical construction of the design whereas the next ones deal with the mathematical structure. Since the origin of this design was more of a trial and error nature as previously discussed, it is hard to pinpoint the reason that these figured were picked other than the reason as to meet the outlined specifications. Figure 5.3 lists the number of "spots" assigned to every individual color and each column.

	Column 1	Column 2	Column 3
Red	1	1	1
Blue	2	1	1
Green	3	2	1
Orange	3	3	1
Purple	2	4	4
White	4	4	7
Total	15	15	15

Figure 5.3 – The number of "spots" assigned to each color in each column

The next logical step after creating Figure 5.3 was to actually assign each of the numbers (one through fifteen) to one of the colors. This was done on as random a basis as possible to try to maximize the unpredictability of the entire machine. The number assignments for each color in each column are given in Figure 5.4.

Digit	Column 1	Column 2	Column 3
1	Blue	Orange	Purple
2	Orange	Green	Red
3	White	White	White
4	Red	Purple	Blue
5	Orange	Blue	Purple
6	Green	Green	White
7	White	Red	Purple
8	Green	Purple	White
9	Purple	White	White
10	White	Orange	Green
11	Orange	Purple	Purple
12	Blue	White	White
13	Green	Orange	Yellow
14	White	Purple	White
15	Purple	White	White

Figure 5.4 – Corresponding digit to color by column

Figure 5.5 on the following page uses the data shown in the previous two figures and displays the payout for each winning combination as well as the probability that each winning combination will occur. The colors can be representative of any type of symbol used. The total payouts for each combination are summed and a total payout percentage of 93.48% is calculated. This falls within the project specifications of between an 83% to 98% payout. Note that the colors represent different symbols while white is a blank.

Possible Combinations	Total Combinations	Payout	Combination Name	Total Payout
1	15*15*15 = 3375	500	3 Red's	500
2	15*15*15 = 3375	100	3 Blue's	200
6	15*15*15 = 3375	50	3 Green's	300
9	15*15*15 = 3375	25	3 Orange's	225
14	15*15*15 = 3375	20	2 Red's then ANY	280
28	15*15*15 = 3375	10	2 Blue's then ANY	280
210	15*15*15 = 3375	5	1 Red then ANY 2	1050
32	15*15*15 = 3375	3	3 Purple's	96
112	15*15*15 = 3375	2	3 White's	224
				3155
	Winning Percentage (Total Payout/Total Combinations):			93.48%

Figure 5.5 – The payout percentage of the slot machine model

Figure 5.6 on the following page lists the probability for each combination during each play. The total of these probabilities gives the total hit frequency. As was mentioned in the project specifications, an attempt was made to make this number as high as possible. As can be seen from Figure 5.6, the hit frequency of this model is a respectable 12.27%. As can be seen from the various figures, it is apparent that different colors have different probabilities of occurring on each wheel. Also, by viewing the color green on Figure 5.3, it is also apparent that the same color can have different probabilities depending on which wheel is being discussed. The design was created like this to increase the user's feeling that he or she was closer to winning then actually might have been.

Combination Probabilities		
3 Red's	1 * 1 * 1 = 1/3375	0.03%
3 Blue's	2 * 1 * 1 = 2/3375	0.06%
3 Green's	3 * 2 * 1 = 6/3375	0.18%
3 Orange's	3 * 3 * 1 = 9/3375	0.27%
2 Red's then ANY	1 * 1 * 14 = 14/3375	0.44%
2 Blue's then ANY	2 * 1 * 14 = 28/3375	0.83%
1 Red then ANY 2	1 * 14 * 15 = 210/3375	6.22%
3 Purple's	2 * 4 * 4 = 32/3375	0.95%
All White's	4 * 4 * 7 = 112/3375	3.32%
Non - Winning	3375 - Winning = 2961/3375	87.73%
Hit Frequency	Winning/Total = 414/3375	12.27%

Figure 5.6 – The probabilities for each combination on any given turn.

CHAPTER VI

THE COMMON SOFTWARE PACKAGE APPROACH

6.1 The Microsoft Excel Approach

Microsoft Excel was chosen as the software package of choice to model a slot machine since it is a program that many people use everyday and are thus familiar with it. Also, it shows that a simple program with flexibility and programmability can be used for purposes that do not typically come to mind.

For the three slot machine designs contained within this report, they can be broken into these main parts: the random number generator, the physical output seen by the user, and the interface between the first two. For this reason, these discussions of the designs will also be broken into these same parts.

Designing the random number generator in Microsoft Excel was perhaps the easiest of the three parts. Excel has a built in random number function called 'rand()' that "returns an evenly distributed random number greater than or equal to zero and less than one" (as taken from Microsoft Excel's help files). Since the goal is to produce random integers between one and fifteen as discussed in Section 5.5, some slight modifications had to be made to this function. In order to achieve the goal, the 'rand()' function was used in the formula as so: '(INT(RAND()*15))+1'. This formula multiplies the random fraction by fifteen so that the number can take on any value between zero and fifteen (not including fifteen). From this, the 'int()' function is then used to drop any fractional parts of the number and keep just the integer value. Since the range is between zero and fourteen, one is added to the final number to produce the desired results. This formula is then placed in three separate cells so that three independent random number generators are produced; one for each virtual wheel.

The second part of the Excel design involves the physical output seen by the user, which is divided into four additional sub-sections. These sub-sections include an interface for the user to begin the game, visual representation of showing the symbol selected on each of the three wheels during a turn, a noticeable signal that a winning combination has occurred, and the resulting payout (if any). The first sub-section was created by inserting a clip-art picture resembling a handle into the Excel spreadsheet. This picture was then assigned a macro, which would run anytime that the "handle" clip-art was clicked. The actual coding of the macro will be discussed later. The manner in which the symbols were to be displayed on the spreadsheet was the next concern. In order to accomplish the mathematical model as described in Section 5.5, there must be actual symbols and blanks that can be seen by the user. The way this design fulfilled the requirement was to make a three column by nine row space available for these symbols. Each of the colors described in Section 5.5 are then actually written out in the individual cells. A base color of brown was used as the background color to show when the symbol is "off", while the color typed in the cell would be the background color when the symbol is "on". A visual of this setup can be seen in Figure 6.1 on the following page, which displays a non-winning combination of "Green-Red-White".

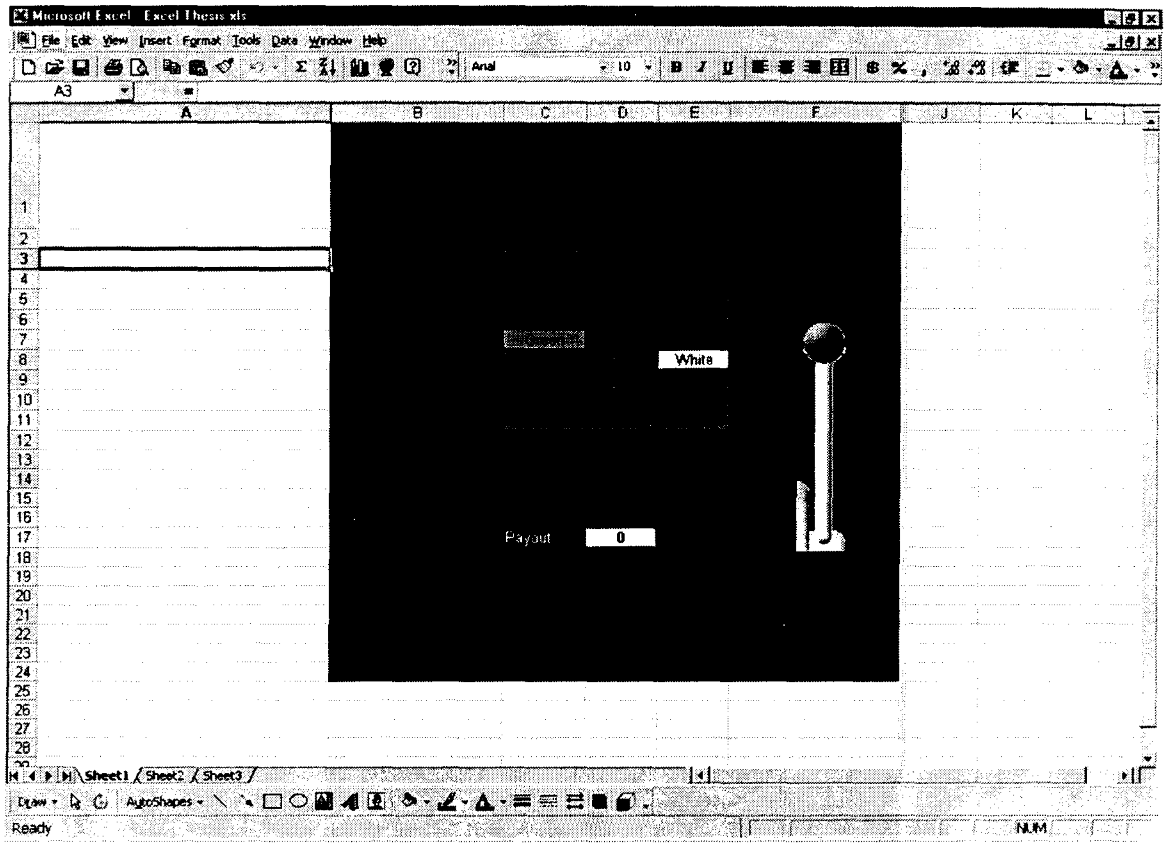

Microsoft Excel - Excel Thesis.xls
File Edit View Insert Format Tools Data Window Help
A3
White
Payout
0
Sheet1 Sheet2 Sheet3
Ready
NUM

Figure 6.1 – Display of a non-winning combination as shown to the user

The third sub-section provided the challenge of sending an obvious indication to the user to signify that a winning combination has been hit. This was accomplished by displaying the word "WINNER!" in large red font at the top of the screen whenever a winning situation occurs. This is depicted in Figure 6.2, which displays the output of the slot machine during a winning combination of "White-White-White".

Lastly, the payout had to be displayed as well when winning combinations occurred. This is listed at the bottom of the screen and is zero for non-winning combinations and the payout value for winning ones.

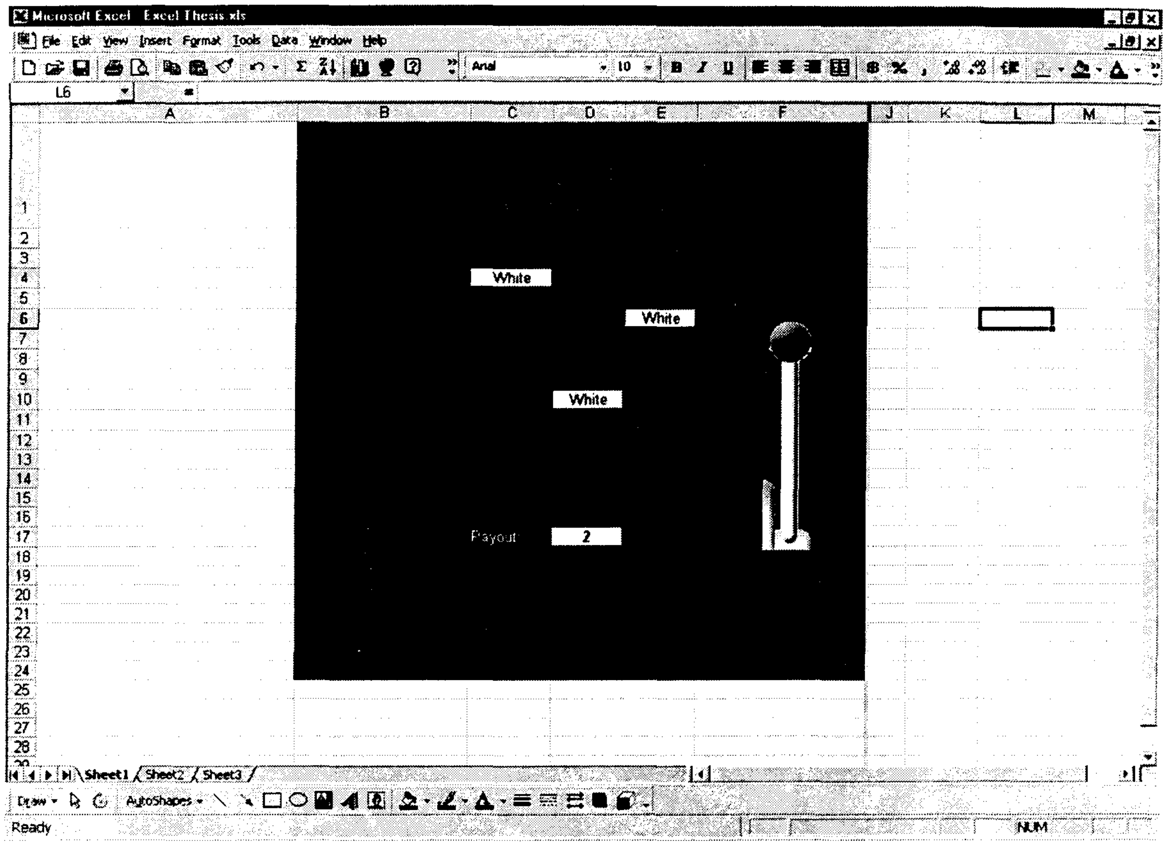


Figure 6.2 – Display of a winning combination as shown to the user

The last part of the slot machine is also the most intricate one to explain since it involves interfacing the random number generator with the physical model seen in Figure 6.1. The first challenge was to get the handle to physically begin the start of a turn upon being clicked. In order to do this, a macro was created and assigned to the handle so that it runs whenever a user clicks on it. A macro is just a small program created in Visual Basic that is run to completion whenever the specified event occurs (in this case, the clicking of the handle is the event). The coding of this macro is shown as Figure 6.3. The manner in which this macro works is a very simple concept as is evident by the concise coding shown in Figure 6.3. The macro invokes the delete command to run in Excel, which produces an effect that is not often observed. The delete command forces

Excel to recalculate all of the formulas present in the worksheet, including generating new numbers for the 'rand' command, thus making it a perfect fit for this application. In summary, each time the handle is clicked, the macro calls the delete command, which in turn re-generates the three random numbers needed to run the entire slot machine. This solves the problem of how to use the handle as a start function. It is important to note that these random numbers should not be visible to the end user as can be seen in Figure 6.1. The user should never be able to see which symbols are represented by each of the random numbers since reconstruction of the mathematical model would be simplified. Part of the excitement and mystery of slot machines is not knowing the exact payout of each machine and is considerably lessened when the mystery can be calculated!

```
Sub Spin_Reels()
'
' Spin_Reels Macro
' Macro recorded 2/24/2002 by Luke Pascute
'
   Selection.ClearContents
End Sub
```

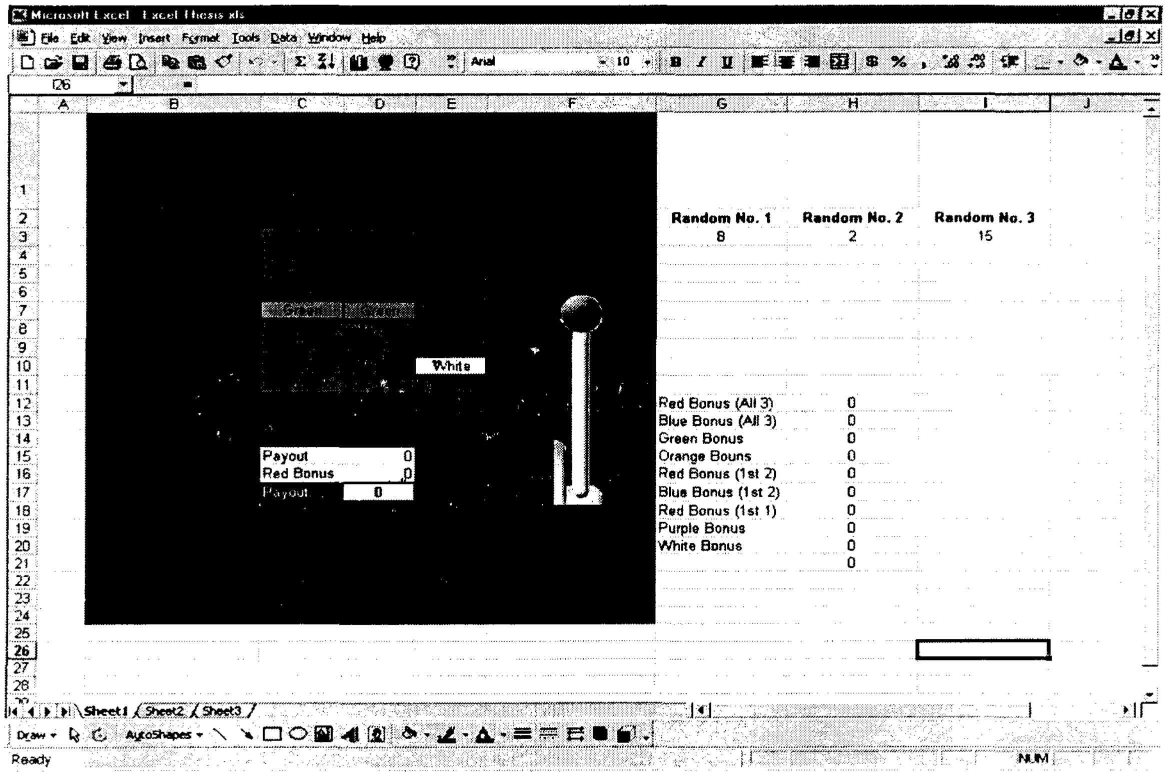
Figure 6.3 – The coding inside of the "handle" macro.

The random numbers generated are then connected to the simulated symbols on the spreadsheet through use of the "conditional formatting" ability in Excel. This option can be found on the drop down menu under FORMAT >> CONDITIONAL FORMATTING and allows the appearance of a particular cell to be changed depending on certain circumstances. This option allows the base color of brown to be the background color of a particular cell when a random number is generated that is not assigned to that symbol. When a random number assigned to a particular symbol is produced, the cell takes on the color typed inside of it. Each of the cells in the grid are then assigned at least one of the numbers one through fifteen so that all fifteen numbers are accounted for in each column (See Figure 5.7). This in effect "lights up" one of the cells in each column. This is the same manner that a typical slot machine works. It generates a random number for each of the wheels present and stops the wheel on the symbol (or blank) that has been designed to correspond to that number.

Once the wheels display the selected symbols, the program needs to determine whether it is a winning combination and if so, how much the payout needs to be. The way that this problem was attacked was to make it as structural and easy to understand as possible. Using intricate IF statements, the same outcome could have been achieved but it would have been more difficult for the reader to understand. The final draft of the design is shown in Figure 6.4, which unlike Figure 6.1, shows the coding involved that is normally hidden from the user. As can be seen from this figure, each possible winning combination is listed with a number to the right of it. Built in 'IF' statements are used to display a zero when that particular combination has not been hit and a one when that bonus has been achieved on the machine. Once these 'IF' statements determine if a winning combinations has been hit, additional 'IF' statements are used to assign the corresponding value to each combination.

The two main cells that determine payout can be found in the normally hidden section of the machine shown in Figure 6.4 and are labeled 'Payout' and 'Red Bonus'. The reason that an initial "payout" and a "red bonus" need to be calculated separately lies within the structure of an Excel 'IF' statement. These statements allow only seven

"ELSIF" statements within each one. Since there is a total of nine different winning combinations with different payouts assigned to them, they need to be broken up into two separate statements. The most logical way to do this was to separate the two bonuses involving the first cell being red and the first and second cells being red from the rest of the other bonuses. The final payout is what is tabulated and shown to the user in the form seen in Figure 6.1.


Microsoft Excel - Excel Thesis.xls
File Edit View Insert Format Tools Data Window Help
Arial
10
White
Sheet1 Sheet2 Sheet3
Draw AutoShapes
Ready
NUM

Random No. 1	Random No. 2	Random No. 3
8	2	15

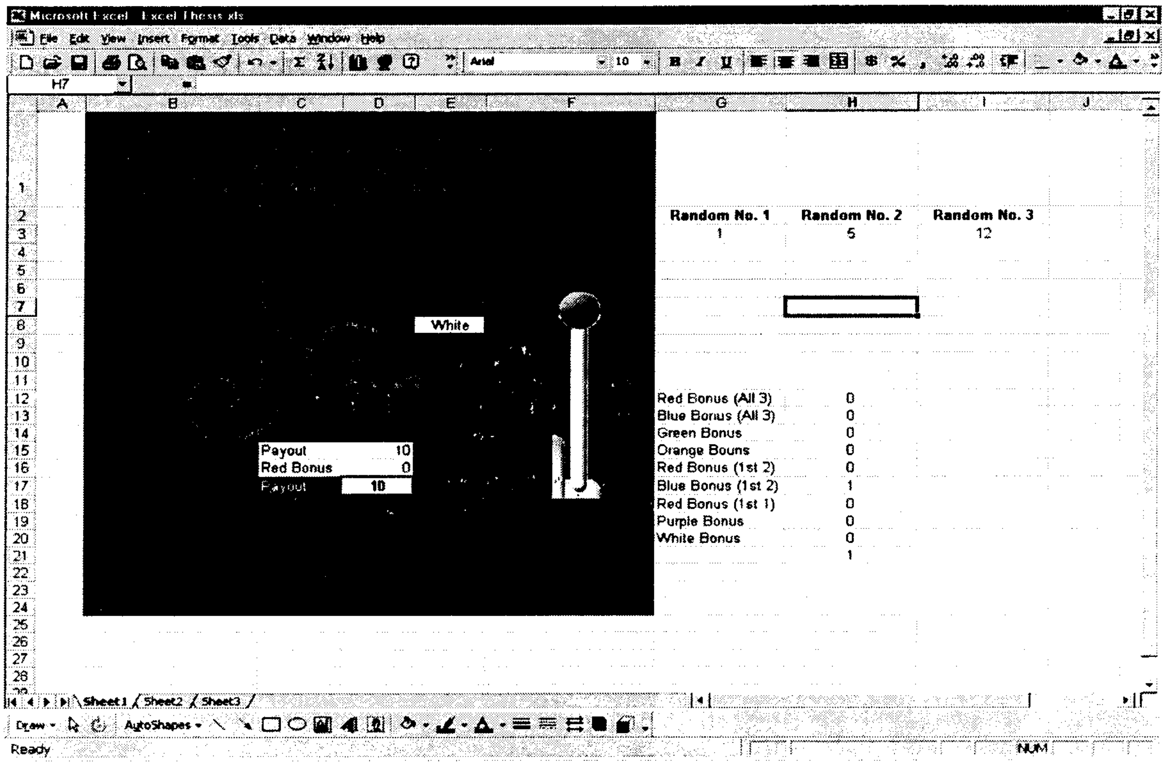
Red Bonus (All 3)	0
Blue Bonus (All 3)	0
Green Bonus	0
Orange Bouns	0
Red Bonus (1st 2)	0
Blue Bonus (1st 2)	0
Red Bonus (1st 1)	0
Purple Bonus	0
White Bonus	0
	0

Payout	0
Red Bonus	0
Payout	0

Figure 6.4 – Display of a non-winning combination as shown "behind the scenes"

Lastly, a total is taken of all of the winning combinations in order to display if one has actually occurred. Since the number next to each of the combinations is normally a zero if no winning combination occurs, the sum would be zero. When any of the combinations are hit, a one is displayed next to it and a total of one would be summed. An 'IF' statement is used at the top of the screen to display the word "WINNER!" real large whenever ANY winning combination occurs. This also serves the purpose of attracting attention to other potential users that this machine has just won someone money. Figure 6.5 shows the internal composition of a winning combination.

A summarization of the coding behind each of the cells is provided in Appendix B. This provides an almost comprehensive description for the reader if he or she would like to try to recreate this design.


Microsoft Excel - Excel Thesis.xls
File Edit View Insert Format Tools Data Window Help
H7
Sheet1 Sheet2 Sheet3
Draw AutoShapes
Ready
NUM

Random No. 1	Random No. 2	Random No. 3
1	5	12

White

Payout	10
Red Bonus	0
Payout	10

Red Bonus (All 3)	0
Blue Bonus (All 3)	0
Green Bonus	0
Orange Bouns	0
Red Bonus (1st 2)	0
Blue Bonus (1st 2)	1
Red Bonus (1st 1)	0
Purple Bonus	0
White Bonus	0
	1

Figure 6.5 – Display of a winning combination as shown “behind the scenes”

CHAPTER VII

THE COMMON LOGIC HARDWARE APPROACH

7.1 Description of Logic Packages Used

This chapter deals with incorporating the mathematical model into a small-scale slot machine using common digital logic chips. This design is made up of five different logic chips that will be examined within this section. These chips include the XOR digital logic gate (74x86), a comparator (74x85), a decoder (74x154), a counter (74x163), and a shift register (74x194). Each of the symbols and corresponding truth tables can be found in Appendix C.

The 74x86 chip contains four two-input XOR digital logic gates. The active-high output is asserted when exactly one of the two inputs is high. Since the output is asserted only when the two inputs are different, an individual XOR gate can be used as a 1-bit comparator (comparators will be discussed more in depth with the 74x85 package). Lastly, this gate can be used in conjunction with counters and shift registers to implement various counting schemes as is the case in this design.

The second package to be discussed is the 74x85 4-bit comparator. This is a standard MSI package that is commonly found within computer systems and networks. The 74x85 takes in two 4-bit binary numbers as inputs and compares them against one another. After this comparison, either the less than, greater than, or equal to, active-high output pin is asserted. Pins with these same labels are also present as input pins to provide the capability of cascading two or more of these chips in order to create comparators of a larger magnitude than 4-bit. It should be noted that the truth table shown in Appendix C.1 does not list all possible combinations since there would be 2^{11} or 2048 possibilities! The output should be quite apparent to the reader for any combination not listed, however, since the functionality of this package is quite simple. It is important to note though that only one of either the less than, greater than, or equal to inputs and outputs should be asserted at any given time if the chip is being used properly.

The 74x154 commercial MSI package works as a 4-to-16 bit decoder. The decoder receives a 4-bit signal unto its four input pins and asserts one of the sixteen active-low output pins. Since there are 2^4 or sixteen possibilities from the four input pins, there is one particular output pin corresponding to each combination. However, for the decoding process to work properly, both of the enable inputs present on this chip must be asserted. This package, like the 74x85, can be cascaded together to construct larger sized decoders.

The fourth package used in the design is the 74x163 4-bit synchronous counter. The word "synchronous" refers to the fact that all of the states change on the rising-edge clock pulse since they are all tied to the same clock. This package is perhaps one of the most popular MSI counters since it can count in multiples of 2, 4, 8, or 16 independently by using either one, two, three, or all four of the outputs respectively. The active low load input, when deasserted, allows the next state to be specified by the inputs D, C, B, A. The active low clear input allows the next state to be the value '0000'. When either

ENP or ENT is not asserted, the counter is held in its current state. The RCO output produces a high logic level when the output count is '1111' and the value of the ENT input is '1'. Once again, as is the case with many of the previous packages, the 74x163 can be cascaded in order to produce counters with ranges larger than 16.

The last chip to be examined is the 74x194 universal 4-bit, bi-directional, parallel in & out shift register. This chip is very popular due to its extreme flexibility. It allows the contents of its outputs to be shifted in a left or right direction and can be "wired" to perform the functions of almost all the other common shift registers as well. In addition to the ability to shift in the left and right directions as mentioned above, a hold and load function is also present on this chip. A common use of this package is to use other digital logic chips with conjunction with it to produce a "random" counter, which cycles through all sixteen combinations in a non-sequential fashion.

7.2 Interaction of Logic Chips in Implementation

As discussed in Chapter 6, the layout of this machine can be broken up into three different parts: the random number generator, the physically seen output, and the other internal hardware. The first of these parts to be discussed is the random number generator. When it comes to the computing world, random is a term that virtually does not exist. The only real random events are the ones that occur in nature and cannot be predicted by scientific methods, which makes the only real way to incorporate random into a computing environment is through an interface with nature (such as amplifying noise and sampling it). However, in most applications, it is sufficient enough to use a "pseudo-random" generator. This style of number generation is not truly random but it is so close that the normal user would not be able to notice. Pseudo-random number generation is much easier, efficient, and cheaper technique so therefore it is the one of choice for this design. The purpose of the pseudo-random generator is to randomly generate three numbers when the 'BEGIN' switch is asserted. This approach uses a linear shift register to continually produce four random binary numbers at the desired clock specification until the user asserts the 'BEGIN' input, which will hold the numbers being generated at this exact time. Once the game is over, the random number generator would work again and generate more random numbers until the same sequence occurs. The reason that the randomness can be assured is that the user has no visible means of seeing just what numbers are being produced at any given time along with the fact that the average user would not know that a 74x194 chip was being used as a random number generator or the actual clock cycle speed for each reel.

The shift register is hooked up so that three of the four binary inputs are wired to ground while the fourth is wired to 5 Volts. Outputs "Qa" and "Qb" of the linear shift register are connected to the inputs of an XOR gate. The output of the XOR gate is then connected to the 'SL' input of the shift register. This allows the outputs of the shift register to cycle through all of the binary numbers starting from '0001' through '1111' in a pseudo-random fashion. All of the above-mentioned connections can be viewed by referencing the schematic in Figure 7.1 at the end of this section.

The second part of the slot machine is the other hardware components besides the random number generator. The chips themselves have already been discussed in depth but the integration of them will be discussed further here. The 74x163 counter is used to cycle through all of the possible combinations starting from '0000' through '1111' in numerical order. These numbers are fed into a 4-to-16 decoder and also the comparator. The comparator compares the number being produced by the 74x163 counter against the one that was produced by the random number generator. Once these two numbers are equal, the comparator's "equal to" output is asserted and is used to stop the 74x163 from cycling. Once the cycling has ceased, the 4-to-16 decoder asserts one of the 16 total outputs possible and thus becomes the selected symbol for that reel. The next reel is then asserted and the process continues until a symbol has been selected by all three and remains there until the begin switch is asserted again. Once more, these connections are available for viewing in Figure 7.1.

The third part of the design that has yet to be discussed is the visible part to the user. The three reels of the slot machine could be simulated using three different columns of multicolored LED's. There are nine visible lights in each reel with a color LED representing a symbol and a white (or clear) LED representing a blank. This output would be displayed in exactly the same manner as is modeled in the Excel model except that it would actually be hardwired to the circuit. The digital logic chips along with the main power and ground are all wired to S/K sockets (breadboards) and a begin switch is constructed on the Heathkit Trainer so as to simulate the beginning of a game when a coin is normally inserted.

In order to verify the functionality of this circuit, the design of a single reel was configured and simulated using MicroSim's PSpice Version 8. While this could have been done using the MAX+PLUS II software as well, PSpice was used in order to show the multitude of tools available at the university to the interested designer. The aforementioned schematic shown in Figure 7.1 was used as the input for the simulation.

Through observation of this figure, the previous discussions in this chapter should become more apparent. An input signal 'BEGIN' is connected to a clock signal, which is used to simulate a user starting the machine. Each of values 'C1R1' to 'C1R16' would represent all of the possible random values for a particular reel. Since the 74x194 is configured to never take on the value zero, it is impossible for the 'C1R1' value (which represents 0) to be selected. For the interested reader, the programming of the four clocks present in this device is shown in Appendix D.1.

Appendix D.2 contains two timing diagrams that were generated upon simulation of the circuit in Figure 7.1. The first diagram shows the circuit's behavior for times 0s to 700us. It can be seen from this figure that once the begin input is asserted, the 'SHIFT1' internal signal (which is the output of the 74x194 linear shift register) holds at it's present number (E_{16} in the first case and B_{16} when 'BEGIN' is asserted for a second time). The 'COUNT1' internal signal (which is the output of the 74x163 counter) would continue counting until the same number as was produced by the random number generator (74x163) was reached. In the first assertion of the 'BEGIN' input, this value was never reached before 'BEGIN' was de-asserted. This would be a problem in real life but generally a delay would be built into the slot machine that would hold the 'BEGIN' signal for at least one count cycle at the absolute minimum. During the second assertion of the 'BEGIN' input, it can be seen that the value of B_{16} is reached by the counter while 'BEGIN' is still asserted and thus holds it's value at $B_{16.}$ It can be seen that the system output that would be connected to the physical output seen by the user is also held and is represented by the ‘C1R12’ signal (representing a value of 11 in the mathematical model).

The second timing diagram in Appendix D.2 is a magnification of the first. This was provided so that it would be possible to see the counting sequence of the ‘SHIFT1’ signal. Since the clock pulse for it is moving so quickly, it is impossible to distinguish anything legible from the first timing diagram. The second shows the characteristics between 0s and 30us and clearly demonstrates that the ‘SHIFT1’ is indeed producing a non-sequential counting sequence. It can also be seen that this counting sequences ceases at almost the exact instance that the ‘BEGIN’ input is asserted.

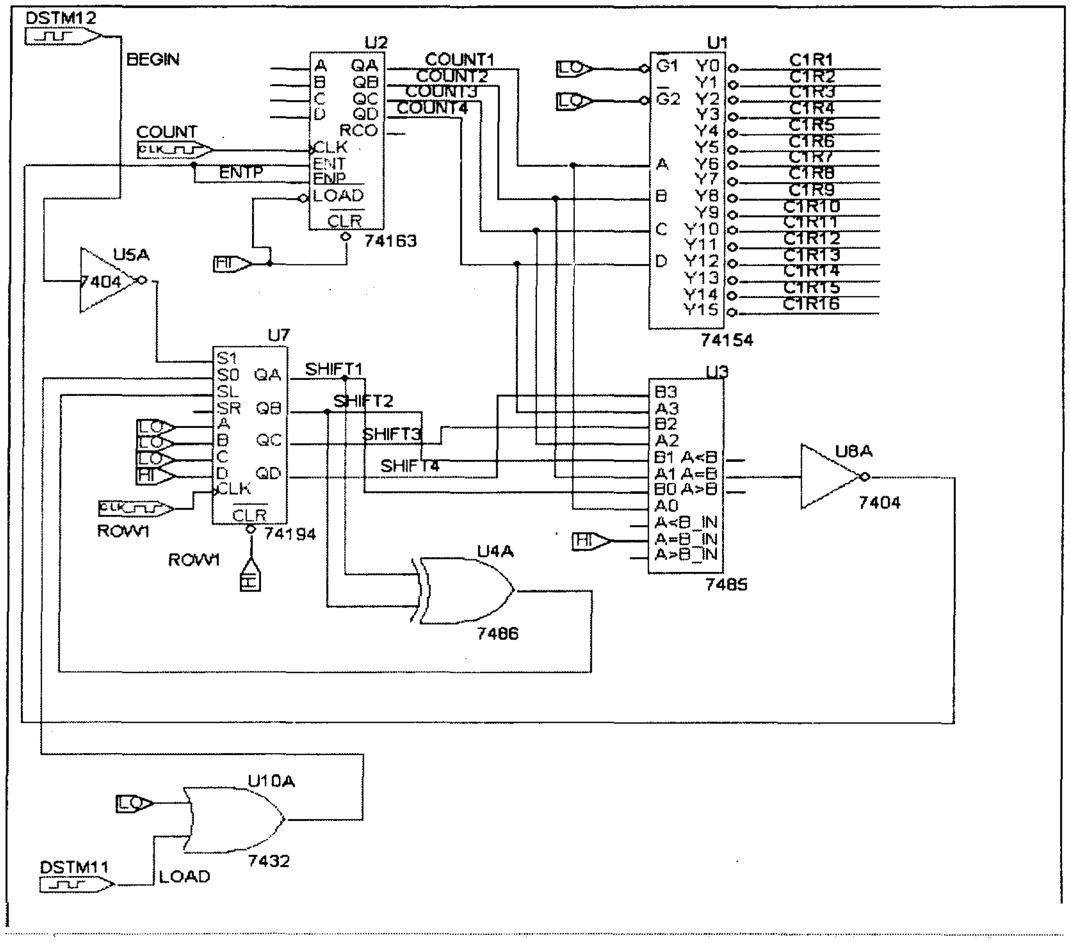


Figure 7.1 – The design for a single-reel of the slot machine as seen in PSpice.

CHAPTER VIII

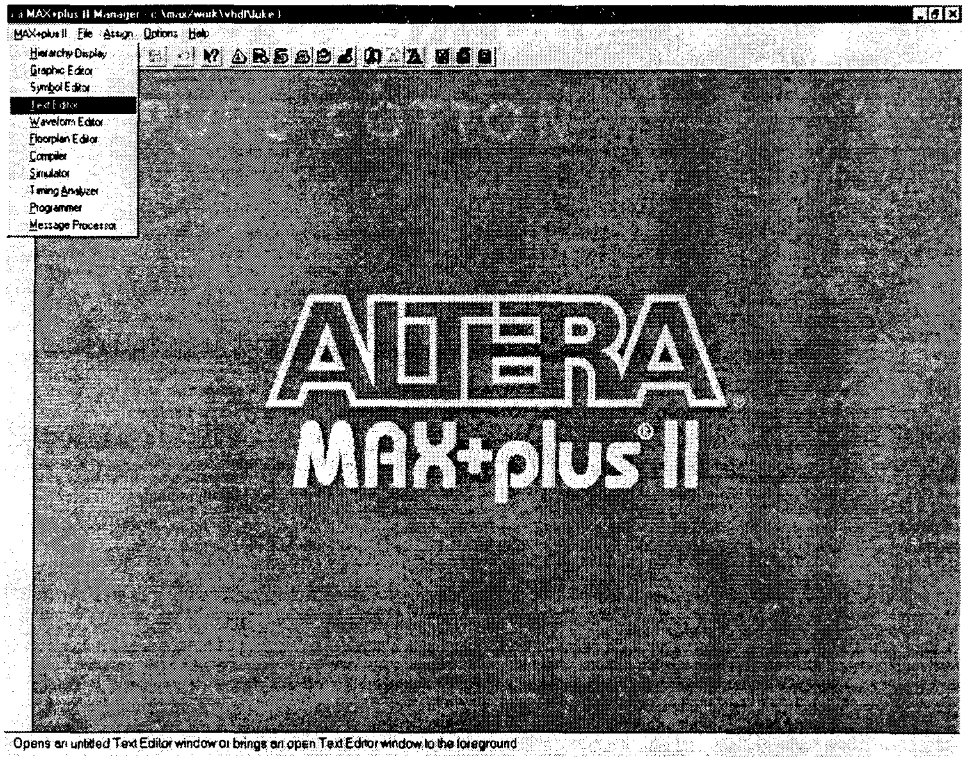
THE CPLD APPROACH USING ALTERA PRODUCTS

8.1 Introduction

A detailed description of how to use the MAX+PLUS® II software distributed by the Altera Corporation along with the UP1 Educational Board in order to implement the slot machine design is presented in this chapter. In order to accomplish this, VHDL was used as the design entry method of choice. The process in which VHDL translates into an actual circuit for this design is through use of the MAX+PLUS® II software design approach as described in Chapter 4, which consists of four main steps: design entry, design processing, design verification, and device programming. Since all of these areas require quite an extensive discussion, they will be examined within this chapter as separate sections. Along with this discussion, the steps of design implementation are outlined as specifically as possible so they could be followed like a tutorial. Most of the information in this chapter is based upon the references [14], [15] and [31], thus the interested reader should consult these sources for a more detailed discussion about any of the topics contained within.

8.2 The Design Entry Stage

When using the MAX+PLUS® II software, there are a variety of options available to the user for design implementation. For this particular project, the goal was to implement the design using VHDL, thus making the text editor the key utility needed for entry. The text editor allows not only VHDL designs to be entered but also Verilog HDL and AHDL coded ones as well. While it is also possible to write the code in other word-processing editors, this particular one is useful since it provides color-coding for reserved keywords, basic templates, and troubleshooting advantages. The text editor is invoked from MAX+PLUS® II's main screen, which is shown in Figure 8.1. As can be seen, the shortcut to launch the editor is found under the menu titled "MAX+plus II".



Opens an untitled Text Editor window or brings an open Text Editor window to the foreground

Figure 8.1 – The main MAX+PLUS® II screen.

Once the mouse is used to click on this shortcut, the text editor is executed and the screen shown in Figure 8.2 appears. This figure shows that there are new menu selections in the text editor that are not available on the main screen. One of these features can be found by selecting the "Utilities" menu on the top of the screen. From this menu, the aforementioned templates for the AHDL, VHDL and Verilog languages are found, which provide the basic format for structures ranging from case statements to architecture bodies to a full counter design. This allows the novice programmer to get a better feel for the language format as well as allow the advanced user to skip some monotonous coding.

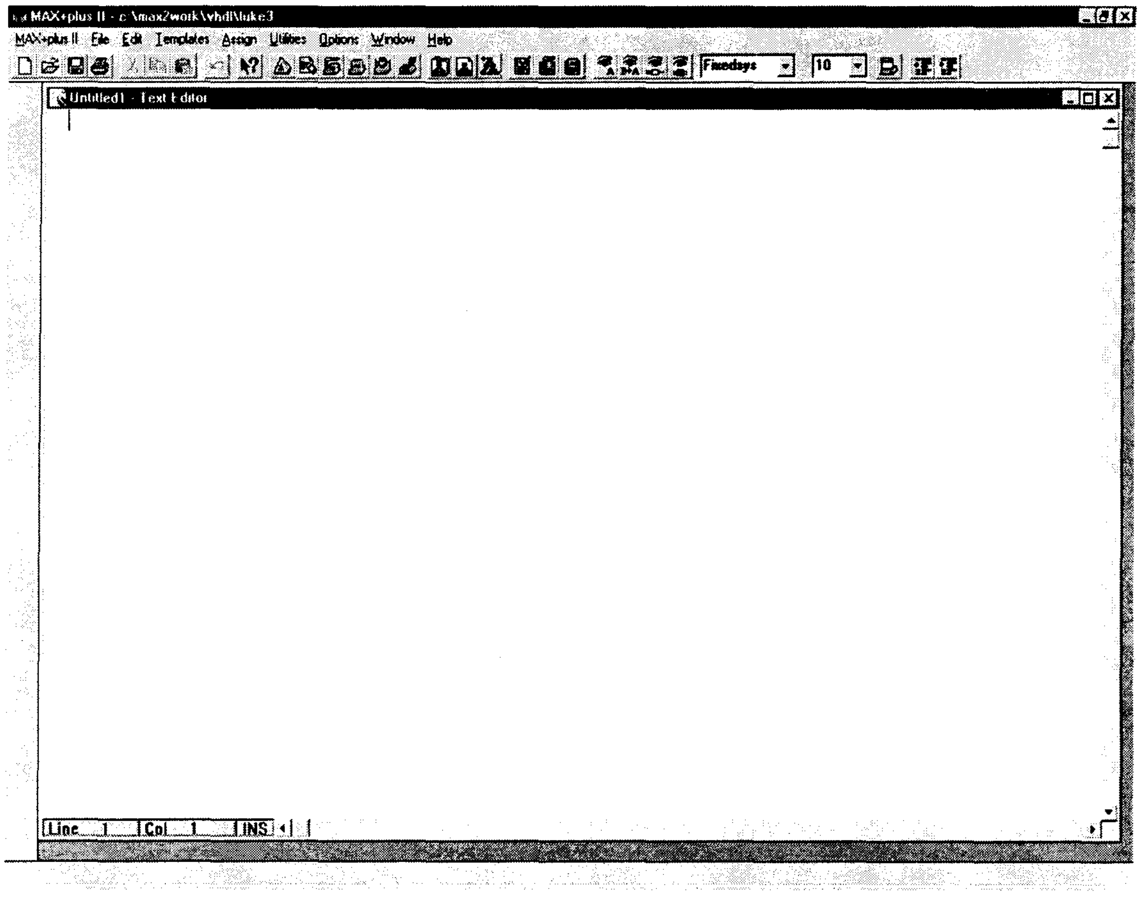


Figure 8.2 – The main MAX+PLUS® II text editor screen.

The text editor also contains utilities that can be used to manually assign pin numbers, hardware devices, and timing requirements to the design. These features can appropriately be found under the "Assign" menu. Also, there are attributes in this editor that are common to most others. A "find text" feature can be found on the "Utilities" menu, the font type, size and the ability to switch on and off syntax coloring is under the "Options" menu, and there is a help section containing information on the text editor in addition to VHDL, AHDL, and Verilog advice.

Now that the basics of the text editor have been discussed, the VHDL code being inserted into it for the slot machine design will be examined. This code is displayed as Figure 8.3 on Pages 70-74 of this report. Through observation of the code, it can be seen that there is a single entity and architecture declaration like any typical VHDL program. The entity reveals that there are 'START' and 'GO' inputs in addition to three clock ones ('CLK', 'CLK2', and 'CLK3'). The clocks inputs will be set to three different frequencies for the design and are used as the drivers for the random number generator. The 'GO' input is used solely as a stability feature to eliminate glitches and will be connected so that it is always set to a '1' logic level. Lastly, the 'START' input is used to simulate the user inserting a new credit just as the 'BEGIN' input was used in the Chapter 7 design. Once 'START' is asserted, one output will be selected from each of the three reels.

The entity also contains the declaration for three output columns in the design. They are labeled as 'COLUMN1', 'COLUMN2', and 'COLUMN3' and are declared so they contain nine elements in the array for each column (which are used to represent each color or blank). The entity is then terminated with the end command. It is important to note that the entity name in the MAX+PLUS® II software must match the filename it is contained within or else it will not compile!

The second main part of the VHDL program is the architecture. Judging by the length of this particular architecture, the reader would assume it is complex. This is not the case, however, since the same string of code is copied over three times for each of the reels with only slight modifications. The first section in the architecture is the declaration of signals and constants. For each of symbols that are present on the reels, there is a corresponding constant associated with it. This constant would set the logic value to '1' for the light that is being declared, while the rest on that reel remain at '0'. Also, three signals are declared for each of the random numbers being generated during execution.

After completion of the declarations, a process statement is used to signify a logic circuit written in a behavioral style. This statement requires that all statements following the 'begin' keyword be performed in a sequential fashion as opposed to the typical concurrent execution.

Each of the three reels is then setup in the same manner with a different clock frequency associated to each of them. A block of 'IF' statements are first used to increment the random number associated with each reel during each clock pulse's rising edge when the 'START' input is not asserted. A nested 'IF' statement then checks to see if the total has reached fifteen, and if so, it resets the value back to one. This creates a sequential counter of integers taking on values from one to fifteen. This sequence continues to execute until the 'START' input becomes asserted. At this time, the current random value of each reel is checked upon a series of conditions within a case statement.

Each of the fifteen possible values has been assigned a corresponding color output as outlined by Figure 5.4. While the 'START' input continues to be asserted, one symbol from each reel is asserted as well to signify the selected combination for that particular turn. Once the user changes the 'START' input to zero (which would represent the end of one game play in a real setting), this entire process begins again. The procedure will be more clear in Section 8.4, which displays the simulation results.

Once any VHDL code is completed within the text editor, it should be saved using a ".vhd" extension to signify it is written in VHDL code. Also, the file should be set to the current project as this will be necessary for future steps. This is done by selecting FILE >> PROJECT >> SET PROJECT TO CURRENT FILE.

The last step of the data entry portion of the design process is to check the program for basic errors. This is accomplished by selecting FILE >> PROJECT SAVE & CHECK or by pressing CTRL+K as a shortcut. This saves the current file and invokes the compiler window. The compiler's extractor module then checks the file for errors and outputs a message stating the total numbers of errors and warnings found.

```
-- Slot Machine Design written in VHDL
library ieee;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

entity luke3 is
       port(CLK, CLK2, CLK3, START, GO: in STD_LOGIC;
                 COLUMN1: out STD_LOGIC_VECTOR(8 downto 0);
                 COLUMN2: out STD_LOGIC_VECTOR(8 downto 0);
                 COLUMN3: out STD_LOGIC_VECTOR(8 downto 0));
end luke3;

architecture SLOT_MACHINE of luke3 is
constant RED: STD_LOGIC_VECTOR(1 to 9) := "100000000";
```

Figure 8.3 – The slot machine design written in VHDL code.

```
constant WHITE1: STD_LOGIC_VECTOR(1 to 9) := "010000000";
constant BLUE: STD_LOGIC_VECTOR(1 to 9) := "001000000";
constant WHITE2: STD_LOGIC_VECTOR(1 to 9) := "000100000";
constant GREEN: STD_LOGIC_VECTOR(1 to 9) := "000010000";
constant WHITE3: STD_LOGIC_VECTOR(1 to 9) := "000001000";
constant ORANGE: STD_LOGIC_VECTOR(1 to 9) := "000000100";
constant WHITE4: STD_LOGIC_VECTOR(1 to 9) := "000000010";
constant PURPLE: STD_LOGIC_VECTOR(1 to 9) := "000000001";
signal RAND1: UNSIGNED (3 downto 0);
signal RAND2: UNSIGNED (3 downto 0);
signal RAND3: UNSIGNED (3 downto 0);
begin
  process(CLK, CLK2, CLK3, RAND1, GO)
  begin
    if CLK'event and CLK='1' then
          if START/='0' then
          elsif (GO='1') and (RAND1=15) then RAND1 <= ('0','0','0','1');
          elsif (GO='1') then RAND1 <= RAND1 + 1;
          end if;
    end if;
    case RAND1 is
     when "0001" => if (START='1') and (GO='1') then COLUMN1 <= BLUE;
                else COLUMN1 <= "000000000";
                end if;
     when "0010" => if (START='1') and (GO='1') then COLUMN1 <= ORANGE;
                else COLUMN1 <= "000000000";
                end if;
     when "0011" => if (START='1') and (GO='1') then COLUMN1 <= WHITE1;
                else COLUMN1 <= "000000000";
                end if;
     when "0100" => if (START='1') and (GO='1') then COLUMN1 <= RED;
                else COLUMN1 <= "000000000";
                end if;
     when "0101" => if (START='1') and (GO='1') then COLUMN1 <= ORANGE;
                else COLUMN1 <= "000000000";
                end if;
     when "0110" => if (START='1') and (GO='1') then COLUMN1 <= GREEN;
                else COLUMN1 <= "000000000";
                end if;
     when "0111" => if (START='1') and (GO='1') then COLUMN1 <= WHITE3;
                else COLUMN1 <= "000000000";
                end if;
     when "1000" => if (START='1') and (GO='1') then COLUMN1 <= GREEN;
                else COLUMN1 <= "000000000";
                end if;
     when "1001" => if (START='1') and (GO='1') then COLUMN1 <= PURPLE;
                else COLUMN1 <= "000000000";
                end if;
     when "1010" => if (START='1') and (GO='1') then COLUMN1 <= WHITE2;
                else COLUMN1 <= "000000000";
                end if;
     when "1011" => if (START='1') and (GO='1') then COLUMN1 <= ORANGE;
                else COLUMN1 <= "000000000";
```

Figure 8.3 (cont.) – The slot machine design written in VHDL code.

```
            end if;
 when "1100" => if (START='1') and (GO='1') then COLUMN1 <= BLUE;
            else COLUMN1 <= "000000000";
            end if;
 when "1101" => if (START='1') and (GO='1') then COLUMN1 <= GREEN;
            else COLUMN1 <= "000000000";
            end if;
 when "1110" => if (START='1') and (GO='1') then COLUMN1 <= WHITE4;
            else COLUMN1 <= "000000000";
            end if;
 when "1111" => if (START='1') and (GO='1') then COLUMN1 <= PURPLE;
            else COLUMN1 <= "000000000";
            end if;
 when others => COLUMN1 <= "000000000";
end case;

if CLK2'event and CLK2='1' then
      if START/='0' then
      elsif (GO='1') and (RAND2=15) then RAND2 <= ('0','0','0','1');
      elsif (GO='1') then RAND2 <= RAND2 + 1;
      end if;
end if;
case RAND2 is
 when "0001" => if (START='1') and (GO='1') then COLUMN2 <= ORANGE;
            else COLUMN2 <= "000000000";
            end if;
 when "0010" => if (START='1') and (GO='1') then COLUMN2 <= GREEN;
            else COLUMN2 <= "000000000";
            end if;
 when "0011" => if (START='1') and (GO='1') then COLUMN2 <= WHITE2;
            else COLUMN2 <= "000000000";
            end if;
 when "0100" => if (START='1') and (GO='1') then COLUMN2 <= PURPLE;
            else COLUMN2 <= "000000000";
            end if;
 when "0101" => if (START='1') and (GO='1') then COLUMN2 <= BLUE;
            else COLUMN2 <= "000000000";
            end if;
 when "0110" => if (START='1') and (GO='1') then COLUMN2 <= GREEN;
            else COLUMN2 <= "000000000";
            end if;
 when "0111" => if (START='1') and (GO='1') then COLUMN2 <= RED;
            else COLUMN2 <= "000000000";
            end if;
 when "1000" => if (START='1') and (GO='1') then COLUMN2 <= PURPLE;
            else COLUMN2 <= "000000000";
            end if;
 when "1001" => if (START='1') and (GO='1') then COLUMN2 <= WHITE1;
            else COLUMN2 <= "000000000";
            end if;
 when "1010" => if (START='1') and (GO='1') then COLUMN2 <= ORANGE;
            else COLUMN2 <= "000000000";
            end if;
 when "1011" => if (START='1') and (GO='1') then COLUMN2 <= PURPLE;
```

Figure 8.3 (cont.) – The slot machine design written in VHDL code.

```
           else COLUMN2 <= "000000000";
           end if;
 when "1100" => if (START='1') and (GO='1') then COLUMN2 <= WHITE3;
           else COLUMN2 <= "000000000";
           end if;
 when "1101" => if (START='1') and (GO='1') then COLUMN2 <= ORANGE;
           else COLUMN2 <= "000000000";
           end if;
 when "1110" => if (START='1') and (GO='1') then COLUMN2 <= PURPLE;
           else COLUMN2 <= "000000000";
           end if;
 when "1111" => if (START='1') and (GO='1') then COLUMN2 <= WHITE4;
           else COLUMN2 <= "000000000";
           end if;
 when others => COLUMN2 <= "000000000";
end case;

if CLK3'event and CLK3='1' then
      if START/='0' then
      elsif (GO='1') and (RAND3=15) then RAND3 <= ('0','0','0','1');
      elsif (GO='1') then RAND3 <= RAND3 + 1;
      end if;
end if;
case RAND3 is
 when "0001" => if (START='1') and (GO='1') then COLUMN3 <= PURPLE;
           else COLUMN3 <= "000000000";
           end if;
 when "0010" => if (START='1') and (GO='1') then COLUMN3 <= RED;
           else COLUMN3 <= "000000000";
           end if;
 when "0011" => if (START='1') and (GO='1') then COLUMN3 <= WHITE1;
           else COLUMN3 <= "000000000";
           end if;
 when "0100" => if (START='1') and (GO='1') then COLUMN3 <= BLUE;
           else COLUMN3 <= "000000000";
           end if;
 when "0101" => if (START='1') and (GO='1') then COLUMN3 <= PURPLE;
           else COLUMN3 <= "000000000";
           end if;
 when "0110" => if (START='1') and (GO='1') then COLUMN3 <= WHITE1;
           else COLUMN3 <= "000000000";
           end if;
 when "0111" => if (START='1') and (GO='1') then COLUMN3 <= PURPLE;
           else COLUMN3 <= "000000000";
           end if;
 when "1000" => if (START='1') and (GO='1') then COLUMN3 <= WHITE2;
           else COLUMN3 <= "000000000";
           end if;
 when "1001" => if (START='1') and (GO='1') then COLUMN3 <= WHITE1;
           else COLUMN3 <= "000000000";
           end if;
 when "1010" => if (START='1') and (GO='1') then COLUMN3 <= GREEN;
           else COLUMN3 <= "000000000";
```

Figure 8.3 (cont.) – The slot machine design written in VHDL code.

```
                  end if;
        when "1011" => if (START='1') and (GO='1') then COLUMN3 <= PURPLE;
                  else COLUMN3 <= "000000000";
                  end if;
        when "1100" => if (START='1') and (GO='1') then COLUMN3 <= WHITE2;
                  else COLUMN3 <= "000000000";
                  end if;
        when "1101" => if (START='1') and (GO='1') then COLUMN3 <= ORANGE;
                  else COLUMN3 <= "000000000";
                  end if;
        when "1110" => if (START='1') and (GO='1') then COLUMN3 <= WHITE4;
                  else COLUMN3 <= "000000000";
                  end if;
        when "1111" => if (START='1') and (GO='1') then COLUMN3 <= WHITE3;
                  else COLUMN3 <= "000000000";
                  end if;
                        when others => COLUMN3 <= "000000000";
    end case;
  end process;
end SLOT_MACHINE;
```

Figure 8.3 (cont.) – The slot machine design written in VHDL code.

8.3 The Design Processing Stage

The design processor within the software is designed to process projects for most of Altera's Hardware including the UP1 educational board devices. The main feature of the processor in the MAX+PLUS® II software is the compiler. The compiler is a utility that performs many critical functions for future stages of design implementation. Some of these functions include error-checking capabilities, timing analyses, and device fitting. It is the link that deciphers the information input by the user into computer-friendly data used in the verification and programming stages.

In order to invoke the compiler, it should be selected from the MAX+PLUS® II menu on the main screen. Once the compiler has been selected, the screen shown as Figure 8.4 (on the following page) will appear. In order to properly run the compiler, however, a device must first be specified so that the design can be properly routed for the desired hardware. The device is specified by selecting ASSIGN >> DEVICE from the menu. A dialog box appears and from the MAX7000 device family, EPM7128SLC84-7 should be chosen when targeting the CPLD on the UP1 educational board. It is important to note that the box next to "Show Only Fastest Speed Grades" on the "Assign Device" dialog box should be unchecked or else the EPM7128SLC84-7 device will not be shown! Once the device is specified, the compilation process is ready to begin. In order to start it, the user must simply click the "Start" button depicted in Figure 8.4. While the compiler is running, the hourglass will begin to empty and flip and a tracking bar indicating the completion percentage at the bottom of the screen will grow. If any errors occur during the process, Altera's message processor window will open and display what kind of problem was found. This feature will be discussed more in depth later in this chapter.

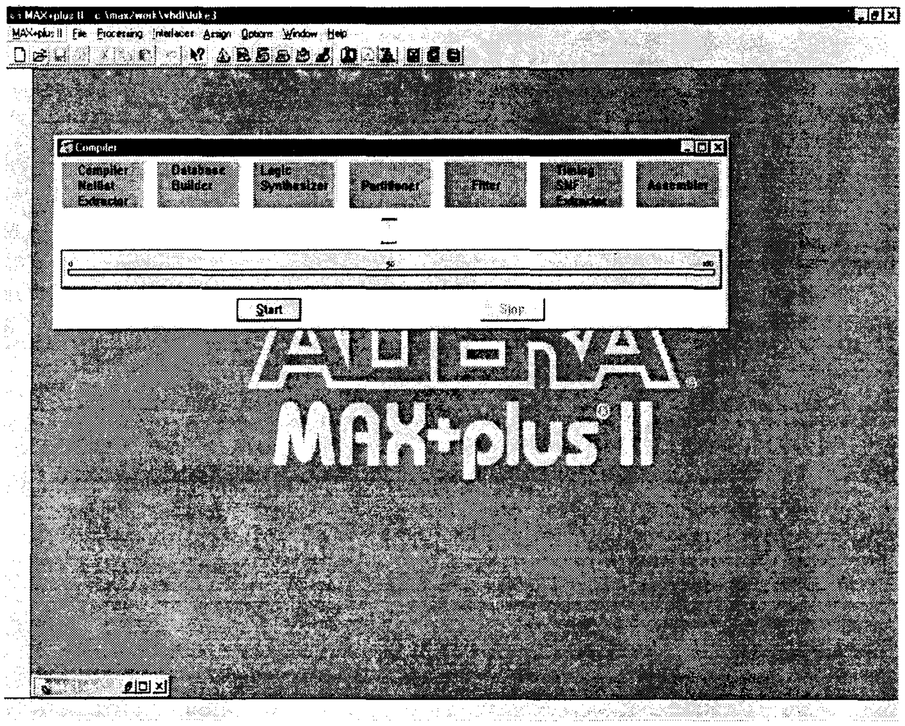


Figure 8.4 – The MAX+PLUS® II compiler screen.

Another event that occurs during the compilation process is that the modules inside the rectangular boxes seen in Figure 8.4 are darkened as they are completed. Each of these modules represents a component of the compilation process. These modules will be summarized briefly to enlighten the reader on all that is occurring "behind the scenes" during this stage of the design process.

The first module the compiler uses is the compiler net extractor. The responsibility of this module is to convert all of the design entry files associated with the project into a single binary netlist file. Also, for projects containing more than one file, this module records how these files and all nodes presented in the design are interrelated.

Using the information generated above, the next module (database builder) is able to construct a single database containing all of the necessary information for the rest of the compilation process. While building the database, this module also checks for connectivity, consistency, and errors throughout the design. This database serves as a base for constant updates as the compiler moves along to the following modules.

Once this database has been created, the logic synthesizer accesses the information and attempts to minimize the necessary resources for the device specified. It also removes information for any unconnected nodes found within the design to maximize efficiency.

Once in a while, despite the fact that the logic synthesizer acts to minimize the number of resources necessary on the design, it is still not possible to fit everything on the single device specified. It is then the job of the partitioner to divide the database in a manner that the smallest number of devices possible is required for implementation.

After leaving the partitioner module, the updated database enters the fitter. The fitter uses the project information from the database in addition to the identified resources available from the specified device in order to best implement the design. The fitter then assigns details of the design such as the logic and I/O cells and specific pin numbers to use. This information is then stored in a report file, which can be viewed by double clicking on the rectangular object titled ".rpt" underneath the fitter block after the compilation is completed. The report file can be quite lengthy such as the case for this particular circuit, which can be found in Appendix E.

The next module in line for the compilation process is the timing simulator netlist file (SNF) extractor. This module's job is to create a netlist file, which is used to hold the timing information for the project. This file is important during the simulation stage of the design process since this information is often needed for functional and timing analyses.

Lastly, the process reaches the assembler module. Where the timing SNF extractor generated crucial information for simulation, the assembler produces critical data for the programming stage. It creates a plethora of different programming images to be used in conjunction with the MAX+PLUS® II software and the Altera hardware. Figure 8.5 below displays the screen shown upon a successful compilation.

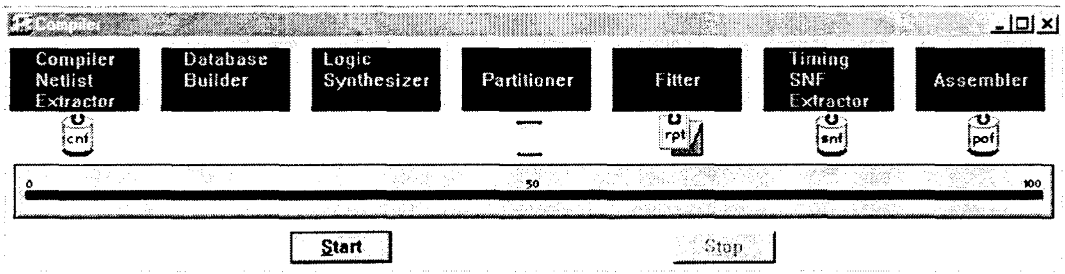
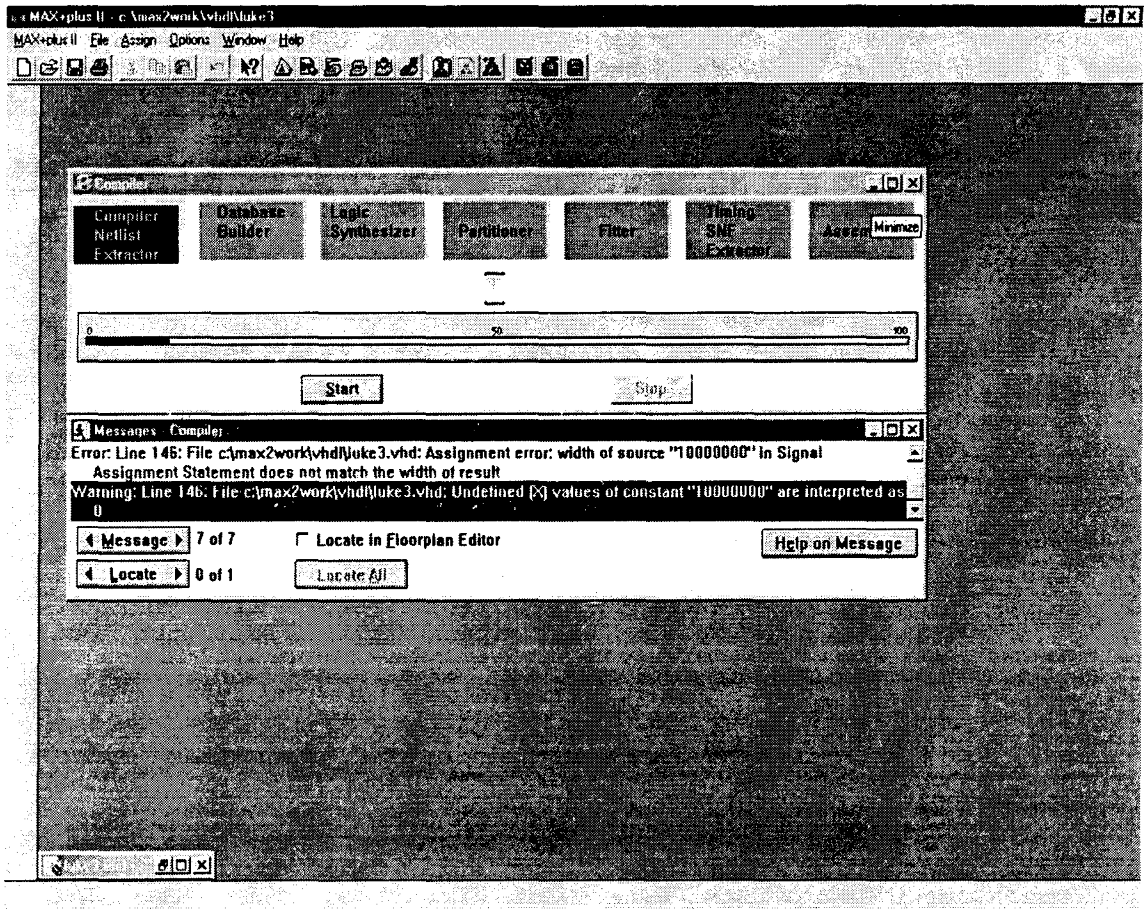


Figure 8.5 – The MAX+PLUS® II screen after a successful compilation.

Despite the fact that a design appears to be correct before attempting to compile it for the first time; there inevitably seems to be at least one error. With this fact in mind, Altera created a message processor to be used in conjunction with the compiler. When a compilation fails for any reason, this processor automatically opens up and gives pertinent information to best help the user troubleshoot the problem. Figure 8.6 shows what the screen looks like during an unsuccessful compilation.

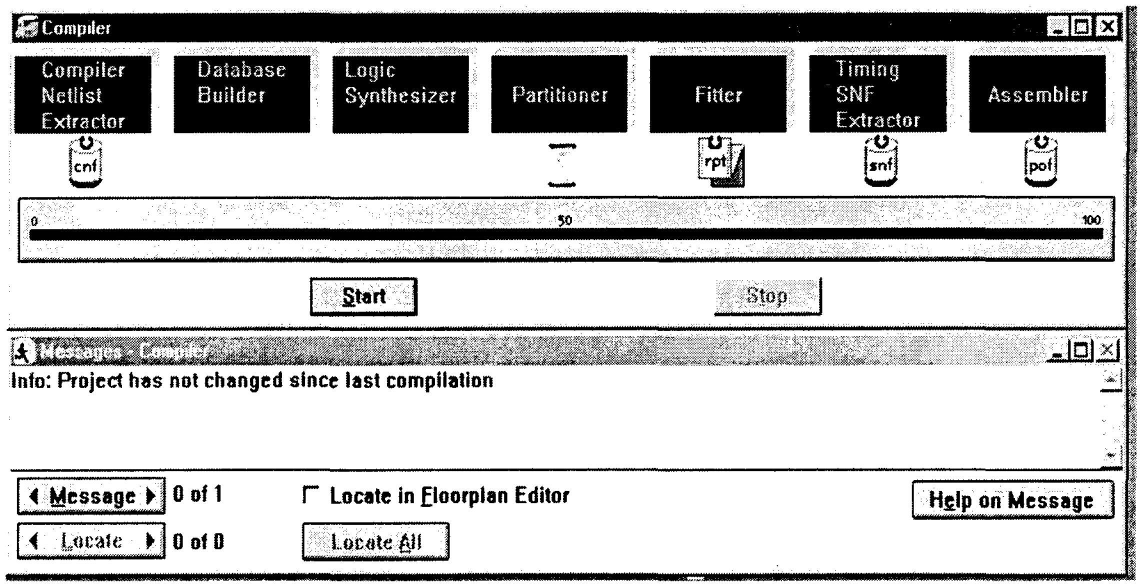


```
Error: Line 146: File c:\max2work\vhdl\luke3.vhd: Assignment error: width of source "10000000" in Signal
   Assignment Statement does not match the width of result
Warning: Line 146: File c:\max2work\vhdl\luke3.vhd: Undefined [X] values of constant "10000000" are interpreted as
   0
```

Figure 8.6 – The MAX+PLUS® II screen after an unsuccessful compilation.

As seen from the figure, the progress bar on the compiler screen does not fully reach 100% and error (written in red font) and warning messages (written in blue font) appear inside of the message processor. For VHDL simulations, these messages detail the line number, the file, and the reason for the error. By pressing the locate button below the processor, the user is automatically taken to the location in the code for the error currently highlighted. In addition to these helpful items, Altera also provides a listing of almost all possible errors and warning messages in the help menu and suggestions to troubleshoot them. This menu can be reached easily by clicking on the help on message button as seen in Figure 8.6.

There are also two other utilities that can be selected during the compilation stage that were found to be very helpful. The first of these is the smart recompile command. After a full compilation, this utility determines which of the modules are needed for subsequent compilations based on changes since the previous one. The advantage of this command is that it can greatly reduce the time required for compilation; especially for large projects. If there were no changes since the last time the compiler was initiated, a screen identical to Figure 8.7 will appear.



Info: Project has not changed since last compilation

Figure 8.7 – The MAX+PLUS® II compiler with the smart recompile command on.

The second useful utility available from the processing menu is the design doctor. This utility, when turned on, checks the design files in the project for consistency on a system-level since these are not easily caught in simulation. The checks performed are user-selected ones through the specification of a set of design rules.

8.4 The Design Verification Stage

Now that the design has been entered and translated into a computer recognizable format, it is time to verify it before implementation. In order to do this, Altera provides a simulator application to be used in conjunction with the waveform editor and timing analyzer.

In order to successfully run the simulator, a ".scf" file must first be created with the waveform editor. Both the simulator and waveform editor can be selected from the MAX+PLUS® II menu on the main screen. When opening the waveform editor, a screen appears as shown in Figure 8.8 on the following page.

On the screen shown in Figure 8.8, various signals can be added that represent inputs, outputs, or internal signals found in the design. In order to enter these signals, the user should select NODE >> ENTER NODES FROM SNF. Once this has done, the dialogue box seen in Figure 8.9 on Page 82 appears. The "List" button located in the upper right hand corner of the dialogue box allows all of the nodes available in the design to be viewed. These nodes will appear selected in the "Available Nodes & Groups" textbox. They can then be selected to use on the waveform editor by pressing the arrow pointing to the right and then OK.

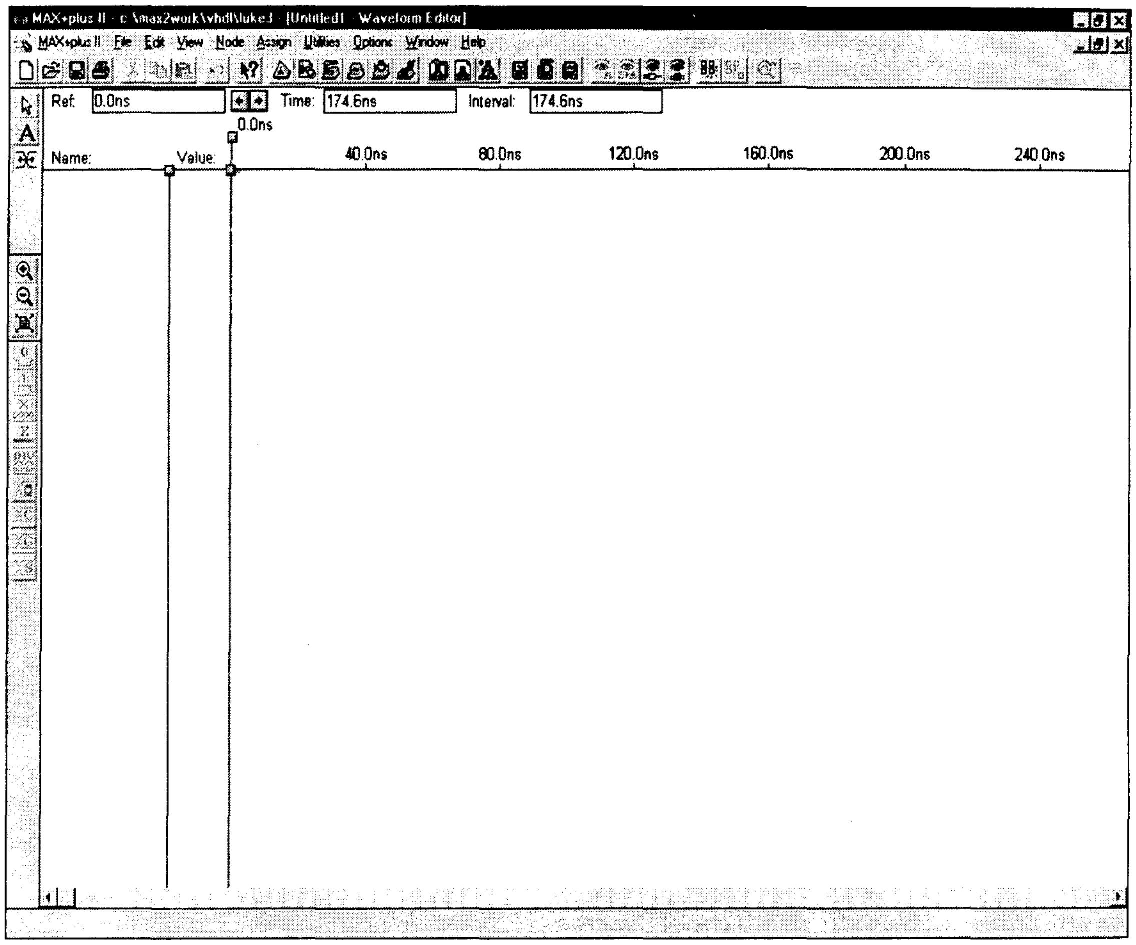

MAX+plus II - c:\max2work\vhdl\luke3 - [Untitled1 - Waveform Editor]
MAX+plus II File Edit View Node Assign Utilities Options Window Help
Ref: 0.0ns
Time: 174.6ns
Interval: 174.6ns
0.0ns
Name:
Value:
40.0ns
80.0ns
120.0ns
160.0ns
200.0ns
240.0ns

Figure 8.8 – The waveform editor's main screen.

Once the desired nodes have been selected, they are denoted on the waveform editor screen with either an 'I' for input, 'O' for output, or 'B' for a buried or internal node. The initial default value is also shown, which is a logic low '0' for inputs and an indeterminate 'X' for outputs and buried nodes.

Now that the desired nodes have been entered, it is time to modify them so that they can be used for simulation. There is no need to change anything on the outputs or buried nodes since the simulator will update them accordingly depending on the input values.

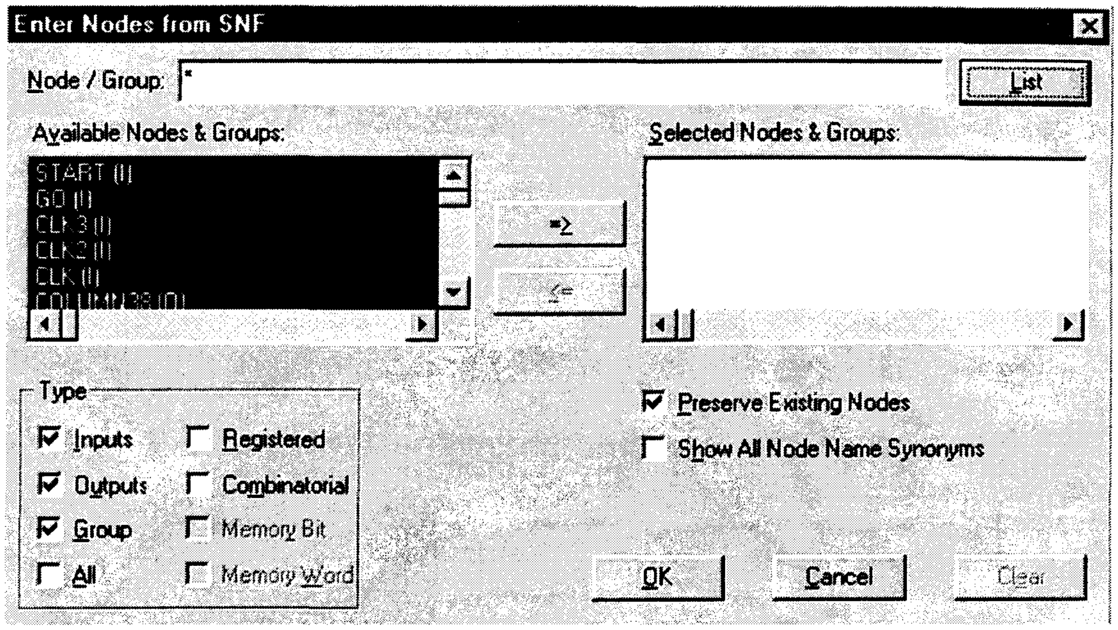
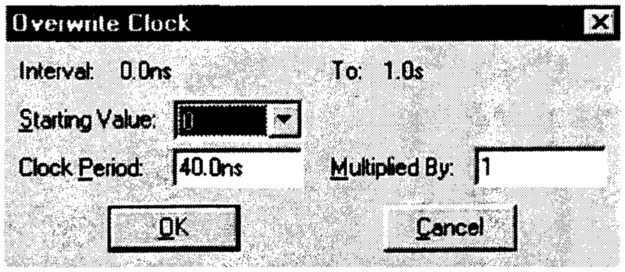


Figure 8.9 – The "Enter Nodes from SNF" dialogue box.

The first step to editing a signal starts with selecting the waveform. This is done by a single mouse click on the desired waveform. Once this is performed, the toolbar on the left side of Figure 8.8 is used to edit the input waveforms. There are four buttons present on this toolbar, which can create a constant waveform throughout the duration of simulation. These buttons are labeled '0', '1', 'X' and 'Z' and represent constant low, high, undefined, and hi-impedance values respectively. These buttons are valuable tools especially when determining outputs for one particular set of inputs, however it is often necessary to simulate clocks or rapidly changing inputs. For this reason, the MAX+PLUS® II software provides functions that can either specify a clock or counter sequence. The button with the alarm clock symbol represents the overwrite clock function. When selecting this function, the dialog box as shown in Figure 8.10 will appear. As can be seen in the figure, the starting value of the clock function can be specified as well as the clock period. One important note to keep in mind is that the clock period will not be able to be varied unless the "Snap to Grid" utility is turned off.

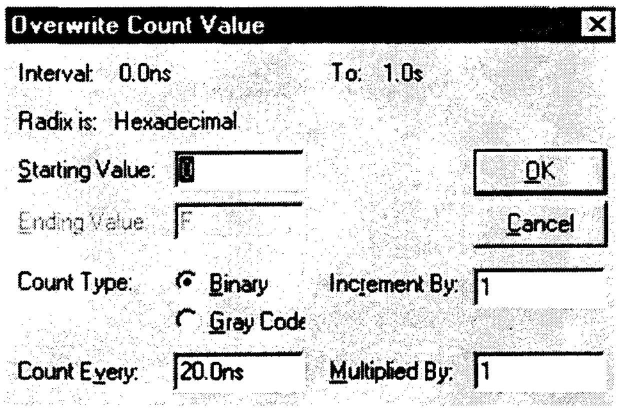


Interval: 0.0ns To: 1.0s
Starting Value: 0
Clock Period: 40.0ns Multiplied By: 1

Figure 8.10 – The "Overwrite Clock" Dialogue Box.

The other waveform editor button that allows changing values to be specified is called "overwrite count value" button. A snapshot of this dialogue box is shown in Figure 8.11. This particular function is very similar to the overwrite clock function but is simpler with multi-node waveforms. It allows the user to specify the starting and ending values of count sequences as well as the increment value, count type, and count interval.

The last tool for editing waveforms can be selected by pressing the button. This tool allows the user to manually edit the waveforms on the editor as deemed necessary.

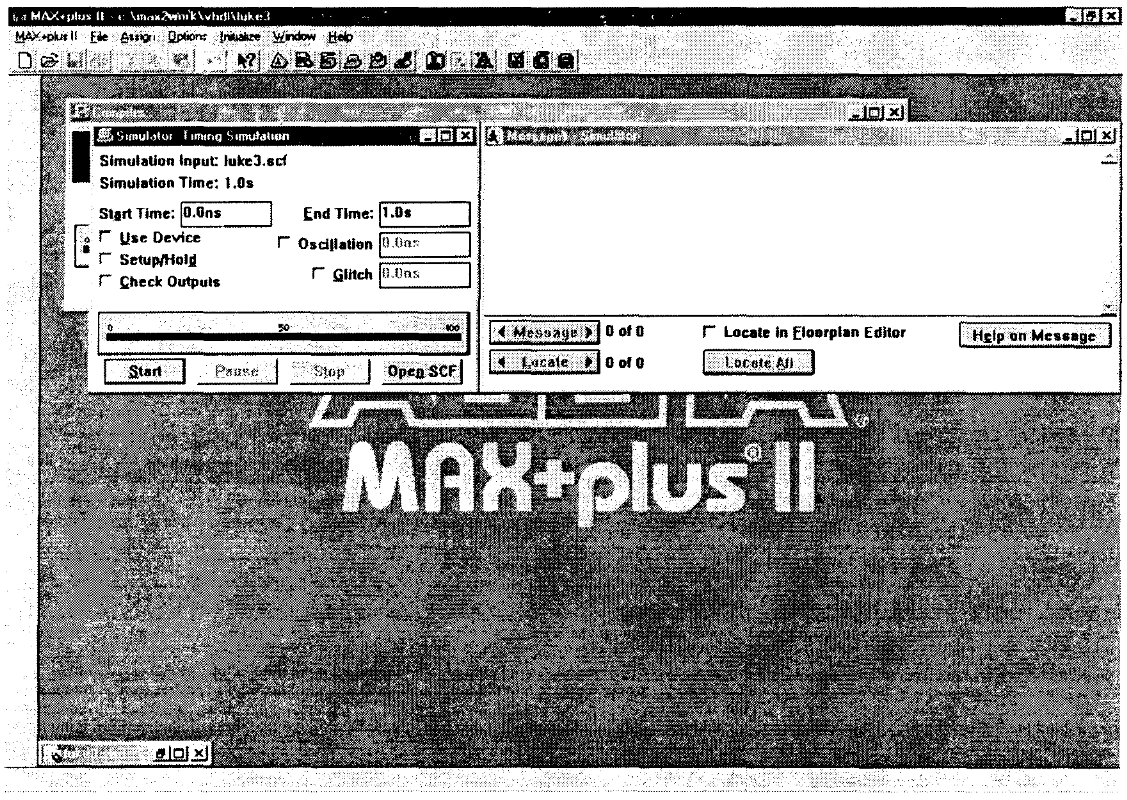


Interval: 0.0ns To: 1.0s
Radix is: Hexadecimal
Starting Value: 0
Ending Value: F
Count Type: [✓ Binary] Gray Code
Increment By: 1
Count Every: 20.0ns Multiplied By: 1

Figure 8.11 – The "Overwrite Count Value" Dialogue Box.

The last step in setting up the ".scf" file is to edit the simulation screen appropriately. The end time of the simulation can be specified by going to the FILE >> END TIME. Also, the grid size can be changed, turned on or off in addition to the snap to grid function from the 'Options' menu to best suit the user's intentions. The procedure of creating the most relevant ".scf" file is often a cyclical one since it is often necessary to go back and edit the waveforms appropriately when changing any of the grid or end time options.

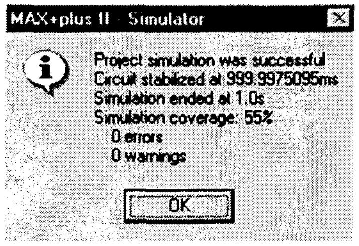
Once the ".scf" file is properly setup, it is possible to now run the simulation at this point. The first step of this process is to launch the simulator by selecting it from the "Max+PLUS II" menu on the main screen (Figure 8.1). A snapshot of the window generated by invoking the simulator is depicted in Figure 8.12.



Simulation Input: luke3.scf
Simulation Time: 1.0s
Start Time: 0.0ns
End Time: 1.0s
Use Device
Setup/Hold
Check Outputs
Oscillation 0.0ns
Glitch 0.0ns
Message 0 of 0
Locate 0 of 0
Locate in Floorplan Editor

Figure 8.12 – The MAX+PLUS® II simulator main screen.

From this screen, the simulator can be executed by pressing the "Start" button. If the waveform file is not visible, it can be opened by pressing the "Open SCF" button. Also, the starting and ending times of the simulation can be specified in the dialogue box. If a successful simulation occurs, a red bar will fill the progress bar and a message box will pop up on the screen similar to the one shown in Figure 8.13. This box displays the simulation's end time and the number of errors and warnings generated.



```
Project simulation was successful
Circuit stabilized at 999.9975095ms
Simulation ended at 1.0s
Simulation coverage: 55%
  0 errors
  0 warnings
```

Figure 8.13 – The message box for a successful simulation.

For the slot machine design, two pertinent simulation waveform graphs are provided on the following two pages. The first simulation shown in Figure 8.14 displays each of the most important signals included in this design. The total time for these signals were scaled down to one second to save on compilation time and resources, yet, it still allows the functionality of the design to be conveyed. The input 'GO' should always be set to a logic high value when the circuit is working properly. The 'CLK', 'CLK2', and 'CLK3' inputs are connected to clocks of different frequencies to drive the three random number generators. The other input 'START' holds each random number being produced at the instant of assertion until the value changes back to a logic low value. The buried signals 'RAND1', 'RAND2', and 'RAND3' are the actual random numbers being generated. It can be seen from Figure 8.14 that these values hold their value once 'START' is asserted just as they were designed to do.

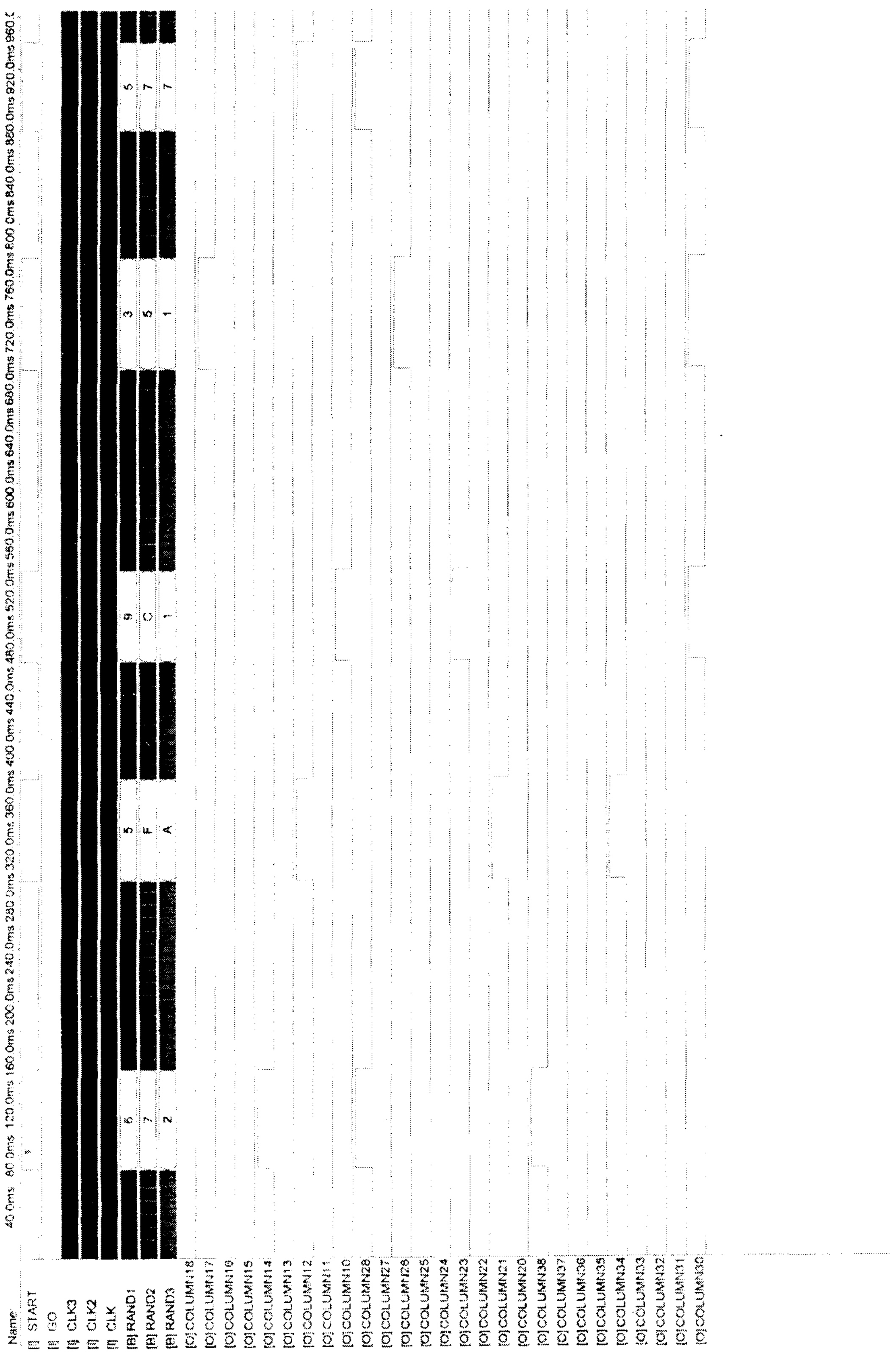


Figure 8.14 – The full simulation results for the digital slot machine design.

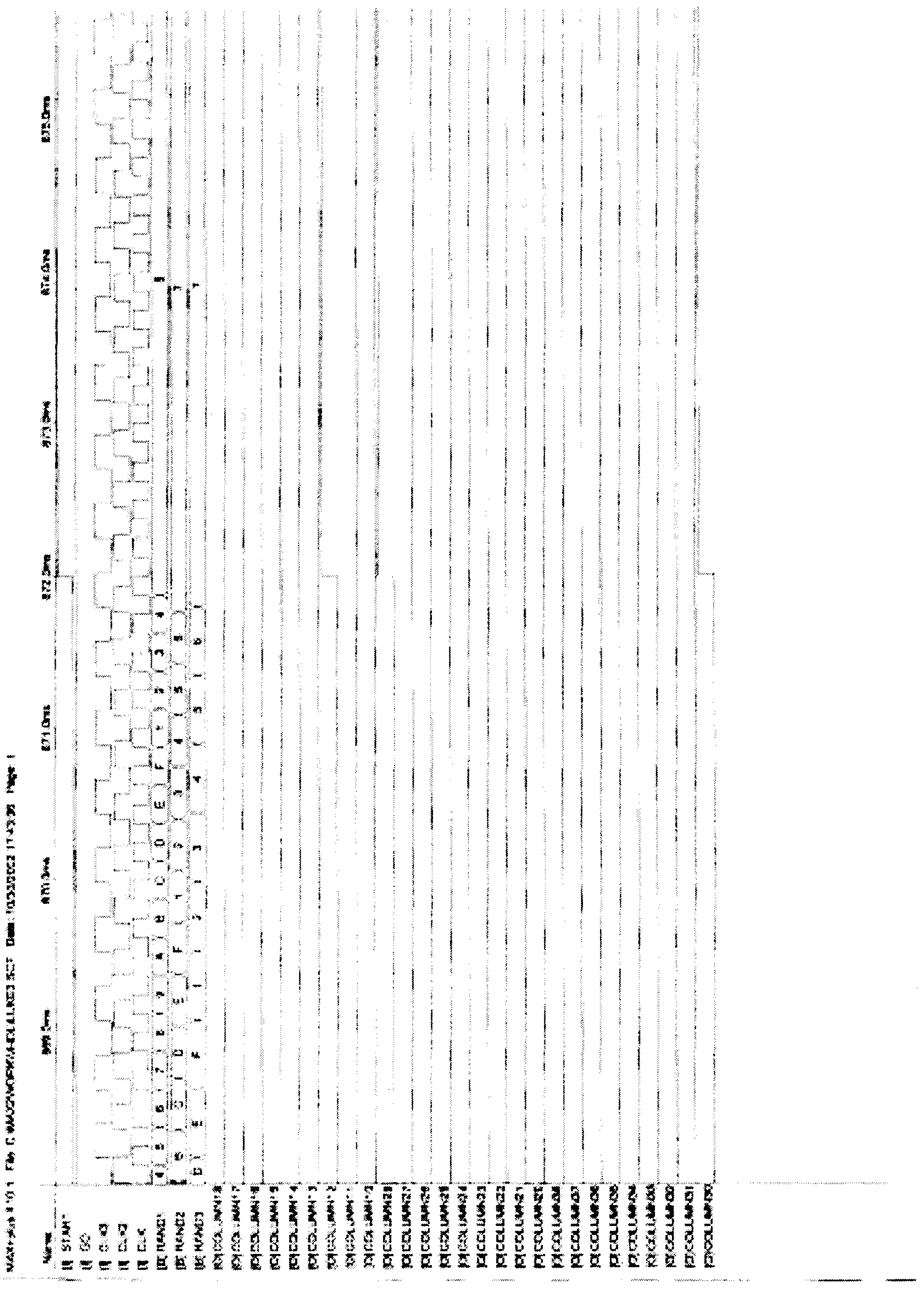


Figure 8.15 – The simulation results over an 8ms time period.

The method by which the outputs are labeled are by applying the word COLUMN followed by the column number and a symbol number displaying where it would appear on the slot machine output. Figure 8.14 shows that one and only one value is selected for each column when 'START' is asserted. Figure 8.15 depicts a portion of the simulation shown in Figure 8.14 over a much smaller time period so that the actual counting sequences of the random numbers can be viewed.

Once the functional simulation part of the verification process has been completed, it is time to perform a timing analysis. Some timing results such as glitches can be found by simply viewing the ".scf" file after simulation. However, the actual values of the timing information can be found by using the timing analyzer in the MAX+PLUS® II software. This utility can be invoked by selecting the timing analyzer from the under the MAX+PLUS® II menu on the main screen (Figure 8.1).

Within this application, there are three typical analysis modes that can be performed and are found under the "Analysis" menu upon launching. The first of these and the most common is the delay matrix. This matrix allows the user to specify a series of input and output nodes and then the corresponding delay time between each is calculated. The main screen for the delay matrix is displayed in Figure 8.16. In order to specify which nodes to use during the analysis, the user should select NODE >> TIMING ANALYSIS SOURCE to specify the desired input nodes and NODE >> TIMING ANALYSIS DESTINATION for the outputs. Once this is performed, the output nodes are shown on the top of the matrix whereas the inputs will be seen on the left side of it. The last step of the process is to press the 'Start' button on the delay matrix screen and the delay values will then appear.

The results of the delay matrix for this project can be found in Appendix F. The propagation times for the clocks to reach the outputs were calculated to be 9.5ns, 9.5ns, and 12.5ns for the symbols on reels one, two, and three respectively. The important fact is that these times are small enough to be non-visible to the user as well as consistent for all of the symbols on each reel.

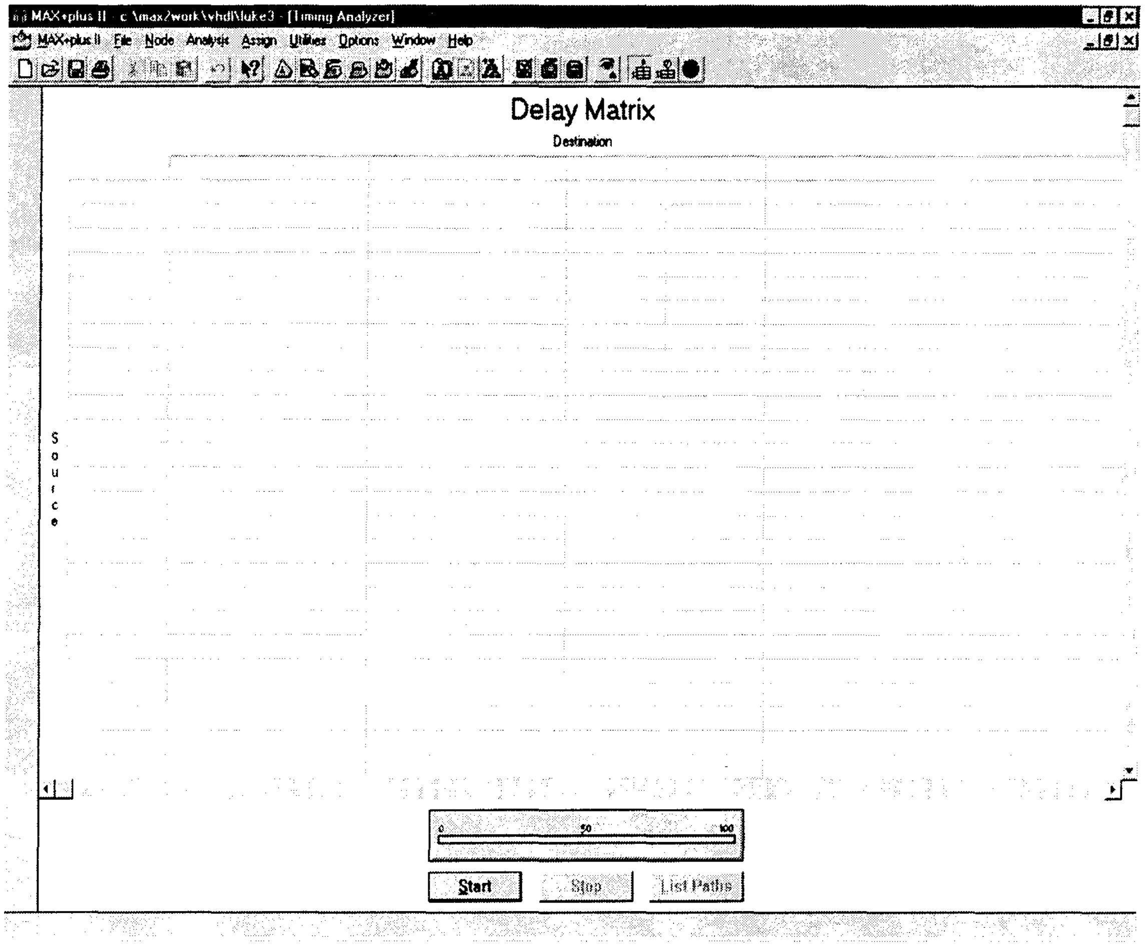

MAX+plus II - c:\max2work\vhdl\luke3 - [Timing Analyzer]
MAX+plus II File Node Analysis Assign Utilities Options Window Help
Delay Matrix
Destination
Source
0
50
100
Start
Stop
List Paths

Figure 8.16 – The delay matrix screen for the timing analyzer.

Another tool that can be selected in the timing analyzer is the registered performance mode. This mode "analyzes registered logic for a performance-limiting delay, minimum clock period, and maximum circuit frequency". Also, a Setup/Hold

Matrix mode is available for use for circuits containing flip-flops and latches. It does exactly what its name states and calculates "the minimum setup and hold requirements from input pins to signal inputs" [15]. These modes were not as pertinent to this design like the delay matrix was so the generated results were omitted from this report.

One useful feature about the timing analyzer is that it allows the user the select a specific node (either source or destination) and view any and all delay paths that are attributed to it. Also, the Message Processor can be used in accordance with the Timing Analyzer by selecting the "List Paths" from the delay matrix screen shown in Figure 8.16 once the analysis has taken place. This opens the message window and displays all paths for a specific node.

8.5 The Design Programming Stage

The last stage of the design process involves the actual programming of the hardware device. In order to transfer the appropriate information from the software to the hardware, the device must first be setup properly. On the UP1 educational board, there are four sets of jumpers that can be set to either program the CPLD, the FPGA, both, or a string of multiple boards together. To configure the jumpers properly, one should consult the user guide [14] that is packaged with the board. For this particular application, the goal was to solely program the CPLD so the jumpers were placed on the upper two positions in each of the columns.

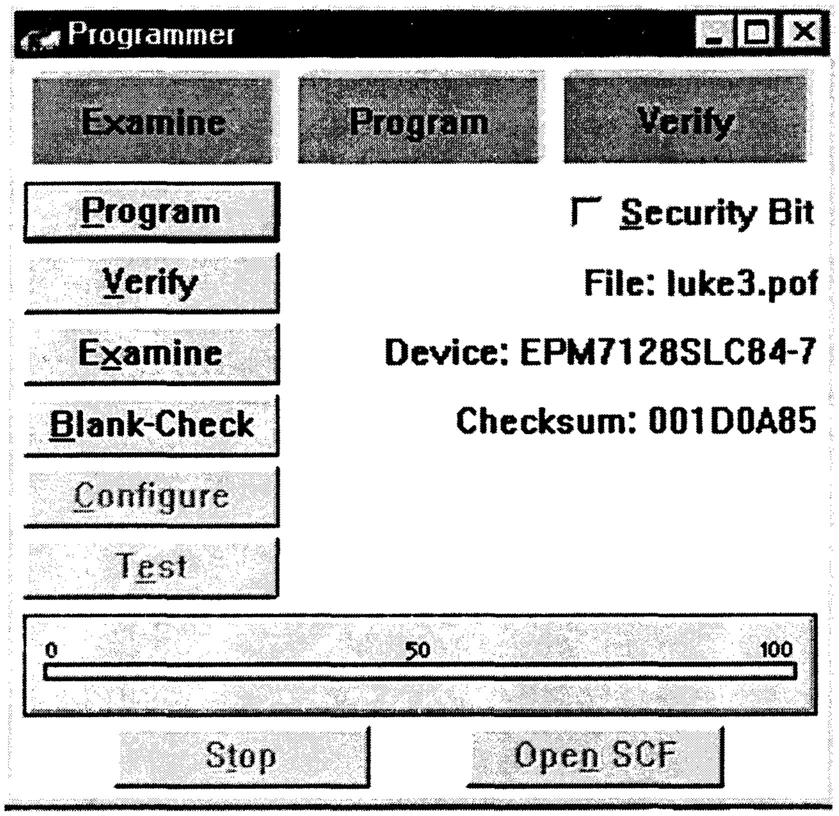
Another requirement for of the UP1 educational board setup is to connect a seven to nine Volt DC power supply with a minimum of three hundred fifty mA to the DC_IN power input. The ideal source is a nine Volt DC power supply with close to one Amp of current. For this particular experiment, a nine Volt, eight hundred mA DC power supply purchased from Radio Shack was used and converts its power from a regular household power outlet. Once proper power is supplied to the board, the 'Power' LED should be lit to indicate it is functioning properly.

The last step with the hardware setup is to connect it to the parallel port of a computer running MAX+PLUS® II with the proper software key. This is done through use of the ByteBlasterMV™ download cable. It connects between the parallel port to the 'JTAG_IN' ten pin male header on the UP1 education board. The board must be powered up to transfer information since the ByteBlasterMV™ receives its power and ground from the board itself. This concludes the necessary adjustments to be performed on the actual hardware.

As for the software aspect of the programming process, the MAX+PLUS® II software provides an application that is launched from the MAX+PLUS® II main screen. The screen for the programmer application is shown in Figure 8.16. It uses files that were previously generated by the compiler and allows the user to do the multitude of activities displayed on the buttons in Figure 8.16. Before performing any of these activities though, it is important to make sure that the file and device are the desired ones for the project. The name of the file should be the name of the project with a '.pof' extension and the CPLD for the educational board is the EPM7128SLC84-7.

Since the CPLD on the education boards is EEPROM based, it is not necessary to run a blank check because the device does not have to be blank to program over the existing information. In order to change the file name, select FILE >>

PROGRAMMING FILE from the menu on the programmer screen. The device can be changed by selecting the device option by going to ASSIGN >> SELECTING THE DEVICE.



File: luke3.pof

Device: EPM7128SLC84-7

Checksum: 001D0A85

Figure 8.16 – The Altera MAX+PLUS® II programmer screen.

The last task is then to simply press the "Program" button from the programmer screen. If no problems arise, the progress bar will fill to completion and no errors or warnings will appear in the message processor window. To test that the data contained within the current '.pof' file is the same that is now on the hardware device, the "Verify" button can be selected from the programmer menu. The device can now be unconnected from the ByteBlasterMV^{TM} download cable and be used in the desired application. Since the EPM7128SLC84-7 CPLD is a non-volatile hardware device, the power supply can also be unconnected and moved without the chip losing the contents.

CHAPTER IX

CONCLUSION

9.1 – Summary

The development of programmable logic devices began in the mid 1970's out of a need to produce a logic device that could not only be designed but also built by the same company. As the need for larger capabilities continued to grow, so did the complexity of the architectures of these devices. The most commonly used devices today are integrated circuits in the very large scale integration category such as complex programmable logic devices (CPLDs) and field programmable gate arrays (FPGAs).

Since the architecture of these devices is so complex that it is virtually impossible to list all the single connections needed to implement a design, computer aided design tools and hardware description languages were developed. One of the established manufacturers of programmable logic hardware and software products is the Altera Corporation. One of the most common software packages they produce is the MAX+PLUS® II software, which provides a fully integrated environment for programmable logic design. This software allows the designer to use this package for the entire design process from entry to implementation.

The multitude of entry methods supported with this software helps provide a friendly environment for designers of various skill levels. The advent of the hardware description languages provides the designer with the ability to input the specifics and behavior of the desired circuit into a text editor to allow the software to generate the design logistics that formerly had to be worked out manually. This allows the designer to make slight modifications to a design in a quick and efficient manner.

The author has provided two designs to contrast the previous design methodology with the one currently used in industry today. The circuit using VHDL demonstrates the advantages of the newer design processes, which includes faster development time, less hardware components, and ease of modification.

9.2 – Project Results

The intent of this design was to provide an educational resource from which future students could learn the basics of programmable logic and the design process involved. In order to accomplish this, the history for many aspects of the topic was given as well as a thorough discussion of the design thought process. Also, a series of tutorials is given to aide the novice programmer to easily learn VHDL. Furthermore, the design implementation is described in a step-by-step fashion so that any interested reader can easily follow the steps to develop his or her own design.

While providing this research, some ideas were generated for improvement on the design tools. One of these was to make the VHDL text editor in Altera more user friendly. It seems as if the company is trying to impose their version, AHDL, upon the users by providing additional features. VHDL is such a complex language to learn that all attempts at providing a friendly environment (such as Visual Basic's) should be made. Also, it would be nice if the schematic editor provided a list of components to choose from, as does MicroSim's PSpice and Xilinx's Foundation Series. These programs allow the user to see what components can be used for a schematic without having to reference the help menu in addition to drag and drop capabilities. Lastly, the hardware provided with Altera's educational package is not very conducive to a lab environment. The pin numbers are poorly labeled on both the hardware and manual, the connections for wire connections are very tight, and the components seem to be very fragile. The author's experience in undergraduate laboratory leads him to believe that these problems will lead to the need for numerous replacements.

9.3 – Future Ideas for Research

The extensive research on this particular topic also provided a variety of additional interesting ideas that were beyond the scope of this project. Most of them arose from the software packages and hardware devices that keep rapidly evolving and changing.

In addition to the Altera Corporation, the Xilinx Company is also another major programmable logic company. They provide hardware and software packages that are very similar to that of their competitor. An interesting study would be to produce a unique design in both of the environment and compare and contrast the two methodologies. Currently, Youngstown State University has the resources available from both companies to be able to do this type of comparison.

In a similar fashion, the different hardware description languages could be compared through developing a design implementation in each of them. The Altera

MAX+PLUS® II software provides the capability of creating a design using AHDL, VHDL, and Verilog HDL.

Finally, the design created within this report utilized only the CPLD on the Altera UP1 educational board. However, there are plenty of other features available on this hardware including the "FLEX" device. An interesting study would be to program a design on both devices and discuss the timing and performance characteristics and differences between the two.

REFERENCES

[1] "Altera University Program." *Altera Corporation*. 8 August 2002. <http://www.altera.com/education/univ/unv-index.html>.

[2] Wakerly, John F. Digital Design Principles & Practices. 3rd ed. updated. Upper Saddle River, NJ: Prentice Hall, 2001.

[3] Armstrong, James R. Chip-Level Modeling with VHDL. Englewood Cliffs, NJ: Prentice Hall, 1989.

[4] Pellerin, David and Michael Holley. Practical Design Using Programmable Logic. Englewood Cliffs, NJ: Prentice Hall, 1991.

[5] "The First Integrated Circuits." *Maxfield & Montrose Interactive Inc.* 28 May 2002. <http://www.maxmon.com/1952ad.htm>.

[6] Alcorn, R. B. "A Digital Circuit Design Implementation Using ABEL-HDL and Programmable Logic Devices." Master's Thesis, Department of Electrical Engineering, Youngstown State University, 1997.

[7] "FPGA, EPLD, CPLD, PLD Suppliers." *OptiMagic Inc.* 20 May 2002. <http://www.optimagic.com/companies.html>.

[8] "FPGA, EPLD, CPLD, PLD Suppliers Summary Table." *OptiMagic Inc.* 20 May 2002. <http://www.optimagic.com/summary.html>.

[9] "Programmable Devices/Technology." *Arbeitsgruppe CAD*. 12 April 2002. <http://argon.iaee.tuwien.ac.at/lehre/fhwn/prog_logic_devices.pdf>.

[10] "Frequently Asked Questions About Programmable Logic, FPGAs, and CPLDs." *OptiMagic Inc.* 20 May 2002. <http://www.optimagic.com/faq.html>.

[11] "Field-Programmable Devices." *EDN Magazine*. 28 May 2002. <http://archives.e-insite.net/archives/ednmag/reg/1996/101096/21df_07.html>.

[12] "Programmable Logic Devices." *Imperial College of Science, Technology & Medicine*. 17 June 2002. <http://www.ee.ic.ac.uk/pcheung/teaching/ee3_DSD/DSD3small.pdf>.

[13] Sandige, Richard S. Digital Design Essentials. Upper Saddle River, NJ: Prentice Hall, 2002.

[14] Altera Corporation. University Program Design Laboratory Package User Guide. Version 2.0. October 2001.

[15] Altera Corporation. MAX+PLUS II Programmable Logic Development System & Software Data Sheet. Version 8. January 1998.

[16] "Altera Corporation: The Programmable Solutions Company." *Altera Corporation*. 10 May 2002. <http://www.altera.com>.

[17] Bhasker J. A VHDL Primer. 3rd ed. Upper Saddle River, NJ: Prentice Hall, 1999.

[18] "History of VHDL." *MicroLab – Swiss Microelectronics Laboratory*. 14 January 2002. <http://www.microlab.ch/academics/courses/vlsi/vhdl-ieee/TUTORIAL/MOD1/SEC2/HTML/SLIDE4.HTM>.

[19] "Basic VHDL." *RASSP E&F*. 14 January 2002. <http://www.people.vcu.edu/~rhklenke/tutorials/vhdl/modules/m10_23/sld001.htm>.

[20] Miller, Scott. "History of VHDL." *The Duct Tape Guide to VHDL*. 14 January 2002. <http://phoenix.nmt.edu/~hllywood/history.html>.

[21] Eaton, Deran S. "The VHDL Solution." *Naval Surface Warfare Center*. 14 January 2002. <http://www.nswc.navy.mil/cosip/feb98/vi0298-2.shtml>.

[22] "A Brief History of VHDL." *Doulos Limited*. 14 January 2002. <http://www.doulos.co.uk/fi/desguidevhdl/vb2_history.htm>.

[23] Chang K.C. "Digital Design and Modeling with VHDL and Synthesis – Preface." *Institute of Electrical and Electronics Engineers, Inc*. 14 January 2002. <http://www.computer.org/cspress/catalog/bp07716/chapt.html>.

[24] "Case Study – Verilog vs VHDL." *University of California, Berkeley*. 14 January 2002. <http://www-inst.eecs.berkeley.edu/~eecsba1/s97/reports/eecsba1h/verilog.html>.

[25] "Glossary of EDA Terms." *IKOS Verification Innovation*. 21 March 2002. <http://www.ikos.com/investors/glossary>.

[26] "Slot Machines – Their history, how they work & a couple of tips." *Real Entertainment LTD*. 17 January 2001. <http://www.casino-info.com/slots.htm>.

[27] "Odds and Strategy for Slot Machines." *The Wizard of Odds*. 17 January 2001. <http://www.thewizardofodds.com/game/slot.html>.

[28] Bourie, Steve. "American Casino Guide." *Slot Machines.* 17 January 2001. <http://www.americancasinoguide.com/Tips/Slots.shtml>.

[29] Boelhouwer, Steve. "Playing for Keeps: Developing Casino Games." *CMP Media LLC.* 2001 February 5. <http://www.gamasutra.com/features/20000424/gambling_01.htm>.

[30] Strictly Slots – The Magazine for Slot & Video Poker Players. *ACE Marketing Inc.* Volume 4 No.2. February 2002.

[31] Dueck, Robert K. Digital Design with CPLD Applications and VHDL: A Lab Manual. Albany, NY: Delmar Publishers, 2001

[32] "Xilinx: Programmable Logic Devices, FPGA & CPLD" *Xilinx Company.* 20 July 2002. <http://www.xilinx.com>.

[33] "Altera" *Georgia Tech University.* 1 December 2002. <http://users.ece.gatech.edu/~hamblen/ALTERA/altera.htm>.

APPENDIX A

VHDL LESSONS FOR STUDENT USE

A.1 - Lesson 1: Introduction & Examples of VHDL

What is VHDL and what does it stand for? VHDL stands for Very high-speed integrated circuits Hardware Description Language with the abbreviation coming from the initials of the capitalized words. As the title states, it is a descriptive language that is used to model digital systems at any level from a simple gate to a massive digital electronic system. The language provides the capability of building systems with or without timing constraints as well as providing the means of simulating concurrent and sequential logic statements.

The two main parts of any design simulated in VHDL are:

- **Entity** – The entity is used to describe the external interfaces (usually the inputs and outputs) of a digital logic system. *Port* statements are then used inside the entity to describe the internal and external terminals and their directions.
- **Architecture** - The architecture is used to describe the internal functionality of how the system (and entity) operates.

The generic syntax of a basic entity is:

```
entity entity name is
        [ port ( list of interface port names and their types ) ; ]
end entity name ;
```

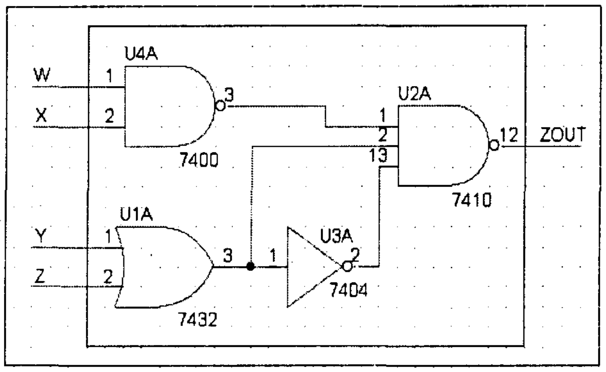
There are other parts of an entity as well but these are the main concepts to know to begin learning the basics. For a more complete description of entities consult the world wide web or your textbook. Please note in VHDL examples that the text in **BOLD** are VHDL keywords and the text in *italics* are user-defined variables, types, etc.

As can be seen, the main parts of an entity are:

1. **Entity name** – The name of the entire entity. It must follow the rules as outlined in the identity discussion.
2. **Port statement** – The ports are defined as the signals through which the entity interfaces with other objects in the outside environment. They are often the input & output signals of the entire system.
3. **Port name** – The names of each individual port. They must follow the rules as outlined in the variable discussion.

4. **Port mode** – There are four modes that a port can be defined as that are pertinent to us:
   a. **in:** The value is only able to be read within the model.
   b. **out:** The value can only be updated within the model.
   c. **inout:** The value can be read AND updated within the model.
   d. **buffer:** The value can be read AND updated as well but cannot have greater than one source and can only be connected to other buffers or signals with one source.
5. **Port type** – The set of values that the port can hold. Common types are INTEGER, BOOLEAN, BIT, and STD_ULOGIC. A discussion on types can be found later in this report.

An example of creating an entity is given below with Figure 1 and the corresponding code:

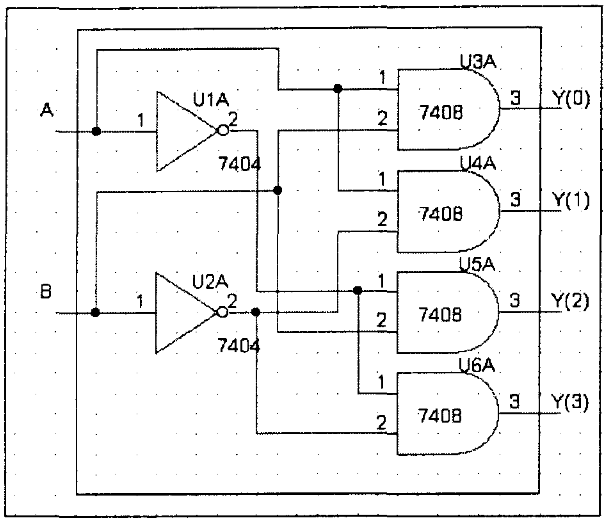


```
entity FIRST is
        port (W, X, Y, Z: in BIT;  ZOUT: out BIT);
end FIRST;
```

Figure 1: Example circuit for the creation of an entity titled "First" in the VHDL language

Figure 1 gives a good feel of how VHDL relates to the circuit implementation. The entity name in this example is 'FIRST', the port names are 'W', 'X', 'Y', 'Z', and 'ZOUT', the port modes are 'in' for the input signals and 'out' for the output signal, and the port types are 'BIT' which allows only the values of '0' and '1' to be passed along these signals. Notice how all of the commas, semicolons, and parenthesis are used in this example; syntax is very important in VHDL!

The signals can be named individually as was seen in Figure 1 or can be stored in a vector array. This notation is common in decoders, multiplexers, etc. In Figure 2, the outputs are defined in a 'Y' vector ranging from zero to three so that the following values are created: Y0, Y1, Y2, Y3. It should be noted that this array can be named anything and be any size; it is entirely up to the user.



```
entity ARRAY is
     port(A,B: in BIT; Y: out BIT_VECTOR(0 to 3));
end ARRAY;
```

Figure 2: Example circuit using the 'BIT_VECTOR' array type

Now that entity declarations have been discussed in depth, certain parts of the entity will be examined further. The first involves the naming of the ports, entities, and variables as well as future things such as architectures, functions, etc. These names are given by objects, which are called identifiers.

Identifiers are any names that are permissible for such objects that are described above. Within VHDL, there are two basic types of identifiers with their own set of rules.

1. **Basic Identifier** – A basic identifier is a combination of one or more characters that adhere to the following constraints.
   - The first character of the identifier must be a lower or uppercase letter.
   - All alphanumeric characters are allowed; this includes only uppercase and lowercase letters, digits, and the underscore character.
   - Two of the underscore characters cannot be used consecutively.
   - Basic identifiers are never case-sensitive.
   - Spaces cannot be present within basic identifiers.

- Keywords such as **begin, for, in, case, after, next** cannot be used as basic identifiers. Keywords in examples of this report are given in bold to make the reader familiar with which are reserved and which are not.
- ENGINEER, O1N1E1, UNDER_SCORE are examples of valid basic identifiers.
- 1_UNDER, WHAT!, TWO_ _SPACE are examples of invalid basic identifiers.

2. **Extended Identifier** – An extended identifier includes any combination of characters in between two backslashes that adheres to the rules of the basic identifier in addition to these constraints.
    - All of the characters that are valid in basic identifiers are also valid for extended identifiers. However, the . ! @ ‘ $ characters are valid as well.
    - Unlike basic identifiers, extended identifiers are case sensitive within the backslashes.
    - In order to represent a backslash within an extended identifier, two consecutive backslashes are used (in addition to the two backslashes denoting the start and end of the extended identifier.
    - \Extended\. \Money$$$\ are examples of valid extended identifiers

Lastly, comments can be provided within the VHDL code using two consecutive hyphens. These consecutive hyphens must be provided before every line of comment in the code. An example of a comment is shown in Figure 3 below.

```
entity FIRST is
        port (W, X, Y, Z: in BIT;  ZOUT: out BIT);
        --This is how to insert a comment into Figure 1
        --This is where a second line of commenting would go
end FIRST;
```

Figure 3 – An example of how to comment within a block of code.

A.2 - Lesson 2: Introduction to Architecture Bodies.

Introduction to Architecture Bodies:

As mentioned in Lesson 1, there are two main parts of any design simulated in VHDL code and are the entity and the architecture. Since the entity was discussed in the first lesson, the architecture will be further examined in this and future lessons. As also mentioned previously, the architecture is used to describe the internal functionality of how the system (and entity) operates.

The general syntax of a basic architecture body is:

```
architecture architecture-name of entity-name is
        signal declarations
        constant declarations
        variable declarations
        type declarations
        other declarations
begin
        concurrent statement 1
        concurrent statement 2
        .
        .
        concurrent statement n
end architecture-name;
```

As can be seen, the main parts of an architecture body are:

1. **Architecture name** – The name of the particular architecture, which must follow the rules laid out in the identifier section. It can be seen from the architecture declaration statement that it is linked to a particular entity as well.
2. **Declaration statements** – These statements will be further discussed in this lesson. These statements (as well as any entity declarations) declare items that can be used within the architecture body.
3. **Concurrent statements** – These parallel statements are used to describe the internal composition as outlined by the entity statements.

There are also 3 different popular styles that can be used to describe an architecture body

1. **Structural modeling** – An entity is described as a set of components that are connected by signals.
2. **Dataflow modeling** –Concurrent statements are used to describe the circuit by its operations and flow instead of explicitly defining the structure of the design.
3. **Behavioral modeling** – Uses processes to execute a collection of sequential statements.

Declarations:

What are declarations and what are they used for? Declaration statements are used to name and define the data type associated with a data object. Data objects are just memory spaces named by an identifier that are available to hold a certain value(s).

The three main type of data objects are:

1. **Signal** – As seen with entities in Lesson 1, signals are data objects that can hold a signal's present and future values. Within architectures, signals are most often used for intermediate steps within the circuitry.
2. **Constant** – Holds a single value of a specified type. A constant cannot be changed within execution of the program. Allows identifiable and clear name to be given to such things as universal constants (pi, e, etc.).
3. **Variable** – Similar to a constant in that it holds a single value of specified type. Unlike a constant, however, is that this value can be changed during execution. Also, unlike a signal, variables need not have a physical representation within the circuitry.

How are these data objects declared? Now that the data objects have been discussed, there needs to be a structure so as to formally name and associate a data type with each. The valid names have been discussed with identifiers and data types will be discussed later in this lesson. Figure 1 shows the generic syntax for declaring the above data objects

```
constant name : type := initial value;
variable name : type := initial value;
signal name : type := initial value;
```

```
constant MyAge : integer := 21;
variable TodaysDate : integer := 3185;
signal AandB : bit := 0;
```

Figure 1 – Generic Syntax and Examples for declaration statements of data objects.

What are data types? Data types should be somewhat familiar since they have been touched on in previous discussions but have yet to be covered in depth. A data type represents a set of values that a data object can hold. The flexibility of VHDL allows the user to define certain data types in addition to the ones that are pre-defined within the VHDL language or in popular library packages. A common variation of a data type is a subtype. Subtypes are just data types with some constraint. A subtype range is used to specify the constraint on the type and can be declared in ascending order using 'to' or descending order using 'downto' as seen in Figure 2.

```
type name is values;
subtype name is type-name range start-value to end-value;
subtype name is type-name range start-value downto end-value;
```

Figure 2 – Generic syntax and examples for declaration statements of data types.

There are 2 main categories of common data types:

1. **Scalar** – Values of this type are in sequential order.
2. **Composite** – Consists of many elements of same type (array) or elements that can be of different type (record). This data type will be discussed further in Lesson 3.

The 3 main scalar types belong to these categories:

1. **Enumeration** – defined type that lists all specified values; can be numbers, characters, statements, etc. Some common ones are listed below along with all of the values associated with the particular type.
   - **CHARACTER** – pre-defined type that uses the 191 characters of the ISO 8-bit coded character set (ASCII + some more). These characters are typically written as between single quotes.
   - **BIT** – pre-defined type that can take on either a '0' or '1' value
   - **BOOLEAN** – pre-defined type that can take on a FALSE or TRUE value
   - **STD_LOGIC** – popular type that is pre-defined in the IEEE-1164 package. This type can hold the following values 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'. See the text for explanations for each of these values.

2. **Integer** – defined type that specifies a set of integer values. These values at least include the range –(2^31-1) to (2^31-1) and sometimes more in different software packages. Scientific notation can also be used such as: 6E2 can be used to represent 600. Common subtypes of integers that the user must be define include natural (positive integers plus zero) and positive (positive integers).
   - **INTEGER** – pre-defined type that includes the integers from –(2^31-1) to (2^31-1).

3. **Float** – defined type that specifies a set of real numbers. This means that fractional numbers can be used in decimal notation. Numbers of different bases, such as binary, can also be represented using this notation: (*base#value#*).
   - **REAL** – pre-defined type that includes real numbers from –(10^38) to (10^38) and 6 digits of precision.

A.3 - Lesson 3: Arrays and Operators

Arrays:

What is an array? An array is one of the two categories of composite data types (the other being a record type). As opposed to the scalar data types discussed in the previous lesson, arrays are a set of values (instead of an individual value) of the same data type. A record type can include values of different data types but will not be discussed since it is not as commonly used. Arrays are most commonly 1-dimensional, which can be thought of as a list or a single column of values. They can also be 2 or more dimensional as well. A 2-dimensional array can be visualized as a table or a set of values with columns as well as rows.

Just as with scalar types, arrays are created once they are declared. Some common ways to declare arrays are shown in Figure 1 below.

```
type name is array (start-value to end-value) of element-data-type;
type name is array (start-value downto end-value) of element-data-type;
```

Figure 1 – Typical syntax used for 1-dimensional array declarations.

As can be seen in Figure 1, there are 3 parts to an array declaration:

1. **Array Name** – Creates name for the array using a valid identifier.
2. **Number of Items in List** – Uses a range of numbers to create the number of items present in the list. This order is created as specified in the declaration statement from start value to end value. These individual numbers of the array are often referred to as the index.
3. **Data Type of Elements** – Since in an array, the values must be of the same type, this data type is specified within the declaration statement here.

Some examples of array declarations are provided in Figure 2.

```
type fall_classes is array (0 to 4) of INTEGER;
type angle_radians is array (7 downto 0) of REAL;
```

Figure 2 – Examples of 1-dimensional array declarations.

There are 2 predefined 1-dimensional array types as well as 1 in the IEEE 1164 package:

- **STRING** – This predefined array is an array of characters.
- **BIT_VECTOR** – As previously used in lesson 1, this predefined array is one of bits.
- **STD_LOGIC_VECTOR** – This array is defined within the IEEE 1164 package and is an array consisting of all the valid **STD_LOGIC** values (as discussed in Lesson 2).

Once an array has been created by the declaration statement, the next step is to input values into each element. This can be done using literals. Literals are used to assign values to the entire array. There are two main types of literals: string literals and bit literals. String literals are sequences of characters while bit literals are sequences of bits. The values for each literal are assigned by putting them within double quotation marks. Bit literals can be assigned using octal and hex notation as well. This is done by putting an 'O' or an 'X', respectively, before the first quotation mark.

Examples of the assignment of pre-defined array data types to data objects for both string and bit literals are given below in Figure 3.

```
variable My_Name: STRING(1 to 4);
My_Name := "Luke";
```

```
variable FORTY: BIT_VECTOR(7 downto 0);
FORTY := "00101000";
```

Figure 3 – Examples of assigning array data types to data objects.

Individual elements or partial groups of elements can be accessed or assigned values as well. This is done by typing the name of the array followed by the index number(s) in parenthesis. Figure 4 shows how element number 3 of the variable My_Name from Figure 3 would be accessed as well as elements 3 to 6 of variable FORTY.

```
My_Name(3)
FORTY(6 downto 3)
```

Figure 4 – Example of accessing individual or partial groups of elements in arrays.

Lastly, values can then be assigned to these individual or groups of elements as well. This can be done as shown in Figure 5.

```
My_Name(3) := "k";
FORTY(6 downto 3) := (4 => '1', 6 => '1', others => '0');
```

Figure 5 – The assignment of values to individual or groups of elements in arrays.

The assignment to the elements of FORTY in Figure 5 should look somewhat new to the beginning VHDL student. This statement is just a different way to assign values to arrays. It is showing that element 4 is being assigned a value of '1' as is element number 6. The keyword **others** is used by VHDL to assign the specified value (in this case '0') to all the other elements (in this case, it would be elements 3 and 5).

<u>Operators:</u>

Operators in this discussion are broken into 4 categories:

1. **Logical**
2. **Comparative**
3. **Shift**
4. **Mathematical**

Logical operators are those that are commonly used in discrete mathematics as well as in digital design. These operators are defined as types BIT and BOOLEAN. A list of all the logical operators can be found in Figure 6.

and -	logical and function
nand -	logical nand function
or -	logical or function
nor -	logical nor function
xor -	logical xor function
xnor -	logical xnor function
not -	logical not function (inverter)

Figure 6 – The logical operators available in VHDL

Comparative operators are used to compare two elements (or arrays) against one another. The resulting data type is always BOOLEAN. A list of the comparative operators is given as Figure 7.

<	less than	>=	greater than or equal
<=	less than or equal	=	equal to
>	greater than	/=	not equal to

Figure 7 – The comparative operators available in VHDL

Shift operators are used to either shift or rotate arrays in the direction specified within the operator. The array is used on the left side of the operand while an integer is taken on the right side. The integer determines how many spaces to shift or rotate the array. Shift logical operators fill vacated spaces with '0', while shift arithmetic operators fill vacated bits with the rightmost bit of the left operand (for **sla**) or the leftmost bit of the left operand (for **sra**). Lastly, rotate operands fill the vacated bits as if they were in a ring. For instance, in an eight element array being rotated right three spaces, the seventh element from the left would become the second element. A list of the shift operators is given as Figure 8.

sll -	shift left logical	**sra** -	shift right arithmetic
sla -	shift left arithmetic	**rol** -	rotate left
srl -	shift right logical	**ror** -	rotate right

Figure 8 – The shift operators available in VHDL

Mathematical operators are used to perform typical mathematical functions. For addition, subtraction, multiplication and division functions, the operands must be of the same data type. The concatenation operator can be used to combine elements as well as arrays from left to right as specified. Also important to note is that the exponentiation operator must have an integer as the right operand. The mathematical operators are given as Figure 9. It is important to note that not all operators will be available in all software packages. For example, in the Altera software, the mod and div functions are not valid operators.

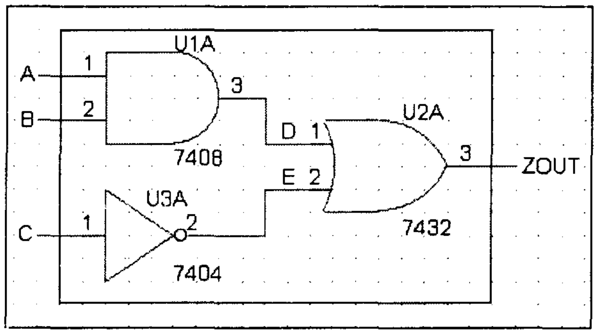
+	addition	**mod**	modulus
-	subtraction	**rem**	remainder
*	multiplication	**abs**	absolute value
/	division	**&**	concatenation
**	exponent		

Figure 9 – The mathematical operators available in VHDL

A.4 - Lesson 4: Structural Modeling

Now that many of the nuts and bolts of the VHDL programming language have been discussed, it is time to actually represent a total design. As discussed in Lesson 2, there are three basics styles of modeling used in VHDL and will be discussed further in the next three lessons. These styles are: Structural, Dataflow, and Behavioral.

What characterizes the structural style? The structural style involves using components to define the interconnection of signals declared in the entity. A structural design corresponds exactly to a schematic, since it lists all components (or gates) and all of the inputs and outputs to these components, and is perhaps the easiest style (but not usually the most efficient) to use for the programming beginner. It is important to know that all VHDL statements (except process statements) are executed concurrently, which means they are executing at the same time (in parallel). An example of an entire structural style model is given below in Figure 1.



```
entity FIRST_VHDL is
        port (A, B, C: in BIT; ZOUT: out BIT);
end FIRST_VHDL;
```

```
architecture FIRST_VHDL_ARCH of FIRST_VHDL is
        component AND2
                port (F, G: in BIT; H: out BIT);
        end component;
        component OR2
                port (I, J: in BIT; K: out BIT);
        end component;
        component INV
                port (L: in BIT; M: out BIT);
        end component;
        signal D, E: BIT;
begin
        A1: AND2 port map (A, B, D);
        A2: INV port map (C, E);
        B1: OR2 port map (D, E, ZOUT);
end FIRST_VHDL_ARCH;
```

Figure 1 – An example of a schematic and its corresponding structural style code.

As with any modeling style (and as seen in Figure 1), the structural architecture can be broken up into two sections:

1. **Component Declarations** – takes place before the **begin** keyword; components are declared here.
2. **Component Statements** – takes place after the **begin** keyword; associates the ports within the entity with signals within the architecture and performs the appropriate functions to describe the design.

Some points to consider about the component declarations:

- Components can either be pre-defined or user created. If they happen to be user created, they must use configuration statements to bind them.
- This statement is used to define the components that are used within the statement section of the architecture body.
- The port names, as seen in Figure 1, can be different that those used within the entity declaration.
- A component declaration declares the name, type, and mode of the component in a comparable fashion to entity declarations. The generic syntax of a component declaration can be seen in Figure 2.

```
component name
        port(signal-names :  signal-mode signal-type;
             signal-names :  signal-mode signal-type);
end component;
```

Figure 2 – Generic syntax for a component declaration in a structural architecture.

Some points to consider about the component statements:

- Uses the **port map** keyword to link the ports declared in the entity with signals present within the current architecture.
- These statements, also called component instantiation statements, can use the keyword **open** for any component part that is left unconnected.
- The generic syntax of a component statement can be seen in Figure 3. The label can be any valid identifier and should be an appropriate one to best describe the statement (for easy reading of the code). Also, the component name must be one that was previously declared in the declaration section of the architecture. The signals are then linked in the same order, type, and mode as was previously declared in the declaration statement for that component. It is important to make sure that the size of the signals used in the instantiation statements correspond exactly to that of those in the component declaration statements.

```
label: component-name port map (signal-names);
```

Figure 3 – Generic syntax of a component statement in a structural architecture.

There is also a shortcut for using more than one copy of a particular component within an architecture. This structure is called a generate statement and acts as a "for loop" on a single component instantiation statement. The generic syntax for a for-generate loop is given as Figure 4. Once again, as in the single line component statements, a label is given that can be any valid identifier. Another identifier is also used to name the range values. The generate loop statement is then executed once and the index number incremented until the end-value is reached. An example of using a for-generate statement is given below the general syntax in Figure 4. This example shows how a two-input "AND" gate can be repeated four times where the inputs are the vectors A(1)-A(4) & B(1)-B(4) and the outputs are the vectors D(1)-D(4). Please note that the vectors must be declared as this size in the component declaration statement as well or else an error will occur.

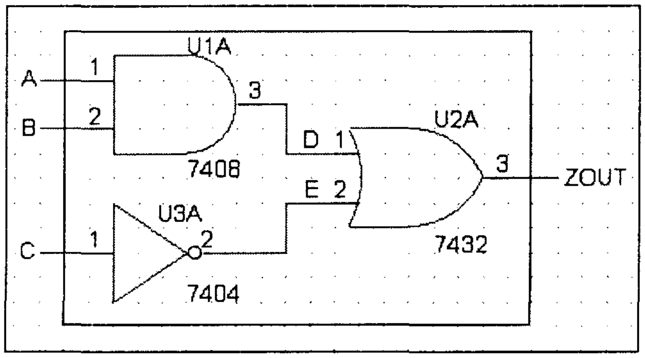
```
label:  for identifier in start-value to end-value generate
        component-instantiation-statements;
        end generate;
```

```
gen1:   for t  in 1 to 4 generate
        inside1: AND2 portmap (A(t), B(t), D(t));
        end generate;
```

Figure 4 – Generic syntax and example of a for-generate loop.

A.5 - Lesson 5: Dataflow Modeling

How does the dataflow style compare to the previously discussed structural style? Like the structural style, dataflow models the circuit through use of concurrent statements. However, the flow of data is expressed as opposed to individual components. Instead of being able to see within the code just exactly how the circuit is interconnected, the functionality is instead expressed. For example, compare Figure 1 from Lesson 4 to that of the dataflow style used in Figure 1 of this lesson. The intermediate signals D and E are not needed in the example but are shown for the ease of the reader. Without these signals, there would be one single statement after the keyword **begin**, which would be: [ZOUT <= (A and B) or (not C);]



```
entity FIRST_VHDL is
        port (A, B, C: in BIT; ZOUT: out BIT);
end FIRST_VHDL;
```

```
architecture FIRST_VHDL_ARCH of FIRST_VHDL is
        signal D, E: BIT;
begin
        D <= A and B;
        E <= not C;
        ZOUT <= D or E;
end FIRST_VHDL_ARCH;
```

Figure 1 – An example of a schematic and its corresponding dataflow style code.

Once again, there is a declarations and statements section in the architecture body.

- In the declarations section of a dataflow model, signals, variables, or other data objects not declared in the entity but needed for the architecture are declared here; not components. These declaration statements can be made using the methods learned in Lesson 2. Thus, the declaration section of dataflow modeling requires little further discussion.

- The concurrent statements in the architecture body, however, require more discussion since the only concurrent statements examined thus far were the component instantiation statements from Lesson 4. The most common statement used in dataflow modeling can be seen in Figure 1 and is called a signal assignment statement. The generic syntax of this statement can be found as Figure 2 on the following page.

```
target-signal-name <= expression after time-period;
```

Figure 2 – The generic syntax for a signal assignment statement.

How does the signal assignment work? The statement assigns a value to the data object on the left side (which is called the target signal) of the '<=' symbol. The signal assignment statements are then executed whenever any value to the right of the '<=' symbol changes and this is otherwise known as an event. Also, as seen in Figure 2, delays can be accounted for by specifying a delay time (usually in nanoseconds (ns)) following the **after** keyword. The statement would then execute once an even to occurs as mentioned before and assign the value of the target signal after (*time-period)* delay. If no delay is specified, the delay is called a "delta delay". This **after** statement can be used to create a clock within the architecture in the manner shown in Figure 3.

```
CLOCK <= not CLOCK after time-period;
```

Figure 3 – The implementation of a clock using signal assignment statements.

There are two additional signal assignment statements: conditional and selected. A conditional signal assignment statement is similar to an if statement since it selects values for the target signal based upon certain conditions. The generic syntax of the conditional signal assignment statement as well as an example of how it is used can be found in Figure 4. A boolean comparison after the **when** keyword is used to determine whether the expression to the left of the **when** keyword will be assigned to the target signal.

```
target-signal-name <= expression when boolean-comparison else
                         expression;
```

```
Go_Signal <= '1' when Cross_Signal='0' and Sensor='1' else '0';
```

Figure 4 - Generic syntax and example for a conditional signal assignment statement

A selected signal assignment statement is similar to a case statement since it selects a value from a given expression that matches one of the choices and then assigns the expression to the target signal. The selected signal assignment is executed whenever an event occurs on either the select expression or on one of these signals present within the statement. The generic syntax of the selected signal assignment statement as well as an example of how it is used can be found in Figure 5 on the following page. The example in Figure 5 assumes that the type ALU has been declared with the following values: ADD, SUB, INCR, DECR, F_AND, F_OR, A_NOT, and F_XOR.

```
with select-expression select
        target-signal-name <= expression when choices,
                              expression when choices,
                              ..............................
                              expression when choices;
```

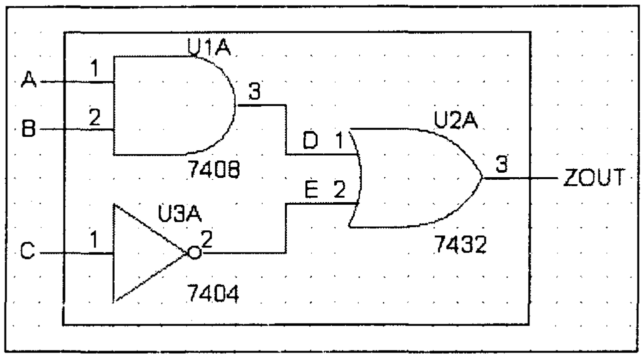
```
with ALU select
        F <= A+B when ADD,
        F <= A-B  when SUB,
        F <= A+1 when INCR,
        F <= A-1 when DECR,
        F <= A and B when F_AND,
        F <= A or B when F_OR
        F <= not A when A_NOT
        F <= A xor B when F_XOR;
```

Figure 5 - Generic syntax and example for a selected signal assignment statement.

The last expression to discuss with the dataflow modeling is the **unaffected** keyword. This keyword is analogous to the sequential **null** keyword that is used within the behavioral modeling style. The **unaffected** keyword is used with concurrent signal assignment statements so as to cause no change to the target signal. This statement is often used within selected and conditional signal statements combined with the **other** keyword to keep the target signal status quo unless some specified condition occurs. It is important with the dataflow modeling especially to look at some examples within the text to learn the in's and out's of this style more in depth. There are many small variations that are impossible to list that are best learned by example.

A.6 - Lesson 6: Behavioral Modeling

How does the behavioral style compare to the previously discussed styles? This style is significantly different from the previously discussed styles in that it uses sequential statements instead of concurrent ones. This is similar to the way that a language like C++ operates. The way that these sequential statements are used is within what is called a process statement. Just as with the dataflow style, this structure does not describe the structure of an entity but instead its behavior or functionality. This style allows almost no simulated time during execution and provides an outlet for users used to high-level programming languages. The same circuit used in Lessons 4 and 5 is provided below as well as the corresponding behavioral styled VHDL coding.



```
entity FIRST_VHDL is
        port (A, B, C: in BIT; ZOUT: out BIT);
end FIRST_VHDL;
```

```
architecture FIRST_VHDL_ARCH of FIRST_VHDL is
begin
        process (A,B,C)
                variable D, E: BIT;
        begin
                D <= A and B;
                E <= not C;
                ZOUT <= D or E;
end FIRST_VHDL_ARCH;
```

Figure 1 – An example of a schematic and its corresponding behavioral style code.

Once again, there is a declaration and statement section of the architecture body. The declaration section would be used in the same manner as in the dataflow style. The difference between these styles though is that the declarations take place within the process statement in the behavioral style as can be seen in Figure 1 above.

How is the process statement used with the behavioral style of modeling? The process statement is used to contain the sequential statements used in this style. It is contained within the architecture body after the **begin** keyword where the statements of an architecture are normally written. This **begin** keyword should not be confused with the **begin** keyword inside the process statement. This new structure is similar to the previously discussed styles of architecture bodies since it has a declarations and statements section of its own. The generic syntax of a process statement can be seen in Figure 2.

```
process (sensitivity-list)
        process declaration statements
begin
        sequential statements
        report statements
        null statements
        if statements
        case statements
        loop statements
end process;
```

Figure 2 – The generic syntax of a process statement.

What are the parts of a process statement?

- The first section of the process statement is the process line, which contains the keyword **process**. The list of signals after **process** is called a sensitivity list. This list contains a set of signals that causes the statements within the process to execute whenever an event appears on one of them. The statements then execute sequentially until the last one has completed and then the process suspends (unless another signal in the sensitivity list had its value change, which would cause the process to execute again). As is the case with other languages, infinite loops can occur because of the properties stated and one must be careful to avoid this dilemma.
- The second section of the process statement contains the declaration statements. Data objects that are declared within here can only be used within the process statement and are thus called local variables. A discussion of object declarations can be found in Lesson 2.
- The third section of the process statement is where all of the sequential statements are contained. As can be seen from Figure 2, there are quite a few different sequential statements that can be used and these will be the topic of the rest of this lesson. The three main groupings of these sequential statement structures are: if statements, case statements, and loop statements.

What is an 'IF' statement? An 'IF' statement is a conditional statement that is used to execute a series of sequential statements within it if it is true. Also, zero or more **elsif** statements can be used within an established 'IF' statement to specify alternative conditions. In the same manner, an **else** statement can optionally be used to cover the entire set of alternative possibilities for when the previous conditions in the 'IF' statement are false. Lastly, two or more 'IF' statements can be nested within each other as in most high level programming languages. Figure 3 provides the generic syntax for a single 'IF' statement.

```
If conditional statement then
        sequential statement(s)
elsif conditional statement then
        sequential statement(s)
else
        sequential statement(s)
endif;
```

Figure 3 – The generic syntax for an 'IF' statement in VHDL.

What is a 'CASE' statement? A case statement uses an established expression in it's open declaration and selects a branch from the series of sequential statements to be executed depending on the value of this expression. A **when others** statement can be used within a 'CASE' statement in the same fashion as 'ELSE' was used in an 'IF' statements. "WHEN OTHERS" allows all other conditions to be covered that were not previously in the set of branches. This is important because a 'CASE' statement in VHDL requires that all possible values that the expression can take on must be defined. More than one value can be added to a single branch through use of the '|' operator. The generic syntax of a 'CASE' statement can be viewed in Figure 4 below.

```
case case expression is
        when branch value => sequential statement(s)
        when branch value => sequential statement(s)
        when branch value => sequential statement(s)
        when others => sequential statement(s)
end case;
```

Figure 4 – The generic syntax of a 'CASE' statement in VHDL.

What is a 'LOOP' statement? A 'LOOP' statement is used to execute a series of sequential statements repeatedly unless otherwise specified. There are three main types of 'LOOP' statements: 'FOR', 'WHILE', and just a plain 'LOOP'. The 'FOR' loop executes the set of statements each time the identifier is within the range specified. The 'WHILE' loop executes continuously as long as the conditional statement remains true. Lastly, the plain 'LOOP' statement executes repeatedly until a condition specified within the loop causes it to cease (such as an **exit** or **return** command). Please note that the MAX+PLUS II software does not allow 'FOR' loops. The generic syntax for each of these three types of loops is displayed below in Figure 5.

```
for identifier in specified range loop
        sequential statement(s)
end loop;
```

```
while conditional statement
        sequential statement(s)
end loop;
```

```
loop
        sequential statement(s)
exit when conditional statement
end loop;
```

Figure 5 – The generic syntax for 'FOR', 'WHILE' and 'LOOP' statements.

APPENDIX B

EXCEL SLOT MACHINE DESIGN CODING

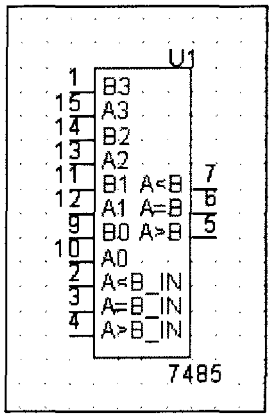
Cell No.	Conditional Formatting	Notes
A1 - E1		Merge Cells, =IF(G21=1,"WINNER!","") Font Color - Red, Font Type - Arial Font Size 48, Background Color = Black
A2		Background Color = Black
B2		Background Color = Black
C2		Background Color = Black
D2		Background Color = Black
E2		Background Color = Black
A3		Background Color = Black
B3	IF 'F3' = 4, Then Background Color = Red	Background Color = Brown, Red
C3	IF 'G3' = 7, Then Background Color = Red	Background Color = Brown, Red
D3	IF 'H3' = 2, Then Background Color = Red	Background Color = Brown, Red
E3		Background Color = Black
A4		Background Color = Black
B4	IF 'F3' = 3, Then Background Color = White	Background Color = Brown, White
C4	IF 'G3' = 3, Then Background Color = White	Background Color = Brown, White
D4	IF 'H3' = 3 OR 6, Then Background Color = White	Background Color = Brown, White
E4		Background Color = Black
A5		Background Color = Black
B5	IF 'F3' = 1 OR 12, Then Background Color = Blue	Background Color = Brown, Blue
C5	IF 'G3' = 5, Then Background Color = Blue	Background Color = Brown, Blue
D5	IF 'H3' = 4, Then Background Color = Blue	Background Color = Brown, Blue
E5		Background Color = Black
A6		Background Color = Black
B6	IF 'F3' = 7, Then Background Color = White	Background Color = Brown, White
C6	IF 'G3' = 9, Then Background Color = White	Background Color = Brown, White
D6	IF 'H3' = 8 OR 9, Then Background Color = White	Background Color = Brown, White
E6		Background Color = Black
A7		Background Color = Black
B7	IF 'F3' = 6 OR 8 OR 13, Then Background Color = Green	Background Color = Brown, Green
C7	IF 'G3' = 2 OR 6, Then Background Color = Green	Background Color = Brown, Green
D7	IF 'H3' = 10, Then Background Color = Green	Background Color = Brown, Green
E7		Background Color = Black
A8		Background Color = Black
B8	IF 'F3' = 10, Then Background Color = White	Background Color = Brown, White
C8	IF 'G3' = 12, Then Background Color = White	Background Color = Brown, White
D8	IF 'H3' = 12 OR 14 Then Background Color = White	Background Color = Brown, White
E8		Background Color = Black
A9		Background Color = Black
B9	IF 'F3' = 2 OR 5 OR 11, Then Background Color = Orange	Background Color = Brown, Orange
C9	IF 'G3' = 1 OR 10 OR 13, Then Background Color = Orange	Background Color = Brown, Orange
D9	IF 'H3' = 13, Then Background Color = Orange	Background Color = Brown, Orange
E9		Background Color = Black
A10		Background Color = Black
B10	IF 'F3' = 14, Then Background Color = White	Background Color = Brown, White
C10	IF 'G3' = 15, Then Background Color = White	Background Color = Brown, White
D10	IF 'H3' = 15, Then Background Color = White	Background Color = Brown, White
E10		Background Color = Black
A11		Background Color = Black
B11	IF 'F3' = 9 OR 15, Then Background Color = Purple	Background Color = Brown, Purple
C11	IF 'G3' = 4 OR 8 OR 11 OR 14, Then Background Color = Purple	Background Color = Brown, Purple
D11	IF 'H3' = 1 OR 5 OR 7 OR 11, Then Background Color = Purple	Background Color = Brown, Purple
E11		Background Color = Black
A12		Background Color = Black
B12		Background Color = Black
C12		Background Color = Black
D12		Background Color = Black
E12		Background Color = Black
A13		Background Color = Black
B13		Background Color = Black
C13		Background Color = Black
D13		Background Color = Black
E13		Background Color = Black
A14		Background Color = Black
B14		Background Color = Black
C14		Background Color = Black
D14		Background Color = Black
E14		Background Color = Black
A15		Background Color = Black
B15		Payout (Hidden)
C15		IF(G12=1,500,IF(G13=1,100,IF(G14=1,50,IF(G15=1,25,IF(G17=1,10,IF(G19=1,3,IF(G20=1,2,0))))))) - (HIDDEN)
D15		Background Color = Black
E15		Background Color = Black
A16		Background Color = Black
B16		Red Bonus (Hidden)
C16		IF(G16=1,20,IF(G18=1,5,0)) - (HIDDEN)
D16		Background Color = Black
E16		Background Color = Black

Cell No	Conditional Formatting	Notes
A17		Background Color = Black
B17		Background Color = Black, Font Color = Yellow, Payout:
C17		C15+C16
D17		Background Color = Black
E17		Background Color = Black
A18		Background Color = Black
B18		Background Color = Black
C18		Background Color = Black
D18		Background Color = Black
E18		Background Color = Black
A19		Background Color = Black
B19		Background Color = Black
C19		Background Color = Black
D19		Background Color = Black
E19		Background Color = Black
A20		Background Color = Black
B20		Background Color = Black
C20		Background Color = Black
D20		Background Color = Black
E20		Background Color = Black
A21		Background Color = Black
B21		Background Color = Black
C21		Background Color = Black
D21		Background Color = Black
E21		Background Color = Black
A22		Background Color = Black
B22 - D22		Merge Cells, Backround Color = Black, Font Color = Red, Click on Handle to Spin Reels
E22		Background Color = Black
A23		Background Color = Black
B23		Background Color = Black
C23		Background Color = Black
D23		Background Color = Black
E23		Background Color = Black
A24		Background Color = Black
B24		Background Color = Black
C24		Background Color = Black
D24		Background Color = Black
E24		Background Color = Black
F3		(INT(RAND()*15))+1
F12		Red Bonus (All 3)
F13		Blue Bonus (All 3)
F14		Green Bonus
F15		Orange Bouns
F16		Red Bonus (1st 2)
F17		Blue Bonus (1st 2)
F18		Red Bonus (1st 1)
F19		Purple Bonus
F20		White Bonus
G3		(INT(RAND()*15))+1
G12		IF(AND(F3=4,G3=7,H3=2),1,0)
G13		IF(AND(OR(F3=1,F3=12),G3=5,H3=4),1,0)
G14		IF(AND(OR(F3=6,F3=8,F3=13),OR(G3=2,G3=6),H3=10),1,0)
G15		IF(AND(OR(F3=2,F3=5,F3=11),OR(G3=1,G3=10,G3=13),H3=13),1,0)
G16		IF(AND(F3=4,G3=7,H3<>2),1,0)
G17		IF(AND(OR(F3=1,F3=12),G3=5,H3<>4),1,0)
G18		IF(AND(F3=4,G3<>7),1,0)
G19		IF(AND(OR(F3=9,F3=15),OR(G3=4,G3=8,G3=11,G3=14),OR(H3=1,H3=5,H3=7,H3=11)),1,0)
G20		IF(AND(OR(F3=3,F3=7,F3=10,F3=14),OR(G3=3,G3=9,G3=12,G3=15),OR(H3=3,H3=6,H3=8,H3=9,H3=12,H3=14,H3=15)),1,0)
G21		SUM(G12:G20)
H3		(INT(RAND()*15))+1

APPENDIX C

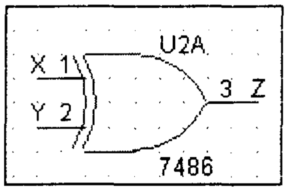
LOGIC CHIP SYMBOLS AND TRUTH TABLES

C.1 – The 74x85 4-Bit Magnitude Comparator



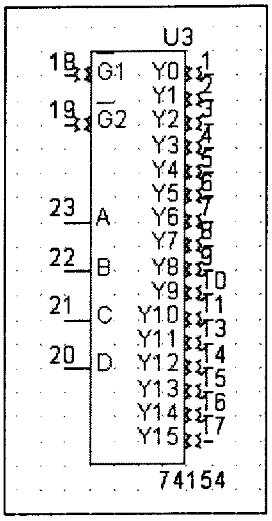
B3 - B0	A3 - A0	A>B_IN	A=B_IN	A<B_IN	A>B	A=B	A<B
EQUAL	EQUAL	0	0	0	0	1	0
EQUAL	EQUAL	0	0	1	0	0	1
EQUAL	EQUAL	0	1	0	0	1	0
EQUAL	EQUAL	1	0	0	1	0	0

C.2 – The 74x86 2-Input XOR Gate



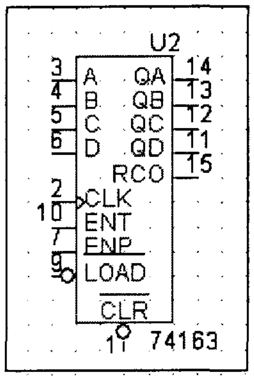
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

C.3 – The 74x154 4-Line to 16-Line Decoder/Multiplexer



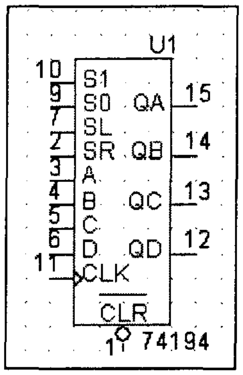
G1	G2	D	C	B	A	Y15	Y14	Y13	Y12	Y11	Y10	Y9	Y8	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
1	---	---	---	---	---	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	1	---	---	---	---	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
---	---	1	---	---	---	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1
1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1
1	0	1	0	0	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1
1	0	1	0	0	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1
1	0	1	0	1	0	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1
1	0	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	0	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	0	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

C.4 – The 74x163 Synchronous 4-Bit Counter



CLR_L	LOAD_L	ENP	ENT	QD	QC	QB	QA	RCO	QD*	QC*	QB*	QA*
0	---	---	---	---	---	---	---	---	0	0	0	0
1	0	---	---	---	---	---	---	---	D	C	B	A
1	1	0	---	---	---	---	---	---	QD	QC	QB	QA
1	1	---	0	---	---	---	---	---	QD	QC	QB	QA
1	1	1	1	0	0	0	0	0	0	0	0	1
1	1	1	1	0	0	0	1	0	0	0	1	0
1	1	1	1	0	0	1	0	0	0	0	1	1
1	1	1	1	0	0	1	1	0	0	1	0	0
1	1	1	1	0	1	0	0	0	0	1	0	1
1	1	1	1	0	1	0	1	0	0	1	1	0
1	1	1	1	0	1	1	0	0	0	1	1	1
1	1	1	1	0	1	1	1	0	1	0	0	0
1	1	1	1	1	0	0	0	0	1	0	0	1
1	1	1	1	1	0	0	1	0	1	0	1	0
1	1	1	1	1	0	1	0	0	1	0	1	1
1	1	1	1	1	0	1	1	0	1	1	0	0
1	1	1	1	1	1	0	0	0	1	1	0	1
1	1	1	1	1	1	0	1	0	1	1	1	0
1	1	1	1	1	1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1	1	0	0	0	0
1	1	0	1	1	1	1	1	1	1	1	1	1

C.5 – The 74x194 4-Bit Bidirectional Universal Shift Register

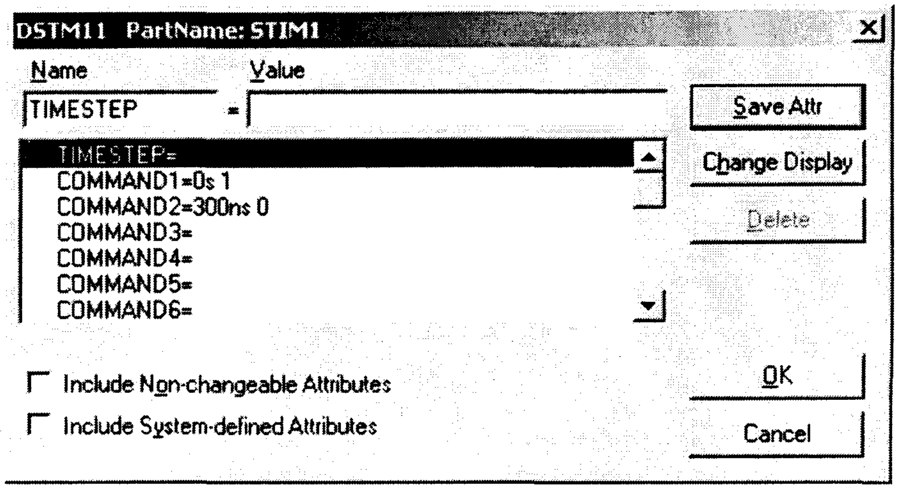


CLR_L	S1	S0	QA*	QB*	QC*	QD*
0	---	---	0	0	0	0
1	0	0	QA	QB	QC	QD
1	0	1	SR	QA	QB	QC
1	1	0	QB	QC	QD	SL
1	1	1	A	B	C	D

APPENDIX D

SSI/MSI DESIGN TIMING INFORMATION

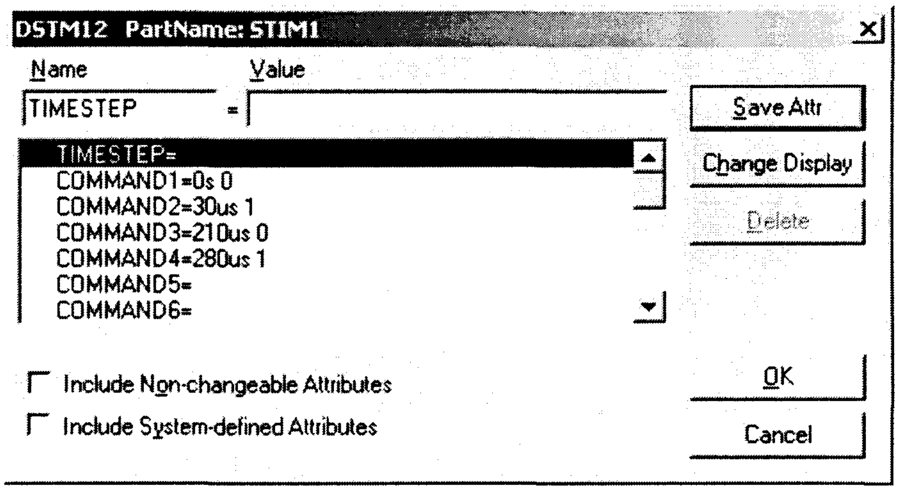
D.1 – The Clock(s) Setup for the PSpice Simulation



TIMESTEP =

```
TIMESTEP=
COMMAND1=0s 1
COMMAND2=300ns 0
COMMAND3=
COMMAND4=
COMMAND5=
COMMAND6=
```

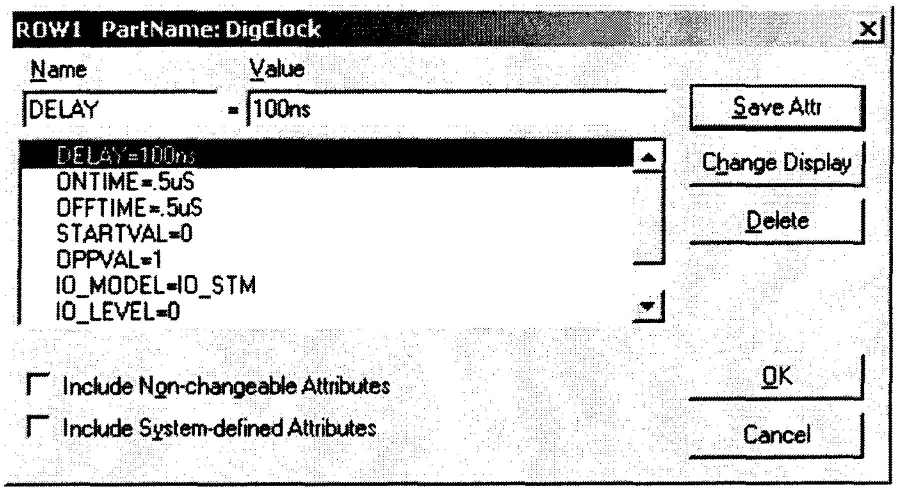
☐ Include Non-changeable Attributes
☐ Include System-defined Attributes



TIMESTEP =

```
TIMESTEP=
COMMAND1=0s 0
COMMAND2=30us 1
COMMAND3=210us 0
COMMAND4=280us 1
COMMAND5=
COMMAND6=
```

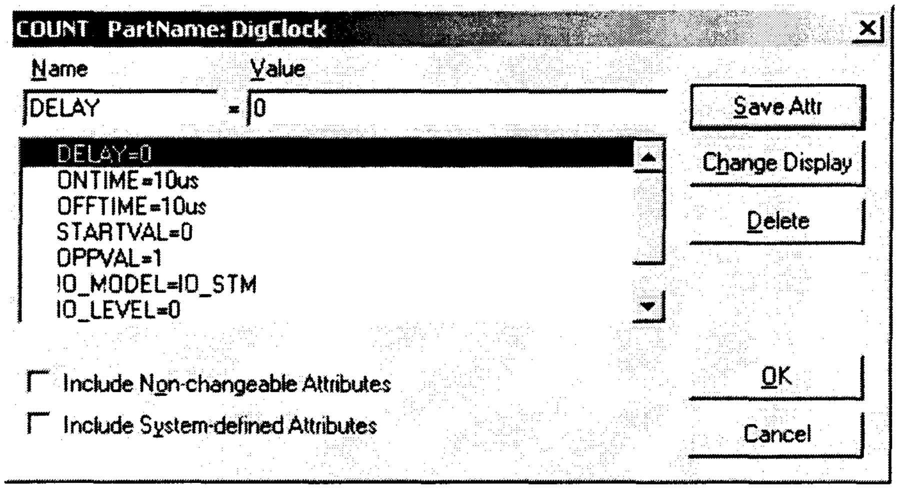
☐ Include Non-changeable Attributes
☐ Include System-defined Attributes



DELAY = 100ns

DELAY=100ns
ONTIME=.5uS
OFFTIME=.5uS
STARTVAL=0
OPPVAL=1
IO_MODEL=IO_STM
IO_LEVEL=0

☐ Include Non-changeable Attributes
☐ Include System-defined Attributes



DELAY = 0

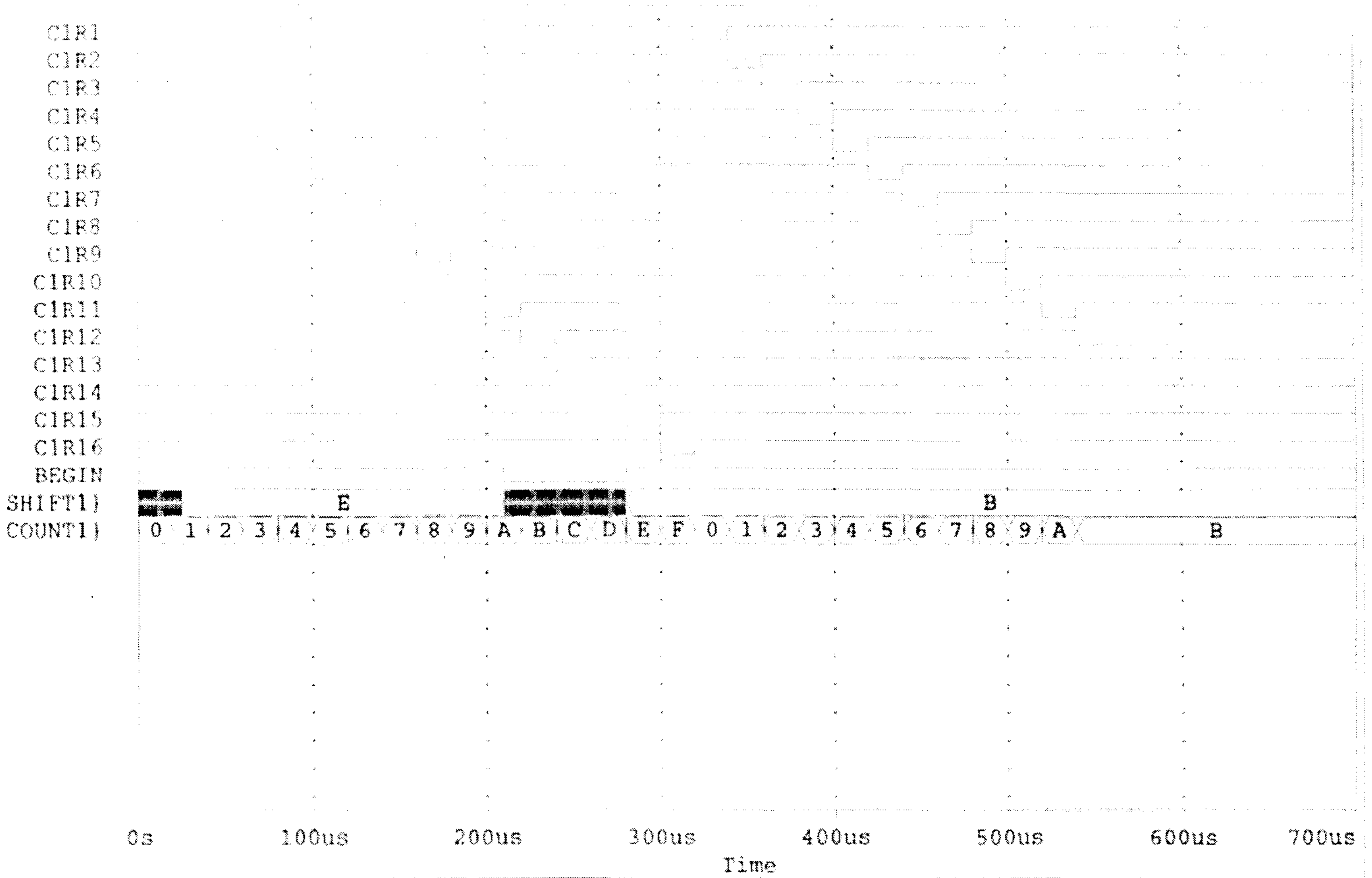
DELAY=0
ONTIME=10us
OFFTIME=10us
STARTVAL=0
OPPVAL=1
IO_MODEL=IO_STM
IO_LEVEL=0

☐ Include Non-changeable Attributes
☐ Include System-defined Attributes

**** Note: For these displays, only the actual data that was changed in the dialogue boxes is shown. All of the other variables that are not displayed were left to default values.**

Date/Time run 10/24/102 16:24:04 Temperature: 7.0

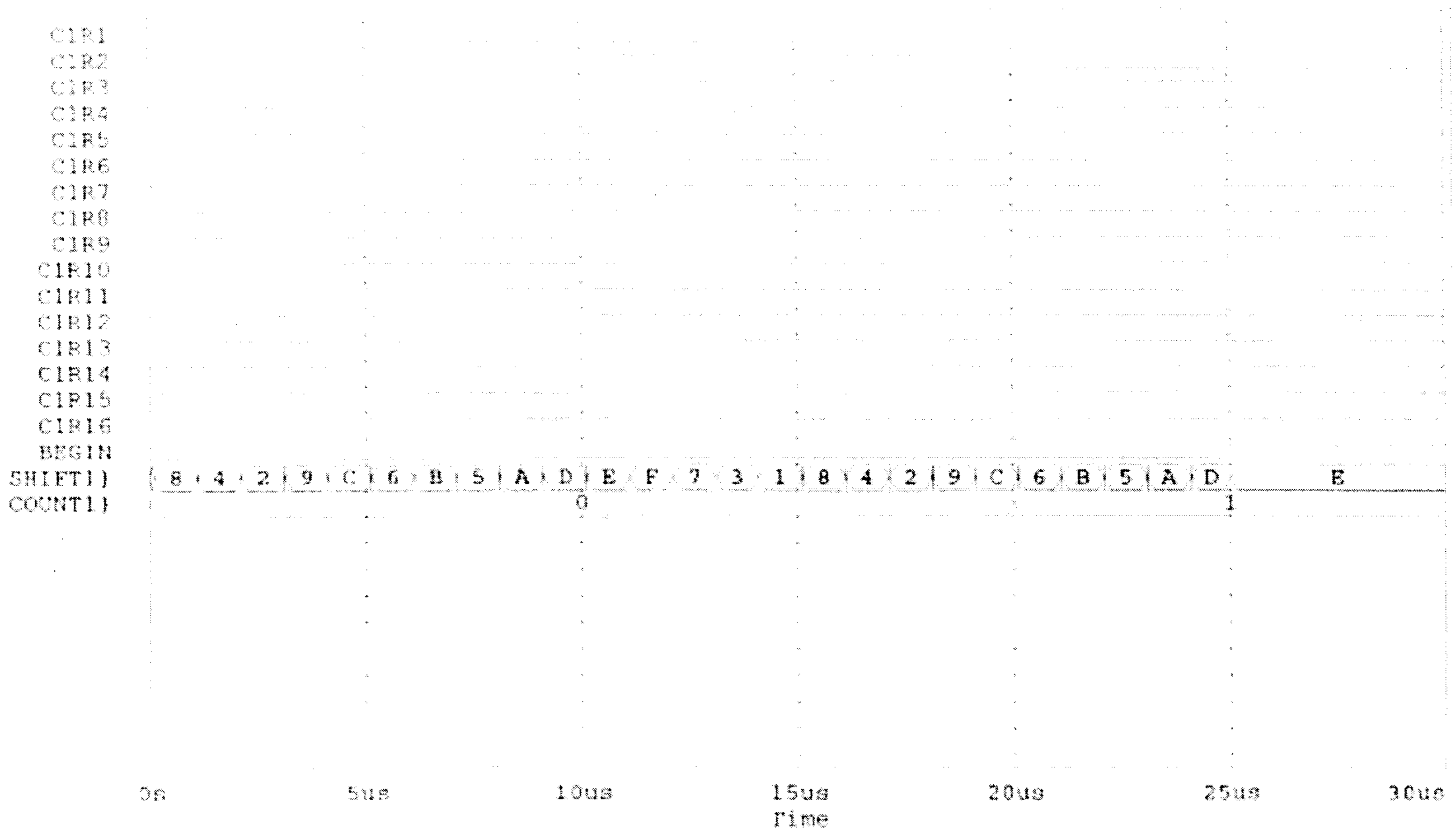
(A) New.dat



Date October 24, 2002 Page 1 Time: 16:28:20

Date/Time run: 10/24/102 16:24:04 Temperature: 27.0

(A) New.dat



Date October 24 2002 Page 1 Time 6:28:28

APPENDIX E

THE DESIGN REPORT FILE

```
Project Information                          c:\max2work\vhdl\luke3.rpt

MAX+plus II Compiler Report File
Version 10.1 06/12/2001
Compiled: 10/29/2002 17:37:26

Copyright (C) 1988-2001 Altera Corporation
Any megafunction design, and related net list (encrypted or decrypted),
support information, device programming or simulation file, and any other
associated documentation or information provided by Altera or a partner
under Altera's Megafunction Partnership Program may be used only to
program PLD devices (but not masked PLD devices) from Altera.  Any other
use of such megafunction design, net list, support information, device
programming or simulation file, or any other related documentation or
information is prohibited for any other purpose, including, but not
limited to modification, reverse engineering, de-compiling, or use with
any other silicon devices, unless such use is explicitly licensed under
a separate agreement with Altera or a megafunction partner.  Title to
the intellectual property, including patents, copyrights, trademarks,
trade secrets, or maskworks, embodied in any such megafunction design,
net list, support information, device programming or simulation file, or
any other related documentation or information provided by Altera or a
megafunction partner, remains with Altera, the megafunction partner, or
their respective licensors.  No other licenses, including any licenses
needed under any third party's intellectual property, are provided herein.


***** Project compilation was successful


LUKE3


** DEVICE SUMMARY **

Chip/                     Input   Output   Bidir          Shareable
POF       Device          Pins    Pins     Pins     LCs   Expanders  % Utilized

luke3     EPM7128SLC84-7   5        27       0      39       0           30 %

User Pins:                 5        27       0
```

```
Project Information                              c:\max2work\vhdl\luke3-rpt

** AUTO GLOBAL SIGNALS **


INFO: Signal 'CLK' chosen for auto global Clock
INFO: Signal 'CLK2' chosen for auto global Clock
```

```
Project Information                              c:\max2work\vhdl\luke3.rpt

** FILE HIERARCHY **


|lpm_add_sub:963|
|lpm_add_sub:963|addcore:adder|
|lpm_add_sub:963|addcore:adder|addcore:adder0|
|lpm_add_sub:963|altshift:result_ext_latency_ffs|
|lpm_add_sub:963|altshift:carry_ext_latency_ffs|
|lpm_add_sub:963|altshift:oflow_ext_latency_ffs|
|lpm_add_sub:3395|
|lpm_add_sub:3395|addcore:adder|
|lpm_add_sub:3395|addcore:adder|addcore:adder0|
|lpm_add_sub:3395|altshift:result_ext_latency_ffs|
|lpm_add_sub:3395|altshift:carry_ext_latency_ffs|
|lpm_add_sub:3395|altshift:oflow_ext_latency_ffs|
|lpm_add_sub:5827|
|lpm_add_sub:5827|addcore:adder|
|lpm_add_sub:5827|addcore:adder|addcore:adder0|
|lpm_add_sub:5827|altshift:result_ext_latency_ffs|
|lpm_add_sub:5827|altshift:carry_ext_latency_ffs|
|lpm_add_sub:5827|altshift:oflow_ext_latency_ffs|
Device-Specific Information:                     c:\max2work\vhdl\luke3.rpt
luke3

***** Logic for device 'luke3' compiled without errors.


Device: EPM7128SLC84-7

Device Options:
    Turbo Bit                                    = ON
    Security Bit                                 = OFF
    Enable JTAG Support                          = ON
    User Code                                    = ffff
    MultiVolt I/O                                = OFF

                     R  R     R  R  R                    C  C  C     C  C  C
                     E  E     E  E  E                    O  O  O     O  O  O
                     S  S     S  S  S  V                 L  L  L     L  L  L
               S     E  E     E  E  E  C                 U  U  U  V  U  U  U
               T  C  R  R     R  R  R  C  C              M  M  M  C  M  M  M
               A  L  V  V  G  V  V  V  I  L  G  G  C  G  N  N  N  C  N  N  N
               R  K  E  E  N  E  E  E  N  K  N  N  L  N  3  3  3  I  3  3  3
               T  3  D  D  D  D  D  D  T  2  D  D  K  D  7  6  0  O  1  2  3
           ----------------------------------------------------------------_
            / 11 10  9  8  7  6  5  4  3  2  1 84 83 82 81 80 79 78 77 76 75   |
      GO | 12                                                               74 | COLUMN34
   VCCIO | 13                                                               73 | COLUMN38
    #TDI | 14                                                               72 | GND
RESERVED | 15                                                               71 | #TDO
RESERVED | 16                                                               70 | COLUMN23
RESERVED | 17                                                               69 | COLUMN21
RESERVED | 18                                                               68 | COLUMN20
     GND | 19                                                               67 | COLUMN24
RESERVED | 20                                                               66 | VCCIO
RESERVED | 21                                                               65 | COLUMN35
RESERVED | 22                        EPM7128SLC84-7                         64 | COLUMN25
    #TMS | 23                                                               63 | COLUMN22
RESERVED | 24                                                               62 | #TCK
RESERVED | 25                                                               61 | COLUMN28
   VCCIO | 26                                                               60 | COLUMN10
RESERVED | 27                                                               59 | GND
RESERVED | 28                                                               58 | COLUMN26
RESERVED | 29                                                               57 | COLUMN13
RESERVED | 30                                                               56 | COLUMN27
RESERVED | 31                                                               55 | COLUMN12
     GND | 32                                                               54 | COLUMN11
           |_ 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53  _|
             ------------------------------------------------------------------
               R  R  R  R  R  V  R  R  R  G  V  R  R  C  G  C  C  C  C  R  V
               E  E  E  E  E  C  E  E  E  N  C  E  E  O  N  O  O  O  O  E  C
               S  S  S  S  S  C  S  S  S  D  C  S  S  L  D  L  L  L  L  S  C
               E  E  E  E  E  I  E  E  E     I  E  E  U     U  U  U  U  E  I
               R  R  R  R  R  O  R  R  R     N  R  R  M     M  M  M  M  R  O
               V  V  V  V  V     V  V  V     T  V  V  N     N  N  N  N  V
               E  E  E  E  E     E  E  E        E  E  1     1  1  1  1  E
               D  D  D  D  D     D  D  D        D  D  7     8  6  5  4  D


N.C. = No Connect. This pin has no internal connection to the device.
VCCINT = Dedicated power pin, which MUST be connected to VCC (5.0 volts).
VCCIO = Dedicated power pin, which MUST be connected to VCC (5.0 volts).
GND = Dedicated ground pin or unused dedicated input, which MUST be connected to GND.
RESERVED = Unused I/O pin, which MUST be left unconnected.

^ = Dedicated configuration pin.
+ = Reserved configuration pin, which is tri-stated during user mode.
* = Reserved configuration pin, which drives out in user mode.
```

PDn = Power Down pin.
@ = Special-purpose pin.
* = JTAG Boundary-Scan Testing/In-System Programming or Configuration Pin. The JTAG inputs TMS and TDI should be tied to VCC and TCK should be tied to GND when not in use.
& = JTAG pin used for I/O. When used as user I/O, JTAG pins must be kept stable before and during configuration. JTAG pin stability prevents accidental loading of JTAG instructions.

```
Device-Specific Information:                      c:\max2work\vhdl\luke3.rpt
luke3

** RESOURCE USAGE **

                                                   Shareable      External
Logic Array Block      Logic Cells    I/O Pins     Expanders    Interconnect

A:     LC1 - LC16      0/16(  0%)    3/ 8( 37%)   0/16(  0%)    0/36(  0%)
B:    LC17 - LC32      0/16(  0%)    1/ 8( 12%)   0/16(  0%)    0/36(  0%)
C:    LC33 - LC48      0/16(  0%)    1/ 8( 12%)   0/16(  0%)    0/36(  0%)
D:    LC49 - LC64      0/16(  0%)    0/ 8(  0%)   0/16(  0%)    0/36(  0%)
E:    LC65 - LC80      8/16( 50%)    5/ 8( 62%)   0/16(  0%)    6/36( 16%)
F:    LC81 - LC96      8/16( 50%)    8/ 8(100%)   0/16(  0%)   10/36( 27%)
G:   LC97 - LC112     11/16( 68%)    8/ 8(100%)   0/16(  0%)    9/36( 25%)
H:  LC113 - LC128     12/16( 75%)    8/ 8(100%)   1/16(  6%)    7/36( 19%)


Total dedicated input pins used:                   2/4       ( 50%)
Total I/O pins used:                              34/64      ( 53%)
Total logic cells used:                           39/128     ( 30%)
Total shareable expanders used:                    0/128     (  0%)
Total Turbo logic cells used:                     39/128     ( 30%)
Total shareable expanders not available (n/a):     1/128     (  0%)
Average fan-in:                                    6.02
Total fan-in:                                     235

Total input pins required:                         5
Total fast input logic cells required:             0
Total output pins required:                       27
Total bidirectional pins required:                 0
Total reserved pins required                       4
Total logic cells required:                       39
Total flipflops required:                         12
Total product terms required:                     67
Total logic cells lending parallel expanders:      0
Total shareable expanders in database:             0

Synthesized logic cells:                           0/ 128    (  0%)
```

```
Device-Specific Information:                          c:\max2work\vhdl\luke3.rpt
luke3

** INPUTS **

                                                 Shareable
                                                 Expanders     Fan-In    Fan-Out
 Pin     LC   LAB  Primitive    Code    Total Shared n/a INP  FBK  OUT  FBK  Name
  83      -    -      INPUT  G              0      0   0   0    0    0    0  CLK
   2      -    -      INPUT  G              0      0   0   0    0    0    0  CLK2
  10    (6)  (A)      INPUT                 0      0   0   0    0    0    4  CLK3
  12    (3)  (A)      INPUT                 0      0   0   0    0   27   12  GO
  11    (5)  (A)      INPUT                 0      0   0   0    0   27   12  START


Code:

s = Synthesized pin or logic cell
t = Turbo logic cell
+ = Synchronous flipflop
/ = Slow slew-rate output
! = NOT gate push-back
r = Fitter-inserted logic cell
G = Global Source. Fan-out destinations counted here do not include destinations
that are driven using global routing resources. Refer to the Auto Global Signals,
Clock Signals, Clear Signals, Synchronous Load Signals, and Synchronous Clear Signals
Sections of this Report File for information on which signals' fan-outs are used as
Clock, Clear, Preset, Output Enable, and synchronous Load signals.
```

```
Device-Specific Information:                          c:\max2work\vhdl\luke3.rpt
luke3

** OUTPUTS **

                                             Shareable
                                             Expanders      Fan-In    Fan-Out
 Pin      LC  LAB  Primitive      Code    Total Shared n/a INP  FBK  OUT  FBK  Name
  60      93   F      OUTPUT        t         0      0   0   2    4    0    0  COLUMN10
  54      83   F      OUTPUT        t         0      0   0   2    4    0    0  COLUMN11
  55      85   F      OUTPUT        t         0      0   0   2    4    0    0  COLUMN12
  57      88   F      OUTPUT        t         0      0   0   2    4    0    0  COLUMN13
  51      77   E      OUTPUT        t         0      0   0   2    4    0    0  COLUMN14
  50      75   E      OUTPUT        t         0      0   0   2    4    0    0  COLUMN15
  49      73   E      OUTPUT        t         0      0   0   2    4    0    0  COLUMN16
  46      69   E      OUTPUT        t         0      0   0   2    4    0    0  COLUMN17
  48      72   E      OUTPUT        t         0      0   0   2    4    0    0  COLUMN18
  68     105   G      OUTPUT        t         0      0   0   2    4    0    0  COLUMN20
  69     107   G      OUTPUT        t         0      0   0   2    4    0    0  COLUMN21
  63      97   G      OUTPUT        t         0      0   0   2    4    0    0  COLUMN22
  70     109   G      OUTPUT        t         0      0   0   2    4    0    0  COLUMN23
  67     104   G      OUTPUT        t         0      0   0   2    3    0    0  COLUMN24
  64      99   G      OUTPUT        t         0      0   0   2    4    0    0  COLUMN25
  58      91   F      OUTPUT        t         0      0   0   2    4    0    0  COLUMN26
  56      86   F      OUTPUT        t         0      0   0   2    4    0    0  COLUMN27
  61      94   F      OUTPUT        t         0      0   0   2    4    0    0  COLUMN28
  79     125   H      OUTPUT        t         0      0   0   2    4    0    0  COLUMN30
  77     123   H      OUTPUT        t         0      0   0   2    4    0    0  COLUMN31
  76     120   H      OUTPUT        t         0      0   0   2    4    0    0  COLUMN32
  75     118   H      OUTPUT        t         0      0   0   2    4    0    0  COLUMN33
  74     117   H      OUTPUT        t         0      0   0   2    4    0    0  COLUMN34
  65     101   G      OUTPUT        t         0      0   0   2    3    0    0  COLUMN35
  80     126   H      OUTPUT        t         0      0   0   2    4    0    0  COLUMN36
  81     128   H      OUTPUT        t         0      0   0   2    4    0    0  COLUMN37
  73     115   H      OUTPUT        t         0      0   0   2    4    0    0  COLUMN38


Code:

s = Synthesized pin or logic cell
t = Turbo logic cell
+ = Synchronous flipflop
/ = Slow slew-rate output
! = NOT gate push-back
r = Fitter-inserted logic cell
```

```
Device-Specific Information:                        c:\max2work\vhdl\luke3.rpt
luke3

** BURIED LOGIC **

                                          Shareable
                                          Expanders      Fan-In    Fan-Out
 Pin      LC  LAB  Primitive     Code   Total Shared n/a INP  FBK  OUT  FBK  Name
 (44)     65   E        TFFE   +  t         0      0   0    2    3    9    1  RAND13 (:33)
   -      66   E        TFFE   +  t         0      0   0    2    2    9    2  RAND12 (:34)
 (45)     67   E        TFFE   +  t         0      0   0    2    1    9    3  RAND11 (:35)
   -      81   F        DFFE   +  t         0      0   0    2    4    9    4  RAND10 (:36)
   -     100   G        TFFE   +  t         0      0   0    2    3    9    1  RAND23 (:37)
   -      95   G        TFFE   +  t         0      0   0    2    2    8    2  RAND22 (:38)
   -     103   G        TFFE   +  t         0      0   0    2    1    9    3  RAND21 (:39)
   -     102   G        DFFE   +  t         0      0   0    2    4    9    4  RAND20 (:40)
   -     113   H        TFFE      t         0      0   0    3    3    9    1  RAND33 (:41)
   -     122   H        TFFE      t         0      0   0    3    2    8    2  RAND32 (:42)
   -     116   H        TFFE      t         0      0   0    3    1    9    3  RAND31 (:43)
   -     119   H        DFFE      t         1      0   1    3    4    9    4  RAND30 (:44)


Code:

s = Synthesized pin or logic cell
t = Turbo logic cell
+ = Synchronous flipflop
/ = Slow slew-rate output
! = NOT gate push-back
r = Fitter-inserted logic cell
```

```
Device-Specific Information:                          c:\max2work\vhdl\luke3.rpt
luke3

** LOGIC CELL INTERCONNECTIONS **

Logic Array Block 'E':

                          Logic cells placed in LAB 'E'
        +---------------- LC77 COLUMN14
        | +-------------- LC75 COLUMN15
        | | +------------ LC73 COLUMN16
        | | | +---------- LC69 COLUMN17
        | | | | +-------- LC72 COLUMN18
        | | | | | +------ LC65 RAND13
        | | | | | | +---- LC66 RAND12
        | | | | | | | +-- LC67 RAND11
        | | | | | | | |
        | | | | | | | |   Other LABs fed by signals
        | | | | | | | |   that feed LAB 'E'
LC      | | | | | | | | | A B C D E F G H |     Logic cells that feed LAB 'E':
LC65 -> * * * * * * - - | - - - - * * - - | <-- RAND13
LC66 -> * * * * * * * - | - - - - * * - - | <-- RAND12
LC67 -> * * * * * * * * | - - - - * * - - | <-- RAND11

Pin
83   -> - - - - - - - - | - - - - - - - - | <-- CLK
2    -> - - - - - - - - | - - - - - - - - | <-- CLK2
12   -> * * * * * * * * | - - - - * * * * | <-- GO
11   -> * * * * * * * * | - - - - * * * * | <-- START
LC81 -> * * * * * * * * | - - - - * * - - | <-- RAND10


* = The logic cell or pin is an input to the logic cell (or LAB) through the PIA.
- = The logic cell or pin is not an input to the logic cell (or LAB).
```

```
Device-Specific Information:                              c:\max2work\vhdl\luke3.rpt
luke3

** LOGIC CELL INTERCONNECTIONS **

Logic Array Block 'F':

                        Logic cells placed in LAB 'F'
        +--------------- LC93 COLUMN10
        | +------------- LC83 COLUMN11
        | | +----------- LC85 COLUMN12
        | | | +--------- LC88 COLUMN13
        | | | | +------- LC91 COLUMN26
        | | | | | +----- LC86 COLUMN27
        | | | | | | +--- LC99 COLUMN28
        | | | | | | | +- LC81 RAND10
        | | | | | | | |
        | | | | | | | |   Other LABs fed by signals
        | | | | | | | |   that feed LAB 'F'
LC      | | | | | | | | | A B C D E F G H |     Logic cells that feed LAB 'F':
LC81 -> * * * * - - - * | - - - - * * - - | <-- RAND10

Pin
83   -> - - - - - - - - | - - - - - - - - | <-- CLK
2    -> - - - - - - - - | - - - - - - - - | <-- CLK2
12   -> * * * * * * * * | - - - - * * * * | <-- GO
11   -> * * * * * * * * | - - - - * * * * | <-- START
LC65 -> * * * * - - - * | - - - - * * - - | <-- RAND13
LC66 -> * * * * - - - * | - - - - * * - - | <-- RAND12
LC67 -> * * * * - - - * | - - - - * * - - | <-- RAND11
LC100-> - - - - * * * - | - - - - - * * - | <-- RAND23
LC98 -> - - - - * * * - | - - - - - * * - | <-- RAND22
LC103-> - - - - * * * - | - - - - - * * - | <-- RAND21
LC102-> - - - - * * * - | - - - - - * * - | <-- RAND20


* = The logic cell or pin is an input to the logic cell (or LAB) through the PIA.
- = The logic cell or pin is not an input to the logic cell (or LAB).
```

```
Device-Specific Information:                       c:\max2work\vhdl\luke3.rpt
luke3

** LOGIC CELL INTERCONNECTIONS **

Logic Array Block 'G':

                               Logic cells placed in LAB 'G'
        +--------------------- LC105 COLUMN20
        | +------------------- LC107 COLUMN21
        | | +----------------- LC97 COLUMN22
        | | | +--------------- LC109 COLUMN23
        | | | | +------------- LC104 COLUMN24
        | | | | | +----------- LC99 COLUMN25
        | | | | | | +--------- LC106 COLUMN35
        | | | | | | | +------- LC100 RAND23
        | | | | | | | | +----- LC98 RAND22
        | | | | | | | | | +--- LC103 RAND21
        | | | | | | | | | | +- LC102 RAND20
        | | | | | | | | | | |
        | | | | | | | | | | |   Other LABs fed by signals
        | | | | | | | | | | |   that feed LAB 'G'
LC      | | | | | | | | | | | | A B C D E F G H |     Logic cells that feed LAB 'G':
LC100-> * * * * * * - * - - * | - - - - - * * - | <-- RAND23
LC98 -> * * * * - * - * * - * | - - - - - * * - | <-- RAND22
LC103-> * * * * * * - * * * * | - - - - - * * - | <-- RAND21
LC102-> * * * * * * - * * * * | - - - - - * * - | <-- RAND20

Pin
43    -> - - - - - - - - - - - | - - - - - - - - | <-- CLK
2     -> - - - - - - - - - - - | - - - - - - - - | <-- CLK2
12    -> * * * * * * * * * * * | - - - - * * * * | <-- GO
11    -> * * * * * * * * * * * | - - - - * * * * | <-- START
LC113-> - - - - - - * - - - - | - - - - - - * * | <-- RAND33
LC116-> - - - - - - * - - - - | - - - - - - * * | <-- RAND31
LC114-> - - - - - - * - - - - | - - - - - - * * | <-- RAND30


* = The logic cell or pin is an input to the logic cell (or LAB) through the PIA.
- = The logic cell or pin is not an input to the logic cell (or LAB).
```

```
Device-Specific Information:                    c:\max2work\vhdl\juke3.rpt
juke3

** LOGIC CELL INTERCONNECTIONS **

Logic Array Block 'H':

                                  Logic cells placed in LAB 'H'
         +----------------------- LC125 COLUMN30
         | +--------------------- LC123 COLUMN31
         | | +------------------- LC120 COLUMN32
         | | | +----------------- LC118 COLUMN33
         | | | | +--------------- LC117 COLUMN34
         | | | | | +------------- LC126 COLUMN36
         | | | | | | +----------- LC128 COLUMN37
         | | | | | | | +--------- LC115 COLUMN3A
         | | | | | | | | +------- LC113 RAND33
         | | | | | | | | | +----- LC122 RAND32
         | | | | | | | | | | +--- LC116 RAND31
         | | | | | | | | | | | +- LC114 RAND30
         | | | | | | | | | | | |
         | | | | | | | | | | | |   Other LABs fed by signals
         | | | | | | | | | | | |   that feed LAB 'H'
LC       | | | | | | | | | | | | | A B C D E F G H |     Logic cells that feed LAB 'H':
LC113-> * * * * * * * * * - - * | - - - - - - * * | <-- RAND33
LC122-> * * * * * * * * * * - * | - - - - - - - * | <-- RAND32
LC116-> * * * * * * * * * * * * | - - - - - - * * | <-- RAND31
LC114-> * * * * * * * * * * * * | - - - - - - * * | <-- RAND30

Pin
83   -> - - - - - - - - - - - - | - - - - - - - - | <-- CLK
2    -> - - - - - - - - - - - - | - - - - - - - - | <-- CLK2
10   -> - - - - - - - - * * * * | - - - - - - - * | <-- CLK3
12   -> * * * * * * * * * * * * | - - - - * * * * | <-- GO
11   -> * * * * * * * * * * * * | - - - - * * * * | <-- START


* = The logic cell or pin is an input to the logic cell (or LAB) through the PIA.
- = The logic cell or pin is not an input to the logic cell (or LAB).
```

```
Device-Specific Information:                              c:\max2work\vhdl\luke3.rpt
luke3

** EQUATIONS **

CLK      : INPUT;
CLK2     : INPUT;
CLK3     : INPUT;
GO       : INPUT;
START    : INPUT;

-- Node name is 'COLUMN10'
-- Equation name is 'COLUMN10', location is LC093, type is output.
 COLUMN10 = LCELL( _EQ001 $  GND);
  _EQ001 =  GO &  RAND10 &  RAND11 &  RAND12 &  RAND13 &  START
         #  GO &  RAND10 & !RAND11 & !RAND12 &  RAND13 &  START;

-- Node name is 'COLUMN11'
-- Equation name is 'COLUMN11', location is LC083, type is output.
 COLUMN11 = LCELL( _EQ002 $  GND);
  _EQ002 =  GO & !RAND10 &  RAND11 &  RAND12 &  RAND13 &  START;

-- Node name is 'COLUMN12'
-- Equation name is 'COLUMN12', location is LC055, type is output.
 COLUMN12 = LCELL( _EQ003 $  GND);
  _EQ003 =  GO &  RAND10 &  RAND11 & !RAND12 &  RAND13 &  START
         #  GO &  RAND10 & !RAND11 &  RAND12 & !RAND13 &  START
         #  GO & !RAND10 &  RAND11 & !RAND12 & !RAND13 &  START;

-- Node name is 'COLUMN13'
-- Equation name is 'COLUMN13', location is LC088, type is output.
 COLUMN13 = LCELL( _EQ004 $  GND);
  _EQ004 =  GO &  RAND10 &  RAND11 &  RAND12 & !RAND13 &  START;

-- Node name is 'COLUMN14'
-- Equation name is 'COLUMN14', location is LC077, type is output.
 COLUMN14 = LCELL( _EQ005 $  GND);
  _EQ005 =  GO &  RAND10 & !RAND11 &  RAND12 &  RAND13 &  START
         #  GO & !RAND10 &  RAND11 &  RAND12 & !RAND13 &  START
         #  GO & !RAND10 & !RAND11 & !RAND12 &  RAND13 &  START;

-- Node name is 'COLUMN15'
-- Equation name is 'COLUMN15', location is LC075, type is output.
 COLUMN15 = LCELL( _EQ006 $  GND);
  _EQ006 =  GO & !RAND10 &  RAND11 & !RAND12 &  RAND13 &  START;

-- Node name is 'COLUMN16'
-- Equation name is 'COLUMN16', location is LC073, type is output.
 COLUMN16 = LCELL( _EQ007 $  GND);
  _EQ007 =  GO & !RAND10 & !RAND11 &  RAND12 &  RAND13 &  START
         #  GO &  RAND10 & !RAND11 & !RAND12 & !RAND13 &  START;

-- Node name is 'COLUMN17'
-- Equation name is 'COLUMN17', location is LC069, type is output.
 COLUMN17 = LCELL( _EQ008 $  GND);
  _EQ008 =  GO &  RAND10 &  RAND11 & !RAND12 & !RAND13 &  START;

-- Node name is 'COLUMN18'
-- Equation name is 'COLUMN18', location is LC072, type is output.
 COLUMN18 = LCELL( _EQ009 $  GND);
  _EQ009 =  GO & !RAND10 & !RAND11 &  RAND12 & !RAND13 &  START;

-- Node name is 'COLUMN20'
-- Equation name is 'COLUMN20', location is LC105, type is output.
 COLUMN20 = LCELL( _EQ010 $  GND);
  _EQ010 =  GO &  RAND20 &  RAND21 & !RAND22 &  RAND23 &  START
         #  GO & !RAND20 &  RAND21 &  RAND22 &  RAND23 &  START
         #  GO & !RAND20 & !RAND21 &  RAND22 & !RAND23 &  START
         #  GO & !RAND20 & !RAND21 & !RAND22 &  RAND23 &  START;
```

```
-- Node name is 'COLUMN21'
-- Equation name is 'COLUMN21', location is LC107, type is output.
 COLUMN21 = LCELL( _EQ011 $  GND);
  _EQ011 =  GO &  RAND20 &  RAND21 &  RAND22 &  RAND23 &  START;

-- Node name is 'COLUMN22'
-- Equation name is 'COLUMN22', location is LC097, type is output.
 COLUMN22 = LCELL( _EQ012 $  GND);
  _EQ012 =  GO &  RAND20 & !RAND21 &  RAND22 &  RAND23 &  START
         #  GO & !RAND20 &  RAND21 & !RAND22 &  RAND23 &  START
         #  GO &  RAND20 & !RAND21 & !RAND22 & !RAND23 &  START;

-- Node name is 'COLUMN23'
-- Equation name is 'COLUMN23', location is LC109, type is output.
 COLUMN23 = LCELL( _EQ013 $  GND);
  _EQ013 =  GO & !RAND20 & !RAND21 &  RAND22 &  RAND23 &  START;

-- Node name is 'COLUMN24'
-- Equation name is 'COLUMN24', location is LC104, type is output.
 COLUMN24 = LCELL( _EQ014 $  GND);
  _EQ014 =  GO & !RAND20 &  RAND21 & !RAND23 &  START;

-- Node name is 'COLUMN25'
-- Equation name is 'COLUMN25', location is LC099, type is output.
 COLUMN25 = LCELL( _EQ015 $  GND);
  _EQ015 =  GO &  RAND20 &  RAND21 & !RAND22 & !RAND23 &  START;

-- Node name is 'COLUMN26'
-- Equation name is 'COLUMN26', location is LC096, type is output.
 COLUMN26 = LCELL( _EQ016 $  GND);
  _EQ016 =  GO &  RAND20 & !RAND21 &  RAND22 & !RAND23 &  START;

-- Node name is 'COLUMN27'
-- Equation name is 'COLUMN27', location is LC086, type is output.
 COLUMN27 = LCELL( _EQ017 $  GND);
  _EQ017 =  GO &  RAND20 & !RAND21 & !RAND22 &  RAND23 &  START;

-- Node name is 'COLUMN28'
-- Equation name is 'COLUMN28', location is LC094, type is output.
 COLUMN28 = LCELL( _EQ018 $  GND);
  _EQ018 =  GO &  RAND20 &  RAND21 &  RAND22 & !RAND23 &  START;

-- Node name is 'COLUMN30'
-- Equation name is 'COLUMN30', location is LC125, type is output.
 COLUMN30 = LCELL( _EQ019 $  GND);
  _EQ019 =  GO &  RAND30 &  RAND31 & !RAND32 &  RAND33 &  START
         #  GO &  RAND30 & !RAND31 & !RAND32 & !RAND33 &  START
         #  GO &  RAND30 &  RAND32 & !RAND33 &  START;

-- Node name is 'COLUMN31'
-- Equation name is 'COLUMN31', location is LC123, type is output.
 COLUMN31 = LCELL( _EQ020 $  GND);
  _EQ020 =  GO & !RAND30 &  RAND31 &  RAND32 &  RAND33 &  START;

-- Node name is 'COLUMN32'
-- Equation name is 'COLUMN32', location is LC120, type is output.
 COLUMN32 = LCELL( _EQ021 $  GND);
  _EQ021 =  GO &  RAND30 & !RAND31 &  RAND32 &  RAND33 &  START;

-- Node name is 'COLUMN33'
-- Equation name is 'COLUMN33', location is LC118, type is output.
 COLUMN33 = LCELL( _EQ022 $  GND);
  _EQ022 =  GO &  RAND30 &  RAND31 &  RAND32 &  RAND33 &  START;

-- Node name is 'COLUMN34'
-- Equation name is 'COLUMN34', location is LC117, type is output.
 COLUMN34 = LCELL( _EQ023 $  GND);
  _EQ023 =  GO & !RAND30 &  RAND31 & !RAND32 &  RAND33 &  START;
```

```
-- Node name is 'COLUMN35'
-- Equation name is 'COLUMN35', location is LC101, type is output.
 COLUMN35 = LCELL( _EQ024 $  GND);
  _EQ024 =  GO &  !RAND30 & !RAND31 &  RAND33 &  START;

-- Node name is 'COLUMN36'
-- Equation name is 'COLUMN36', location is LC126, type is output.
 COLUMN36 = LCELL( _EQ025 $  GND);
  _EQ025 =  GO & !RAND30 & !RAND31 &  RAND32 & !RAND33 &  START;

-- Node name is 'COLUMN37'
-- Equation name is 'COLUMN37', location is LC128, type is output.
 COLUMN37 = LCELL( _EQ026 $  GND);
  _EQ026 =  GO &  RAND30 &  RAND31 & !RAND32 & !RAND33 &  START
         #  GO & !RAND30 &  RAND31 &  RAND32 & !RAND33 &  START
         #  GO &  RAND30 & !RAND31 & !RAND32 &  RAND33 &  START;

-- Node name is 'COLUMN38'
-- Equation name is 'COLUMN38', location is LC115, type is output.
 COLUMN38 = LCELL( _EQ027 $  GND);
  _EQ027 =  GO & !RAND30 &  RAND31 & !RAND32 & !RAND33 &  START;

-- Node name is ':36' = 'RAND10'
-- Equation name is 'RAND10', location is LC081, type is buried.
RAND10   = DFFE( _EQ028 $  GND, GLOBAL( CLK),  VCC,  VCC,  VCC);
  _EQ028 =  GO &  RAND10 &  RAND11 &  RAND12 &  RAND13 & !START
         #  GO & !RAND10 & !START
         # !GO &  RAND10 & !START
         #  RAND10 &  START;

-- Node name is ':35' = 'RAND11'
-- Equation name is 'RAND11', location is LC067, type is buried.
RAND11   = TFFE( _EQ029, GLOBAL( CLK),  VCC,  VCC,  VCC);
  _EQ029 =  GO &  RAND10 & !START;

-- Node name is ':34' = 'RAND12'
-- Equation name is 'RAND12', location is LC066, type is buried.
RAND12   = TFFE( _EQ030, GLOBAL( CLK),  VCC,  VCC,  VCC);
  _EQ030 =  GO &  RAND10 &  RAND11 & !START;

-- Node name is ':33' = 'RAND13'
-- Equation name is 'RAND13', location is LC065, type is buried.
RAND13   = TFFE( _EQ031, GLOBAL( CLK),  VCC,  VCC,  VCC);
  _EQ031 =  GO &  RAND10 &  RAND11 &  RAND12 & !START;

-- Node name is ':40' = 'RAND20'
-- Equation name is 'RAND20', location is LC102, type is buried.
RAND20   = DFFE( _EQ032 $  GND, GLOBAL( CLK2),  VCC,  VCC,  VCC);
  _EQ032 =  GO &  RAND20 &  RAND21 &  RAND22 &  RAND23 & !START
         #  GO & !RAND20 & !START
         # !GO &  RAND20 & !START
         #  RAND20 &  START;

-- Node name is ':39' = 'RAND21'
-- Equation name is 'RAND21', location is LC103, type is buried.
RAND21   = TFFE( _EQ033, GLOBAL( CLK2),  VCC,  VCC,  VCC);
  _EQ033 =  GO &  RAND20 & !START;

-- Node name is ':38' = 'RAND22'
-- Equation name is 'RAND22', location is LC098, type is buried.
RAND22   = TFFE( _EQ034, GLOBAL( CLK2),  VCC,  VCC,  VCC);
  _EQ034 =  GO &  RAND20 &  RAND21 & !START;

-- Node name is ':37' = 'RAND23'
-- Equation name is 'RAND23', location is LC100, type is buried.
RAND23   = TFFE( _EQ035, GLOBAL( CLK2),  VCC,  VCC,  VCC);
  _EQ035 =  GO &  RAND20 &  RAND21 &  RAND22 & !START;

-- Node name is ':44' = 'RAND30'
-- Equation name is 'RAND30', location is LC114, type is buried.
```

```
RAND30   = DFFE( _EQ036 $  GND,  CLK3,  VCC,  VCC,  VCC);
  _EQ036 =  GO &  RAND30 &  RAND31 &  RAND32 &  RAND33 & !START
         #  GO & !RAND30 & !START
         # !GO &  RAND30 & !START
         #  RAND30 &  START;

-- Node name is ':43' = 'RAND31'
-- Equation name is 'RAND31', location is LC116, type is buried.
RAND31   = TFFE( _EQ037,  CLK3,  VCC,  VCC,  VCC);
  _EQ037 =  GO &  RAND30 & !START;

-- Node name is ':42' = 'RAND32'
-- Equation name is 'RAND32', location is LC122, type is buried.
RAND32   = TFFE( _EQ038,  CLK3,  VCC,  VCC,  VCC);
  _EQ038 =  GO &  RAND30 &  RAND31 & !START;

-- Node name is ':41' = 'RAND33'
-- Equation name is 'RAND33', location is LC113, type is buried.
RAND33   = TFFE( _EQ039,  CLK3,  VCC,  VCC,  VCC);
  _EQ039 =  GO &  RAND30 &  RAND31 &  RAND32 & !START;


--     Shareable expanders that are duplicated in multiple LABs:
--     (none)
```

```
Project Information                             c:\max2work\vhdl\luke3.rpt

** COMPILATION SETTINGS & TIMES **

Processing Menu Commands
------------------------

Design Doctor                              = off

Logic Synthesis:

   Synthesis Type Used                     = Standard

   Default Synthesis Style                 = NORMAL

      Logic option settings in 'NORMAL' style for 'MAX7000S' family

      DECOMPOSE_GATES                      = on
      DUPLICATE_LOGIC_EXTRACTION           = on
      MINIMIZATION                         = full
      MULTI_LEVEL_FACTORING                = on
      NOT_GATE_PUSH_BACK                   = on
      PARALLEL_EXPANDERS                   = off
      REDUCE_LOGIC                         = on
      REFACTORIZATION                      = on
      REGISTER_OPTIMIZATION                = on
      RESYNTHESIZE_NETWORK                 = on
      SLOW_SLEW_RATE                       = off
      SOFT_BUFFER_INSERTION                = on
      SUBFACTOR_EXTRACTION                 = on
      TURBO_BIT                            = on
      XOR_SYNTHESIS                        = on
      IGNORE_SOFT_BUFFERS                  = off
      USE_LPM_FOR_AHDL_OPERATORS           = off

   Other logic synthesis settings:

      Automatic Global Clock               = on
      Automatic Global Clear               = on
      Automatic Global Preset              = on
      Automatic Global Output Enable       = on
      Automatic Fast I/O                   = off
      Automatic Register Packing           = off
      Automatic Open-Drain Pins            = on
      Automatic Implement in EAB           = off
      One-Hot State Machine Encoding       = off
      Optimize                             = 5

Default Timing Specifications: None

Cut All Bidir Feedback Timing Paths        = on
Cut All Clear & Preset Timing Paths        = on

Ignore Timing Assignments                  = off

Functional SNF Extractor                   = off

Linked SNF Extractor                       = off
Timing SNF Extractor                       = on
Optimize Timing SNF                        = off
Generate AHDL TDO File                     = off
Fitter Settings                            = NORMAL
Smart Recompile                            = off
Total Recompile                            = off

Interfaces Menu Commands
------------------------

EDIF Netlist Writer                        = off
```

```
Verilog Netlist Writer                     = off
VHDL Netlist Writer                        = off

Compilation Times
-----------------

   Compiler Netlist Extractor              00:00:00
   Database Builder                        00:00:01
   Logic Synthesizer                       00:00:00
   Partitioner                             00:00:00
   Fitter                                  00:00:00
   Timing SNF Extractor                    00:00:00
   Assembler                               00:00:03
   --------------------------              --------
   Total Time                              00:00:04


Memory Allocated
-----------------

Peak memory allocated during compilation  = 6,366K
```

APPENDIX F

THE DESIGN DELAY MATRIX

Delay Matrix

Destination

Source	COLUMN10	COLUMN11	COLUMN12	COLUMN13	COLUMN14	COLUMN15	COLUMN16	COLUMN17	COLUMN18	COLUMN20
CLK	9.5ns	9.5ns	9.5ns	9.5ns	9.5ns	9.5ns	9.5ns	9.5ns	9.5ns	
CLK2										9.5ns
CLK3										
GO	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns
START	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns

Delay Matrix

Destination

Source

	COLUMN21	COLUMN22	COLUMN23	COLUMN24	COLUMN25	COLUMN26	COLUMN27	COLUMN28	COLUMN30	COLUMN31
CLK										
CLK2	9.5ns	9.5ns	9.5ns	9.5ns	9.5ns	9.5ns	9.5ns	9.5ns		
CLK3									12.5ns	12.5ns
GO	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns
START	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns

Delay Matrix

Destination

Source	COLUMN32	COLUMN33	COLUMN34	COLUMN35	COLUMN36	COLUMN37	COLUMN38
CLK							
CLK2							
CLK3	12.5ns	12.5ns	12.5ns	12.5ns	12.5ns	12.5ns	12.5ns
GO	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns
START	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns	7.5ns